

# COMPUTER PROGRAMMING FOR BEGINNERS

5 BOOKS IN 1

PYTHON PROGRAMMING + SQL + ARDUINO +  
C# + JAVASCRIPT TO BECOME SKILLED QUICKLY



RYAN TURNER  
& CHARLES WALKER

JAVASCRIPT  
C#  
ARDUINO  
SQL  
PYTHON PROGRAMMING  
5 4 3 2 1

# **PYTHON PROGRAMMING:**

*The Ultimate Beginner's Guide to Master Python  
programming Step by Step with Practical Exercises*

**Charles Walker**

**© Copyright 2019 – Charles Walker**

**All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

**Legal Notice:**

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

# **Table of Contents**

[Introduction](#)

[Chapter 1: Getting Started](#)

[Why Python?](#)

[Installing Python](#)

[Summary](#)

[Chapter 2: Software Design Cycle](#)

[Solving Problems](#)

[Summary](#)

[Chapter 3: Variables and Data Types](#)

[Python Identifiers](#)

[Introduction to Variables](#)

[Python and Dynamic Typing](#)

[Basic Text Operations](#)

[Numbers](#)

[Summary](#)

[Chapter 4: Decision Making in Python](#)

[Conditional Statements](#)

[“While” Loops](#)

[“For” Loops](#)

[Summary](#)

[Chapter 5: Python Data Structures](#)

[Sequences](#)

[Creating a Basic Game](#)

[Summary](#)

[Chapter 6: Functions](#)

[Creating Functions](#)

[Global Variables](#)

[Handling Global Variables](#)

[Writing Tic Tac Toe](#)

[Summary](#)

[Chapter 7: Introduction to Object Oriented Programming](#)

[Classes](#)

[Python Inheritance](#)

[Polymorphism](#)

[Creating Modules](#)

[Summary](#)

[Chapter 8: Exceptions](#)

[When Things Go Haywire](#)

[Summary](#)

[Bibliography](#)

## Introduction

Python is one of today's most powerful and beginner friendly programming languages. In recent years it has gained more ground as the technology of choice for fields such as Machine Learning, Deep Learning, and Data Science. There's never been a better time to start learning how to program with Python.

The purpose of *Python Programming: The Ultimate Beginner's Guide to Master Python programming Step by Step with Practical Exercises* is to make this amazing programming language as available as possible, no matter your current skill and knowledge of programming. With the help of this guide you will get started in no time, even if your knowledge on the topic is nonexistent.

Every chapter in this book is layered in such a way to teach you every core programming concept step by step. Furthermore, you will not learn only theory and challenging definitions that make you fall asleep. You will go through a number of practical examples and exercises in parallel and practice everything you learn. Keep in mind that without practicing every concept and writing your own code you will not manage to get too far. Programming doesn't require any special talents. All you need is practice, practice, and more practice. So let this book guide you, and make sure you take a break every now and then to experiment and come up with your own little programs. Python is the future!

# **Chapter 1: Getting Started**

Programming is becoming an increasingly demanded skill for anything from web design to Machine Learning and the Internet of Things. It's on its way to having a daily use due to the importance of technology. While programming used to be a subject that people started studying for their computer science degree, now it is often taught starting from elementary school. One of the main reasons for its widespread use is accessibility. You don't need much to get started. Thanks to the power of the Internet, all you need is a computer and a number of software tools which you can download and install without spending a penny. In addition, there are many resources to learn from, as well as organized communities you can join and learn from.

In this chapter you are going to learn why Python is one of the best programming languages to start with, as well as progress your career if this isn't your first language. Furthermore, you will explore the tools you need, install them, and start your journey. This chapter will guide you step by step and show you everything you need to know in order to get started. If you are already familiar with any other programming language such as C, C++, or Java, you might want to skip this chapter or simply glance through it to refresh your memory.

## **Why Python?**

Python is a versatile and powerful programming language that was developed in 1991 by Guido van Rossum. As a fun fact, you should know that the name of the language doesn't come from the snake which bears the same name. Guido named his project "Python" after Monty Python, which was a British comedy group he was a big fan of. If you happen to be a fan as well, you will find several "Easter Eggs" within the official documentation of the language.

Since 1991, Python has been used to introduce people to programming due to its simple syntax, as well as to create complex programs or analyze massive amounts of data. As a beginner, with Python you will be able to quickly write a basic program. However, you can easily scale it further and turn it into a commercial project. The main reason why Python is so popular

for beginners is the fact that the language is easy to read and write. Its structure is human-like and easy to understand, therefore the code is very user-friendly. This means that you shouldn't find it too difficult to remember the language and structure. In addition, Python comes with a number of libraries and premade functions that you can immediately add to your code. This way you can save time. In many ways it's like playing with Legos. As long as you pace yourself, learn and practice everything in this book, and extend your knowledge using other resources, you will be able to write a program that you will understand ten years from now. Program maintenance is a crucial part of your responsibilities as a programmer, but luckily Python code is easy to administrate compared to other languages. With that in mind, let's briefly explore the plethora of reasons why you should learn Python instead of any other languages. After all, Python isn't quite the only language that offers you the advantages you've learned about so far.

1. User-friendly: The purpose of a programming language is to form the connection between human and computer. Python, like C# and Java, is a high level programming language, which means that it is quite far from the machine language which the computer then processes. The opposite of this is the low level language, which usually refers to assembly language or machine code. In other words, Python is close to English. This allows you to write code as fast as you write any sentence, once you learn the rules and the syntax.
2. Powerful: Sometimes Python is looked down upon because it is so easy to learn and it's usually the first language programmers explore, whether on their own or at computer science 101. However, Python is a very powerful language that is just as versatile and efficient as more complex languages such as C++. Python is used in every technical department in companies like Google, Microsoft, IBM, Xerox, NASA, and many more. You can even use Python in game development if you prefer to practice a programming language in a more artistic way.
3. OOP: Object oriented programming is many times the optimal computer problem solver. It is a methodology that offers a method of defining data and actions as objects. This type of programming is not

always necessary, however, when working on large applications it is usually the most optimal approach. For instance, programming languages such as C# and Java are object oriented. Python can be considered an object oriented language as well, however this feature is optional. The other mentioned languages don't offer such versatility. This means that with Python you don't necessarily have to learn the object oriented methodology from the start. This is one of the reasons why it's so much easier to start programming with Python than C++. However, you have the massive benefits of OOP at your fingertips, but only when you actually need it. If you are working on a basic program, there's no need for it. Python offers you all the power and versatility you need.

4. Computer-friendly: You can run Python on any kind of computer. You don't need a powerful computer processing unit and a great deal of RAM to start programming. You can even use a credit card-sized computer like the Raspberry Pi. In fact, Python requires so little that it is one of the top languages used in creating little robots that are operated by \$5 computers. In addition, Python runs on any operating system, whether it's Linux, Windows, or Mac. The programs you write do not depend on the platform. You can work on an application on your Windows running computer and then switch it to your Mac. For instance, if you finished creating a program and you need beta testers, you can email your project to a friend that uses Linux and another one with Windows. The program will work.
5. Language adaptability: If you ever write a program in another language, you can integrate Python within it. In other words, you can use Python on a program that was written in Java. In addition, you also combine Python with another language in order to take advantage of the benefits that are offered by both of them. For instance, you can integrate C or C++ in order to benefit from the system optimization and speed that they offer.
6. It's free: Everyone likes free stuff and Python won't cost you a cent. You can always download and install it for free as many times as you want. In addition, Python is an open source language, which means that the license even allows you to make modifications to the source

code. This means that you can modify Python and then sell your own version of it. You might not be interested in these features at this point, but it is one of the reasons why it's such a popular language.

7. Community: Being a powerful and versatile open source programming language brings the benefit of community. There are many online communities dedicated to teaching and learning everything there is to know about Python. You can ask questions on online boards or seek the advice of a master programmer. You can also seek fellow students and work on a project together. Python's popularity has gathered a massive crowd around it and you should take advantage of it.

## Installing Python

Before you can start programming, you need to download and install Python on your machine. The installation is quite straightforward no matter what operating system you're running, however you do need to pay attention to a couple of things.

First, you need to head to Python's homepage at [www.python.org](http://www.python.org) and head to the "Downloads" section. There you will see a number of different installers and each one of them has a different version. Make sure to download the right installer that matches your computer's operating system and select the latest version.

Once the download is complete, run the installer and follow the steps. You should simply accept the standard settings and once the installation is complete, you're ready to go.

If for some reason you don't want to install Python, you may notice that on the website's homepage you have some kind of a console. This is a Python online console and you can use it to practice your coding skills, or to try out some of the examples in this book. It's advisable for you to type the code yourself, even if you copy it from the book, and then try to be creative with it. You need to practice in order to memorize the syntax and specific commands, and the online console is really handy for a quick practice session.

## ***Using a Text Editor***

Python programming can be done with nearly any kind of plain text editor. You can use programs like Notepad, Notepad++, gedit, and many more. Keep in mind that some of these text editors come with a variety of features that are useful to programmers. For instance some of them, such as Notepad++ offer syntax highlighting which will instantly show you any errors you made. If you type code in a basic editor like plain Notepad, the program won't tell you when you've forgotten a semicolon or if you added additional space. There are many programs to choose from, so pick any editor you feel comfortable with.

With that in mind, avoid using word processors such as Microsoft Word or Open Office. They aren't good for programming purposes. They can be used to write code, however the problem is that when saving it the program will sneak in some additional lines of code by itself. That code is specific to the word processor and it can impact your program's speed, or even worse, it will simply not run.

## ***Using an IDE***

An IDE, which stands for Integrated Development Environment, is a program designed with a number of features that are useful to programmers. It has a graphical interface and it makes typing code much faster due to autocomplete and history functions. Programming stays the same whether you are using a text editor or an IDE, however with the IDE you will benefit from many shortcuts, reminders, and error signaling and code autocorrect. Many IDE's even include suggestions on how to fix an error.

There are many IDE's to choose from, but one of the most popular ones is IDLE. It comes in the same package as Python, so there's no need to perform any extra steps. Keep in mind that it can run in two modes, namely interactive and script. Use interactive if you want Python to immediately respond to whatever commands you type.

## ***Your First Program***

Now that your toolkit is prepared, it's time to write your first program. For this example we'll use IDLE because it's important to get used to IDE's

from the start in order to avoid any future frustrations. If you prefer to use a text editor or the online Python console, go ahead, the code will work the same.

Now, start running IDLE in interactive mode. You will now see a window that is known as a Python shell. At the command prompt type the following line:

```
print ("Hello World!")
```

Now you should see the result displayed on your screen as the following:  
Hello World!

That's it! Congratulations, you can call yourself a programmer now. Now let's discuss this bit of code briefly. The first thing you'll notice is that Python code is plain English, easy to read and understand. Even without programming knowledge, you probably knew what this line of code would do because it's self-explanatory. That's the beauty of working with Python.

As for the command we used, "print ()" is a function that displays the text which is written in the parentheses. Keep in mind that the line needs to be surrounded by quotation marks, otherwise you'll get an error. Furthermore, pay attention to how you type the function because in Python everything is case sensitive. The command "print" will work, however if you type it as "Print" it will not.

Now, let's create the same program but this time by using IDLE's script mode. Don't forget that interactive mode gives you instant results. It works the same as the online Python shell. However, you won't be able to save your program so that you can continue working on it later. In order to save it and edit it later, you need to work in script mode. You can run IDLE in script mode simply by clicking on "File" and selecting "New Window". Now type the same line again:

```
print ("Hello World!")
```

Hit the Enter key. You'll notice that nothing happens. That's because you are writing a list of instructions that will be executed at a later date when you run the program. First, you need to save the application by clicking on "Save As" from the "File" menu. You'll notice that by default the file has

the “py” extension. Always make sure your scripts are saved this way in order to be recognized as Python programs. Now if you run the program IDLE will open the interactive mode window and display the result.

For now, you’ve run your “Hello World” program by using IDLE. However, you normally want your applications to run like the ones you are currently using. This means you want an executable file which you double click and it runs. At the moment, if you click on the Python file a window will open and then abruptly close. You may be thinking that the program doesn’t work because nothing happened, however something did happen. It was simply too fast for you to observe anything concrete. The program executed all of its instructions, which means that it displayed the message in a fraction of a second and then it terminated itself. What you need to do is keep the program running once it executes all of its commands so that you can see the results and interact with them. But before you do that, let’s take a moment to discuss how to comment your code and make it readable and easy to understand.

## ***Code Comments and Your Program***

Open your script and type the following lines:

```
# Hello World!
```

```
# This is a demonstration of the “print” function.
```

If you run the program again, you will see that nothing changed. These lines you added aren’t executed as code. They are known as comments and their purpose is to make the code of an application more understandable. You might be thinking that typing such information is a waste of time because as the programmer you already know what your code is about. That may be true, however when you write a complex program and then you abandon it for a week or two, you’re going to have some trouble understanding the purpose of every function and variable. Sure, you can read your code and eventually figure everything out, but that is not a proper use of your time. Code comments are used to label and explain complicated functions so that you don’t have to dive into the code itself. They are especially useful if another programmer is going to work on your program at a later date. Imagine a stranger having to decipher your personal approach to the

development of your application. On a large project he could waste wakes of his time instead of doing some work in order to progress.

Comments are defined by the hash mark in front of a line. Each line you intend as a comment needs to have its own mark, otherwise you will get an error. If you're worried about the efficiency of your programs due to hundreds or even thousands of comments, you shouldn't be. They have no impact on your computer because when the code is executed, the machine ignores all comments and uses no additional resources.

Additionally, to make your comments and code more readable, you can leave empty lines. However, don't do this after every line of code. You use empty space in between blocks of code, or sections. Programs ignore blank space, so nothing will be affected by using it. Now let's get back to your first program. Add the following line after the print function:

```
input ("\n\n Hit the Enter key to exit!")
```

This line will display the console in which the line “Hello World!” is printed, and then display the line “Hit the Enter key to exit!” Finally, the program will stay open and wait for you until you hit the Enter key. This is a simple way to keep the program running until the user performs an action.

## Summary

In this chapter we've gone through an introduction to Python. You learned the strengths of this programming language and what you can do with it. Furthermore, you installed Python on your computer and played with it briefly by creating the very first program that every future programmer builds. You learned about using Python's shell or an IDE and how to immediately test your code after writing it. Now you know how to use the basic tools needed to create a program and you learned the importance of commenting your code. You are ready to begin your Python learning journey, so let's add more tools to your development kit!

# **Chapter 2: Software Design Cycle**

Before we continue exploring Python's syntax, data types, and functions, we are going to take a side step and learn about the design cycle of a program. This topic is often left out from most books and classes dedicated to programming, which is a mistake. Knowing how to design and develop a program from beginning to end is a skill that all programmers should possess. Furthermore, if you're interested in a career as a programmer you should know that questions related to this topic come up frequently in an interview.

In this chapter you are going to learn how to take a problem and figure out how to create a program that solves it. The very first stage of the development cycle is handling the design aspect of an application. You will learn how to turn all of your ideas into small, achievable goals, explore all the problems you need to solve, and develop a number of actions that yield the best result. In the first section of this chapter you will identify the problem and start designing your solution. All the questions you will ask yourself during this stage will result in all the information you need to design a rough program which you can later optimize and refine. In the second part of the chapter, you will transfer the design into actual code, and once the code is written you can go through a testing and refining loop until you achieve an error-free result.

Furthermore, you will learn the technique of using pseudo code, which is known as a design language. It's not actual code, however it serves as a guide which shows you which goal you need to achieve next and how you intend to do it. This is another topic frequently missing from other learning materials, but vital to turning you into an efficient and organized programmer.

## **Solving Problems**

Programming is all about solving problems in the most logical manner. You are going to write a number of programs while practicing what you learn

from this book as well as other sources, and then you are going to develop them on your own. Whether you create applications for fun or as a profession, you do it in order to solve some kind of problem because there isn't any other application that suits your needs. Even if you create a clone of another problem, you can still add your own touch to it in order to improve it to your taste. You probably already know what problems you want to solve since you took the major step of picking up a book about programming. Perhaps you have an amazing idea for a game and you want to share it with the world. Or maybe you thought of an app that offers a service nothing else does. Perhaps you're into robots and you want to build an army of them to take over the world (Python can totally help with that). No matter your goals, the first step is to start asking questions. But why not just jump in straight to writing code? Because programs are complex and you can easily get lost in the development process. For instance, imagine J. R. R. Tolkien writing *Lord of the Rings* without asking himself a number of questions about the world, its culture, and its inhabitants. How could he create something so vast and complex without an outline? Programs work the same way. If you sit down to write the design you will work on your project much faster, with fewer problems along the way, and have an easier time troubleshooting and maintaining the code.

“Why should I?” is probably the first question that will pop in your mind when you look at your project idea, or function, module, etc. Nowadays there's a program for nearly everything you can think of, which means you can find something similar and add to your project without writing everything from scratch. Finding code written and shared by someone else is quite easy and you will often think about adapting it.

The next question that will spring to mind will be “Is there an easier method?” Most programmers don't like starting out with nothing and memorizing everything there is to know about their chosen programming language. Luckily, programming isn't about learning all the theory, definitions, and code. Simplification is key. Most programmers apply the KISS principle within their design process, which appropriately stands for “Keep it simple, stupid.” Your first hurdle is finding the path of least complexity on your way to your goals.

## ***Problem Identification***

Before you get to coding, you need an idea. The easiest way to come up with one is to think of a task you regularly perform. Anything that's repetitive could be automated by writing a script. Once you have the idea, you are confronted with your first problem. You need to find a way to communicate a number of instructions to your computer by using Python and then receive information in return. With this in mind, let's go through the basic development cycle of a program. Just keep in mind that you don't have to follow any kind of rigid structure. You can add your own personal touch to the outline.

Without a structure to start with, the start can be difficult and adding scripts to your project as you go can become a messy experience. Fortunately, you don't need to be a professional developer to perform the next steps:

1. **The software's purpose:** Start planning your project by writing down what problem you want to solve and what your program needs to be capable of. Since you may have very little programming knowledge at this point, you can start off with some very basic tasks. For instance, let's say you need a program that prints some messages to the user. Now that you know the problem, you can start brainstorming for solutions.
2. **The type of user:** The next step is to figure out who your target audience is. Even if you're working on a personal project to get some practice in, you should go through every step. So, who else besides you is going to use this program? Your friends and family? Customers? How much will the user interact with the program? Does the user need to have technical knowledge in order to use it? All of the answers you come up with will define the design of your program. If your users are regular people that just know how to browse the Internet, watch videos, and type in a word processor, you will have to limit the design to fit the intended audience.
3. **The computer system:** The final aspect you need to consider is the operating system your intended audience is most likely to use. You may need to make modifications and perform additional testing if you intend for your program to run on multiple systems such as Linux and Windows. You might also have to consider what other kind of devices

or software they might be using. At this point you should figure out these problems at a general level to get an idea of what you may be facing down the line during the development process.

All of these questions will influence the decisions you make regarding the design of your program. By answering them now, you'll make your journey a lot smoother by knowing what to expect. On the other hand, if you can't answer some of these questions at the moment, it's perfectly alright. You'll find some of them as you start developing the project and getting a better idea of how it's going to turn out.

## ***Finding the Solution***

Now that you have some information, you can continue the design process. Don't worry if you feel it's taking more time than it should. Many developers spend more time designing their programs than actually coding them. The next step is to add more information about what the intended user's requirements are. Once you've figured out the audience you need to understand their needs. As a professional developer, this stage would involve the agreement between you and the client. For now, you only need to look at the problem and make sure you understand everything that you need to resolve. Sometimes the solution isn't that obvious and you're going to encounter complications along the way. However, if you wrote as many details as possible in human language, you will have an easier time transferring that information to code and avoid potential issues. Here are some of the things you need to consider at this point:

1. Functional requirements: In other words, you are looking for everything your program needs to do. For instance, you need your software to print a message such as "Hello World!" By taking the first program you developed earlier as an example, you'll realize that its main functional requirement is that the system prints a message to the display.
2. Results: At some point you need to test your program to see if it's behaving as expected. In the beginning, it might be enough to just run the application and make sure it works, however as it develops and it becomes more complex you will need to improve your testing

technique as well. In most cases it will be enough to create tests with predetermined data to see how the program behaves and if it gives you expected results. In other cases it will be enough to simply write a table with values and then see if the program's results match those values during each operation. Generally, you'll want to test your software as the development progresses. Don't wait until your project is finished. If you do, when a test reveals an error of some kind or it gives you confusing results, you will spend a lot of time figuring out what causes the issue. However, for now it's probably enough to just run the program and see if it gives you the output you planned for.

3. Maintenance: When you design a program you need to think whether it's going to be used multiple times or for an extended period of time. Software that is intended to be used regularly for several years will require support and maintenance. Generally there are two things you need to consider when it comes to maintenance. You need to plan for updates, changes, and extension and find a way to track any reported issues. Additionally, if you're working on a web-based application, you'll need to find a way to integrate your changes while the program is still working. In order to make maintenance as straightforward as possible, you should focus on writing the program in such a way that anyone can understand it at a glance. Many developers who created the program are also the ones who maintain it and update it throughout its life cycle. However, in some cases you will have other people handling this step for you, or at the very least contributing to it to some degree. Furthermore, keep your code even clearer by not repeating yourself. One of the unwritten rules in programming is that if you have to repeat some information more than once, you need to take a break and look at the design of the program. Remember, simplification makes everything much easier, including maintenance.

Here's an example of a software design document based on your "Hello World" program:

Problem: The system needs to display a message to the user's display.

User: Me

Operating System: Windows / Linux

Interface: Command line terminal

Functional requirements: Print a line of text

Testing: Perform application run test, making sure text is displayed.

## ***Start Designing***

Now you are ready to take the first practical step of designing your application. The easiest and fastest approach is to write the program in pseudo code, which you can later translate to actual programming code. Pseudo code involves using your own language to express a series of logical statements and actions that your program needs to execute in order to provide the user with the expected results. In addition, it allows you to brainstorm your approach to coding for this specific project. For instance, it can help you understand what functions, data types, and variables you need to write for the program to work.

Keep in mind that for now your pseudo code can be written any way you want in your own words. However, when you work with other programmers, you will need to follow a standard that everyone uses. But how's this standard created? Usually, pseudo code is in fact the same as Python code when it comes to syntax and rules, however it's not written through a series of commands and instructions, but words. For now, there's no need to worry about such standards. You first grasp the idea behind the structure. Here's how you can write pseudo code:

```
# Start program  
# Enter two values, X and Y  
# Add the values together  
# Print sum  
# End program
```

You'll notice that all the lines are commented out so that you don't get any errors when Python tries to execute the code. In this example, we have the main design of our new program. It's a basic description of all the steps that you need to take. When you perform one of the steps with actual code, the program will execute the statement.

## ***Improving Your Pseudo Code***

As mentioned earlier, readability is one of the most important factors when writing code. This is just as valid when writing pseudo code. One of the best ways to improve your code is by using indentation. Keep in mind however, that Python sees both spaces and tab as indentation and they are interpreted in different ways. They are not interchangeable, so don't mix them up. For instance, you can leave an entire blank line to separate code blocks or sections so that you identify each function without straining your eyes. The same goes for the pseudo code. Indentation can be used to determine which logical statements or actions belong together.

Keep in mind that when you're writing pseudo code, you need to always comment it out. This means that you can use any type of indentation you prefer because it will not affect the code. However, indentation is also used on the actual code and there are certain rules that you need to follow, otherwise your program will be unresponsive or you will encounter errors. You will learn more about this topic in a later chapter. For now you should focus on design and pseudo code alone.

The first line of a code block can be written without any indentation. However, every action that belongs to it should be indented into subsections. This way you'll know how to follow each step logically when writing the Python code. For now, you can make any structure you want, however as you study Python code and syntax you will start writing the pseudo code by following the same rules. Now let's take a look at an improved block of pseudo code:

```
# Perform the first task  
# Enter the first value  
# Verify whether the value is text  
# Print the value
```

As you can see, using indentation when writing pseudo code makes your objectives much clearer. Next, you can add a blank line to create a separation between this block and the next one, and write it the same way.

That's it! Now you can start coding your design by following each objective that you laid out for yourself.

## **Summary**

In this chapter you learned about the software design cycle and how to start the process of creating a Python program. During the process you've also recognized ways to spot problems and analyze them by splitting them into a collection of objectives and tasks. Furthermore, you learned the importance of pseudo code in the developmental process and how to structure it for maximum readability. The purpose of this chapter was to educate you on the importance of planning and outlining your ideas in order to avoid problems and confusion when writing your program. Now you are prepared with the knowledge and the tools to learn more about Python itself and its syntax.

# **Chapter 3: Variables and Data Types**

Variables are used to store data values that the computer processes. Each value can be a different data type, such as an integer, list, string, dictionary, etc. In this chapter you will learn about a number of variables that you can use to create an application. They are crucial to any program, which means that you can't really write code without using them.

Furthermore, you will work with real examples in order to learn how you can change text strings or perform mathematical operations. In programming, many tasks are solved with the help of variables. For instance, you can determine an operation for achieving a result without knowing what values certain variables refer to in advance. Any kind of data that you introduce to your computer is in fact converted into a variable, which you can then use as part of a program that has a number of functions.

## **Python Identifiers**

When you write code you need to be able to clearly understand and identify every type of data by just looking at its identifier or label. You can define identifiers as words that you did not comment out or type in-between quotation marks to turn it into plain text. Identifiers are used for naming purposes in order to make the code readable, provided that you use appropriate names related to your project and variables. Just make sure you don't attempt to use the same identifier for multiple variables and functions, otherwise you will cause conflicts in your code that lead to chaotic results. Always focus on using unique identifiers.

Keep in mind that you can use pretty much any word you can think of, however there are a number of words that are reserved by Python and you cannot use them as identifiers. They are part of the programming language itself and they can't be used as labels. These words are known as keywords. Here are a number of such words: False, lambda, class, import, global, while, yield, continue, and so on. Fortunately, you don't have to start learning all of them right away. Through practice you will eventually memorize them, however, until then Python will help you. If you use one of the keywords as an identifier, Python will warn you and ask you to change the name.

With that in mind, let's see precisely how you can name your variables. When you label a variable you should always start with an underscore or a letter. This isn't a rule set in stone, however it's a generally accepted naming convention which most programmers follow. Keep in mind that you can also use numbers in your identifier as long as it doesn't begin with one. Furthermore, Python is a case sensitive programming language, therefore you should always pay attention when using lowercase or uppercase letters because they are different. For instance, you can name a variable "newVariable," "NewVariable," "NEWVARIABLE," and "\_newVariable," and they will be treated as four entirely unique variables. Additionally, you should stick to one of these naming methods throughout your project. If you choose to name a variable using an underscore, you should label all the other variables the same way. This will prevent any kind of confusion when you and any other programmers go through the code.

## **Introduction to Variables**

In most programming languages, defining a variable is done in two stages. First is known as the initialization stage, and it involves creating a container which you label with an appropriate identifier. The second stage is the assignment, and it involves placing a value. Both stages are in fact performed in one single operation. Here's how it looks:

```
variable = value
```

As you can see, both the initialization and the assignment are done by using the equals sign.

The block of code that achieves something is referred to as a statement. An example of this would be the assignment. On the other hand, we have the expression, which is the code that you can evaluate in order to obtain a value. Here's an example where we have an expression in a list of assignment statements.

```
number = 0
```

```
product_weight = 2.1
```

```
price_per_kilogram = 10
```

```
filename = 'file.txt'
```

```
trace = False  
  
sentence = "this is an example"  
  
total_price = product_weight * price_per_kilogram
```

When programming, you need to be as orderly as possible. Write every statement in a line. In this example we have a simple list. It looks very similar to a grocery list. When you write a program, one of the first things you need to do is define the variables you are going to use. Once they are defined, you can perform various tasks with them.

For now you should know that variables come in different shapes. They can be strings (text), lists, numbers, dictionaries, or Booleans. These types make up a program. They are the ingredients to your recipe. However, before we start exploring each type of variable, let's take a side step and discuss dynamic typing.

## **Python and Dynamic Typing**

In Python, the value of a variable is automatically determined, which means you don't have to define it yourself when writing the code. This is referred to as dynamic typing. Keep in mind that it has nothing to do with your typing skills. Other programming languages require you to declare whether your variable is a number, list, or dictionary. This means that you can focus on your code without thinking about what type of data you're using. Your only concern is how you use it and making sure that it does what you need it to do.

Keep in mind that without being forced to declare a variable, you might sometimes make the mistake of adding various variables throughout your code without even realizing it. The disadvantage of dynamic typing is the fact that Python won't alert you, unless you didn't declare any value for one of the variables. Beginners often struggle with keeping track of their variables and where they assigned their values. Fortunately, there are two things you can do to avoid this time consuming situation when you have many variables. The first thing you should do is declare the values at the beginning of a code block where you will use them. This way you will maintain all of your variables together with their values in one section. The second option is developing your design document. If you are working with

a large number of variables, you can create a table where you declare all of them together with their values. This way you can keep track of them at all times. Knowing your variables inside out with these two methods will allow you to later solve any errors and maintain your code without the frustration that involves dissecting your code.

Another advantage of dynamic typing is that it allows Python to flag any variables that are used with the wrong type of value. If you perform an invalid operation on them, you will receive a “`TypeError`”. Here’s an example:

```
x = 2
```

```
y = 'text'
```

```
trace = False
```

```
x + y
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
y - trace
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for -: 'str' and 'bool'
```

While you might not understand some of the code yet, you will notice that the `x` variable is a number and the `y` variable is text. After declaring them we attempted to calculate the sum of those two variables. The result of course is an error. You can’t do mathematical operations between words and numbers. You can add words or even sentences together or perform arithmetic on numbers, but you can’t mix them up. Therefore, in this example you’ll see something that’s called a traceback. This is Python letting you know that there is an issue with the logic behind the program. Keep in mind that a traceback is just one of the `TypeError`s Python can alert you about. This type of alert tells you that you need to go back to your code and rewrite it in order to make the operation possible. You need to make sure the data types represent real data that can be used together with other data without breaking any of the real world rules.

In the above example we have an integer (int) type variable which means whole numbers, and a string (str) type variable, which means text. These are the most basic types of data and you will use them often in any project. Python does give you the ability to combine these variable types to form more complex data types, however we'll get to them in a later section. For now you need to build your programming foundation by assigning and manipulating different types of basic information.

## Basic Text Operations

Strings are one of the most common data types in Python and in programming in general. You already used it when creating your first program. In the “Hello World” example you learned how to print a line of text to your display by using the print functions. Keep in mind that you can use that same function to also print a variable’s assigned value, as well as strings of characters. In your program, however, you had one single print statement. Now the question is, how do you print multiple statements in a single line? If you go with the basic method you used in your program, you will find out that you need to print each statement in its own line. Let’s introduce you to the concept of string concatenation. Concatenating in this case simply refers to joining strings of text together. You can do this several ways. Here’s one example that is done by separating the values with commas:

```
characterClass = "warrior"  
characterGender = "male"  
print (characterGender, characterClass)
```

The output will be “male warrior”.

There are other ways to join bits of text into one line. For instance, you can even eliminate the commas if you are concatenating the strings themselves instead of variables that represent them. Here’s how:

```
print ("male" "warrior")
```

The output will be “malewarrior”. Keep in mind that when you use this method, you need to include a space within the string itself if you need it,

otherwise the two strings will be combined without one like in this example. Here's how to add the space:

```
print ("male " "warrior")
```

Now the result will be “male warrior”. However, if you try to concatenate a string with a variable this way, you will have a problem. Here's what happens:

```
print (characterGender "warrior")
```

The result will look something like this:

File "<stdin>", line 1

```
print (characterGender "warrior")
```

```
^ SyntaxError: invalid syntax
```

As you can see, you get a syntax error if you try to concatenate a string to a variable without any kind of separation. You can only write two separate strings into one with this method because it's basically the same as writing one string.

Another method of combining strings is using an operator, namely the plus sign. In this case it's referred to as a concatenation operator and it works similarly to an arithmetic operator that is used in an equation. Here's an example:

```
x = "male"
```

```
y = "warrior"
```

```
x + y
```

The result will be “malewarrior”.

Keep in mind that since the plus sign is used to solve mathematical problems, if you concatenate strings this way you will force Python to work harder behind the scenes. This means that if your project is fairly large and you perform many string concatenations using this operator, your program will demand more resources from your computer. This is a perfect example that shows how you need to think about the design of your project before

coding it. If you need to concatenate many strings, you should use any of the other methods that don't involve an operation like this.

## Quotes

In Python, quotes serve an important purpose. As you probably concluded from all the previous examples, they are used to mark characters and words as text. A character is defined as any single letter, punctuation mark, number, or even space. The text is in fact a string of characters, hence the variable type called “string”. In order to separate the text from Python commands and keywords, you need to enclose it in quotes. So far, you have used the double quotation marks, however in Python you have more than one way of achieving the same effect.

You can enclose a single line string with double quotes as you did in your “Hello World” program, or you can use single quotes instead. Here are the two examples:

```
print ("double quote string")
```

```
print ('single quote string')
```

Both of these examples are read the same way. Python doesn't see a difference between the two methods, therefore you can use whichever you prefer. But what do you do when you have a block of text as a string and it takes a large number of lines? In Python you can use block quotes that will include multiple lines, including breaks and other empty spaces. Here are the two options you have:

```
x = "" this is a long line of text
```

```
that requires multiple lines""
```

```
y = """ this is another long line of strings
```

```
but this time we're using double quotes
```

```
because they're cooler """
```

As you can see, both methods achieve the same thing. Like with the single line example, you can either use double quotation marks or single quotation

marks. It's all about preference as long as you use three of them where the string starts and three more where the string ends.

Now the question is, what are you going to do when you have to use actual quotes within a string of text? Simply put, you have to use different quotation marks. For instance, you can use double quotes to mark the string and inside it you use single quotes to mark the real quote. It would look something like this:

```
x = "She told me to 'quote' like this"
```

In this example it looks like we have quoting hierarchy where the double quote marks the end of the string and not the single quote. The text between single quotes becomes a substring of the main string. This type of quote is referred to as a nested quote. But what if you want a quote within a quote within a quote? This would probably be an unlikely scenario, however you can use both single and double quotation marks when marking a multi-line string with triple quotes.

## ***Developing a Text Application***

Now that you know a bit about software design and have a basic understanding of text and strings, it's time to put your knowledge to practice. We are going to create a simple script that generates some information about a number of characters in a roleplaying game. We will have a few variables for their name, gender, race, and description and they will hold string values. Then the values will be printed to the display.

You already possess the required knowledge to do all of this yourself, so before you go through the example you should try to develop your own program. With that in mind, it would look something like this:

```
"""
```

myFile.py

Problem: Generate information on roleplaying characters

Target audience: Personal / Friends and family

System: Windows / Linux

## Interface: Command Line

Functional requirements: Print a character sheet.

User will input name, gender, race and description.

Test: Basic execute test.

~~~~~

Name = “”

Gender = “”

Race = “”

Description = “”

# Prompt user for information

Name = input ('what is your name? ')

Gender = input ('what gender are you? (male / female / not sure): ')

Race = input ('what race are you? (Human / Elf / Orc / Dwarf / Gnome): ')

# Output character sheet

```
cool_line = '<__==|#|=---++**\$/**++---=|#|==__>
```

```
print("\n", cool_line)
```

```
print("\t", Name)
```

```
print("\t", Race, Gender)
```

```
print("\t", Description)
```

```
print(cool_line, "\n")
```

This program is an enhanced version of “Hello World”. We used the same concepts and the same techniques with the addition of a few small details. Firstly, we included our design document in order to always remember what the purpose of the program is. Secondly, we used a number of escape

sequences, such as “\t” and “\n”. These sequences are used to format the text in order to organize it and make it easier to read. The “\n” represents a new line and the “\t” represents a tab. Whitespaces are great for string formatting and these two are the most commonly used.

Now that we have our character sheet, we’ll need to come up with attributes for our roleplaying characters. To do that we’ll need to start working with numbers.

## Numbers

Assigning a number to a variable is no different than assigning a string or any other type of value. Let’s take an example:

```
x = 5
```

```
y = 2
```

In this case, Python automatically detects a numeral character instead of a letter or punctuation mark and determines that it’s dealing with a number, namely an integer. Keep this in mind so that you never write the name of a variable starting with a number. Now, before we continue with numbers, you need to gain a basic understanding of how computers work.

Data is stored as a sequence of zeroes and ones, which means in binary. All of this information is processed by triggering a large number of switches that can only have two states, on or off, or in other words 0 and 1. There’s a lot more to this, however for the purpose of working with Python and using numbers in operations this is all you need to know. With that being said, here are the numeral types you will be often using:

1. Integers: In programming, as well as mathematics, whole numbers have a name and that’s integers. If it has a decimal point, then it’s no longer an integer. However, they can either be negative, positive, or zero. Integers are frequently used in many operations.
2. Floats: These numbers have decimal points. If 1 is an integer, 1.1 is a float. Just like integers, they can be either negative or positive. They are assigned to a variable the same way. Simply put, if any number has

a decimal point, Python automatically sees it as a float and you don't need to do anything else.

3. Booleans: This is the most basic type of numerical value and it's connected to the way your computer works. A Boolean can only have two values, True (1) or False (0). A programmer uses them together with logical operators such as "and", "or", and "not". We'll discuss this in more detail in another section. Variables have assigned Boolean values by using Pythons True or False keywords. Make sure the first letter is always upper case, otherwise the meaning isn't preserved.

## ***Basic Operations***

You know how to assign values to variables, so now let's perform a number of operations to manipulate them. Numeral data is manipulated by using the same operations you learned in elementary school. You can add (+), subtract (-), multiply (\*) and divide (/). Performing any of these operations creates an expression, which is code that the computer needs to process in order to get to the result (the value). Expressions can be assigned to variables by using statements. Here are several examples:

```
strength = 10 + 2
```

```
dexterity = 10 - 4
```

```
endurance = 6 * 2
```

```
awesomeness = strength + dexterity * endurance
```

```
awesomeness
```

```
84
```

You probably saw this output and blinked twice because you expected the result to be  $12 + 6 = 18$  and multiplied by 12 equal to 216? The answer is no, because the calculation goes like this:  $(10 + 2) + ((10 - 4) * (6 * 2))$ . Python knows the rules by which it should process each operation.

This example was a demonstration of how Python evaluates a certain expression and then decides which sections of it go together with other sections. So how can you know its internal decision process? Simply put,

there's an order to operations, which is called operator precedence. You can refer to this as a rule that tells Python which operator comes first. If you don't want Python to use its default rules regarding operators, you should type the code differently. Here's an example:

```
awesomeness = (strength + dexterity) * endurance
```

Now the result is 216.

So far we used only integer values in our example, however you can simply replace them with floats and everything works exactly the same.

With Python you can also convert one type of numeral value to another. Here are the most important ones:

1. Converting any number to an integer: `int (x)`
2. Converting any number to any float: `float (x)`
3. Converting any value to a string: `str (object)`

As you can see, now we are dealing with functions instead of operators and you can tell by the change in syntax. Here the value we're manipulating is in the container, determined by brackets, that is part of the function. Here are some examples with these conversions in action:

```
float (12)
```

```
12.0
```

```
int (10.4)
```

```
10
```

```
float (int (15.5))
```

```
15
```

Now that you have enough knowledge about numerical data, let's create a simple number application and apply everything you've learned so far.

## ***Developing a Number Application***

If you're looking forward to continuing your character generation program, you will need to wait a little longer because you need to learn how to compare values and store data. In other words, you need to learn how conditional statements work and manipulate data types that are slightly more complex than strings and numbers. However, before you pursue that knowledge, you first need to understand how to use simple mathematical data types.

Now let's put math to use in a simple example. Let's say you are looking to buy material to make some custom curtains for your fantasy home. Your first problem is knowing how much of it you need.

The first thing you need to do is determine the exact problem, namely figuring out how much material you need, which you can calculate based on the size of your windows. Then you need to establish the functional requirements, which are that you need to input the window dimensions and as a result you receive the amount of material you need to purchase and how much it costs. The next step is to do your research. Keep in mind that as a software developer or programmer, you will often have to research about fields you know nothing about. In this case you'd have to use your online research skills to learn how the amount of material for curtains is calculated. Additionally, you can go to a store that sells fabric and talk with an expert. Once all of your research is concluded and you have all the information you need, you should create your software design document. Always make a habit of creating this outline for your application. Your future self will be grateful. Here's how it would look in this case:

```
# Prompt user to input window dimensions measured in centimeters  
# Consider a margin of error and add to the dimensions  
# Calculate the width of the fabric  
# and calculate the length of the material for one curtain  
# You need two curtains, double the quantity of material  
# and divide by 10 to have the dimensions in meters  
# Calculate the price of everything
```

## # Print result

First you need some information about the width of the material and the price per meter. Let's take a width of 150 cm and a cost of 10 currency units per meter. Next, the program needs to be able to ask for some information regarding the height and width of the window. This data will come from the user, and to implement this feature we are going to use the input function. This function will return a string, however we will have to convert it to a float. Then we can start doing our calculations. Here's what the program should look like:

```
# One roll of material is 150 cm in width
# One meter of material costs 10 units of currency
material_width = 150
price_meter = 10
# Prompt the user to input the window measurements in cm
window_height = input(' What is the height of your window (cm): ')
window_width = input(' What is the width of your window (cm): ')
# Extra material is needed for the hems
# Convert the string to a float
# Without the conversion you will not be able to do the math
curtain_width = float(window_width) * 0.75 + 20
curtain_length = float(window_height) + 15
# Calculate how much material you need
# The calculation is done in widths
# Calculate the total length of material needed for every curtain (stick to
cm)
widths_of_cloth = curtain_width / material_width
total_length = curtain_length * widths_of_cloth
```

```
# Double the quantity of material because we have two curtains  
# Remember to divide by ten in order to calculate meters  
total_length = (total_length * 2) / 10  
# Calculate the price  
price = total_length * price_meter  
# Print the result  
print("You require", total_length, "meters of cloth for ", price)
```

Take note that the numbers aren't accurate here. If you have experience with curtains and fabric, or if you are willing to do the necessary research, you can gain accurate results. However, for the purpose of a demonstration, the numbers themselves aren't important.

## **Summary**

In this chapter you learned about variables and how to assign values to them. You explored the basic data types such as strings, integers, floats, and practiced several mathematical operations with them. Furthermore, you learned how to combine strings and convert one value to another. Now you have enough information to create a simple program that takes information from the user in order to perform a task. Such applications may be extremely basic, however, the more you progress the more you can expand. In the next chapter you will learn how to compare values and teach the program to make decisions on its own based on those comparisons.

# **Chapter 4: Decision Making in Python**

So far you learned the fundamentals of working with Python and designing applications. You have enough programming knowledge to manipulate numbers and text strings, and you can use these simple data types to create fun little programs. More importantly, you now know how to design a program and how to structure it from the very beginning in order to have reachable milestones throughout the development phase. Next, you need to learn how to improve your design and add the much needed functionality to your programs that only comes with more complex features provided by Python.

The more you learn about programming with Python, the more you will extend your ability to design more complex programs that can solve more complex problems. As you progress, you should always revisit your old applications and see what you can do to improve them. Updating and improving a program is part of any programmer's skillset, but most importantly you will notice your own progress and how your way of thinking changes as well.

With that in mind, in this chapter we are going to explore a number of methods that will improve everything you've done so far. You will learn how to compare values and then instruct your program to make decisions based on that. From every decision, a different action will be taken. This will allow you to create far more flexible programs that can have a limited ability to think on their own. They will be able to assess a situation and make one of several choices that are available to them. However, with choice comes complexity. This means that you need to adapt your way of managing your code as it continues to grow into bigger scripts. Furthermore, by the end of this chapter you will also learn how to use the power of loops.

## **Conditional Statements**

The previous application we discussed can be useful, however at the moment it is not accurate and it lacks certain functionality. In addition, there are some logical problems within the code and we need to solve them in order to improve the program. One of the best ways of obtaining results

with a higher accuracy is by using value comparisons. If we can compare values, we can instruct the program to take a different action based on the result. This is what conditional statements are for.

Conditional statements add a layer of complexity to the program. So far all we have is a list of commands which yield certain results. However, a conditional statement allows you to take your script to the next level by giving it a limited ability to think for itself. In other words, if a certain condition is met, the program will perform an action. Here's how the entire process looks in pseudo code:

If condition is true:

    Perform these actions;

If this condition is true:

    Perform another set of actions;

If this condition is false:

    Don't do anything;

Sounds quite logical doesn't it? Putting your thoughts into pseudo code can be extremely helpful and you should always practice this habit. Furthermore, the pseudo code we have in this example is very similar to how conditional statements are written in Python code, as well as a few other programming languages. As you can see, the conditional statement follows a simple logical structure: If x, then y.

However, before we dive deeper into using conditional statements, you should know that there are several ways you can enforce a condition. You can compare values by using any of the available comparison operators, namely:

1. Less than: <
2. Less than or equal: <=
3. Greater than: >
4. Greater than or equal: >=

5. Equal: =

6. Not equal: !=

All of these operators have a different effect on the data and you can use them in a variety of ways. You can even write the same conditional statement with different operators by simply changing the logic of it. Just keep in mind that conditional statements give you either a true or a false answer. Let's take a look at a few examples by using numbers:

$5 < 9$

True

$-10 \geq 10$

False

$5.5 \neq 5.5$

False

$23.56 > 78.2$

False

Now let's take a look at a conditional expression by first assigning a value to a variable:

```
myVariable = 10
```

```
myVariable == 10
```

True

Pay attention to the equal sign because if we have only one it means we assign a value to a variable. When we use two equal signs back to back, we are comparing two values. The result is either True or False, depending on whether the two values are equal or not. Many programmers confuse the purpose of these two signs when they start out learning. Luckily, if you use the wrong one, such as the assignment operator instead of the comparison operator, Python will throw you a syntax error and you will know what to

do. However, keep in mind that if you use the comparison operator when you should be using the assignment operator, your program might still work without any errors showing. The problem in this case is that you will most likely receive the wrong value and that can lead to the entire program being compromised and giving you wrong results.

## ***Control Flow***

In most cases, when developing an application you will need to be able to choose in which order a section of the code is executed. This is why we have control flow statements which evaluate individual statements and instructions to determine how they should be executed. So far you mostly compared values, but their results are needed to create conditional statements, which are in fact a type of control flow statement. They are used to allow the program to initiate certain actions based on the conditions that are met.

In Python, as well as programming in general, we create conditional statements by using the following keywords: if, elif, else. Keep in mind that if you are familiar with other languages you'll notice that in Python we don't have the "then" keyword. With that being said, let's take a look at the syntax of a conditional statement:

if condition:

#Do this

    print "This condition is true"

elif condition != True:

#Do this instead

    print "This condition isn't true"

else:

#Perform a default action if none of the conditions are met

    print "The condition isn't true or false"

As you can see, the syntax is straightforward and easy to understand because you are already thinking in these terms even if you aren't aware of it. Now let's break it all down. We start the example with an "if" conditional which is then followed by the expressions which gives us the result. The next statement is an "elif" which stands for "else if". This statement will only be evaluated by the program when the first statement is determined to be false. Finally, we have the "else" statement. If all previous statements fail to pass the test, this last one works as a default, or a catch for any data that doesn't fit the conditions we laid out. This way if something that you didn't account for occurs, your program will continue to run instead of throwing an error because it needs a specific condition to be met. Keep in mind that in Python you don't necessarily have to use the elif or else statements. There are situations when you want the application to not do anything if the condition is met. Therefore, all you need is the "if" statement. In addition, you can write as many "if" statements as you want to include more options.

Now, pay attention to the indentation in our example. In Python indentation matters and if you don't respect it, you will get an error when you run the code. Unlike other programming languages that rely on curly braces, Python doesn't use any form of punctuation to establish a block of code. If you don't respect it, you're going to get an error like this from the interpreter:

```
if x:  
    print (x)  
    x + 1  
  
indent = "terrible"  
  
File "<stdin>", line 4  
    indent = "terrible"  
  
    ^
```

IndentationError: unindent does not match any outer indentation level

The importance of conditional statements cannot be ignored. They give a program the ability to verify the data and make sure that it's the way it's supposed to be, according to the developer. This check is referred to as validation. Imagine the program having an internal checklist when looking at an item. It looks at every feature and characteristic and then it ticks it if it's in order. Use conditional statements to validate whether all of the data is provided, the numbers are within a certain range, the data types are the ones we need, and so on. Making checks is one of the most important parts of programming and you will use conditional statements for this reason no matter in which programming language you write your code.

## ***Handling Errors***

So far your Python scripts have been simple and they didn't require anything more than a basic run test. However, the more complex your programs become, you will need to improve your tests to make sure your design works. One of the most popular methods is to use a trace table in order to find problems in your project. Trace tables are used to either test your algorithms or your program for any kind of logic errors. What they do is basically simulate every step of the application's execution. Each statement is evaluated step by step. A simple trace table looks something like this:

| Step | Statement  | Notes | variable1 | variable2 | condition1 |
|------|------------|-------|-----------|-----------|------------|
| 1    | condition2 |       |           |           |            |
| 2    |            |       |           |           |            |
| 3    |            |       |           |           |            |

Now, we're going to go back to our window curtains application and see what we did wrong. We're going to trace the variables' values and statements throughout the execution of the application. Our trace table headers will be replaced with the names of the variables and conditions in the program. Ideally, you'll want to perform multiple traces with different data sets to attempt and find every alternative and possibility. Keep in mind that in most applications some of the most troublesome and time consuming errors occur due to anomalous values, also referred to as critical values.

These values are not supported by the application and they cannot be used. This is why you should test early for this frequent issue in order to expose any large design flaws.

Taking our curtain example, the most important value is the one for the material width, which is set to 150 cm. If we reserve 20 cm for the hems, then the largest individual material width we can use for a curtain is 130 cm. For anything larger than that we would need to stitch two widths, or more, together. By our estimates, the curtains need to occupy three quarters of the window's width, which means one width of material can only cover a maximum of 170 cm. This means that for a larger window we'd need to calculate several widths. Furthermore, you need to ask yourself what if the window is wider than it is deep. This would mean that you could turn the material, providing 15 cm worth of a hem. However, the window's height would have to be less than 125 cm. As you can see, we have several conditions to take into account. In order to verify them, the easiest way would be to create a number of set values for possible windows sizes, like 100 by 100, 100 by 200 and so on. Additionally, you would have to calculate the results by hand in order to make sure that the program outputs valid results. Let's work this out:

1. Starting with the first case where we take 100 x 100 as the measurement, we don't have to worry about the width or height for obvious reasons. If we take the hem into consideration, we will need 115 x 95 cm of material. Now, keep in mind that you have two additional situations here because you can measure the material along the width or the length. You will get different values in both cases. If you go with the length, you will need roughly 230 cm of material, but if you measure the width instead, you would need only 190 cm.
2. In the second example, if the window is 100 x 200 cm, the curtains need to be 150 cm in width. Since this measurement is precisely the width of our material, we need to calculate an additional amount of cloth for the hem (20cm), and for the material join as well (another 20 cm). This brings us up to 190 cm which would make the curtain 115 x 190 cm. Again, if we measure the material along the length we will need more of it than measuring along the width.

You can continue calculating other cases yourself and figure out the values. With that being said, this quick review reveals a few problems in our design. If you would continue with the calculations for other cases, you would notice that if we flip the window dimensions, we are going to have a material problem. In these cases, if we measure it along the width we aren't going to benefit at all unless the width is greater than the depth. Another condition is having the depth less than 125 cm. At this point, it's up to the programmer to make an important design decision. Should you implement the option to consider material measurements along the width, or focus only on the length? The second potential issue is that of accuracy. Ideally, you'd want the results to be rounded up to the next half meter. Keep in mind that this is about the final output and not the calculations during the execution.

Now that we have some information, let's create a trace table and compare the data. Simply use the print functions and it would look something like this:

```
# add headers to the trace table  
print()  
print('\twidth\theight\twidths\ttoal\tprice')  
  
# take the hems into account  
  
# Before anything else, you need to convert the string to a number variable  
# If you don't you will get an error due to using mathematical operators on  
text  
  
curtain_width = (float(window_width) * 0.75) + 20  
print('\t', curtain_width)  
curtain_length = float(window_height) + 15  
print("\t\t", curtain_length)  
  
# Next, you need to figure out the number of widths of material you need  
# and calculate the length of cloth needed for every curtain.  
  
# keep all measurements in centimeters
```

```

widths = curtain_width / roll_width
print("\t\t", widths)

total_length = curtain_length * widths
print("\t\t\t", total_length)

# Don't forget you have two curtains.

# Double the material and divide by ten for result in meters instead of
# centimeters

total_length = (total_length * 2) / 10
print("\t\t\t\t", total_length)

#Now we need the price in total

price = total_length * price_per_metre
print("\t\t\t\t\t", price)

```

That's it for now. Keep in mind that we used a number of tab white spaces “\t” in order to separate the variable columns and arrange our trace table. Now that we have all we need, we can put the design to the test. Here's how the results should look:

Enter the height of the window (cm): 100

Enter the width of the window (cm): 100

| width          | height | widths | total | price |
|----------------|--------|--------|-------|-------|
| 95.0           |        |        |       |       |
| 115            |        |        |       |       |
| 0.678571428571 |        |        |       |       |
| 78.0357142857  |        |        |       |       |
| 15.6071428571  |        |        |       |       |
| 78.0357142857  |        |        |       |       |

You need 15.6071428571 meters of cloth for 78.0357142857.

Now that we have our first test we can see a couple of problems. Firstly, our results are off because the conversion to meters wasn't accurate. You should be able to easily fix this. Additionally, we have far too many decimals as we really don't need to be that accurate with our numbers when it comes to curtains and fabric. In order to solve this minor issue, we should use the "round" function. It will take two parameters, the number from which we need to remove a number of decimal points, and the number of decimal places we want. In this case, we really don't need more than two decimal points. Keep in mind however, that we are rounding the displayed number and not the actual calculations that are done inside the program. We don't want to change the actual values because that may lead to other problems and more errors. Now if you perform another test with these small modifications, the result should look something like this:

You need 2.92 meters of cloth for 14.59.

We have managed to improve the results, however if we actually go to purchase these amounts of material we'll probably end up somewhat short. For some reason, the results are smaller than they should be. In conclusion, we need to know exactly what kind of values to expect in order to confirm that the program is working properly and it's offering accurate results. Without running tests multiple times, it's nearly impossible to make the calculations accurate enough for them to be truly useful to someone. However, while the results of our findings are discouraging, seeing these differences between the result and the expectation can help us figure out what happened. You should take a few minutes now, go through the program, run a few more tests, and try to see what isn't working as it should be. The application's design requires some fine tuning, so let's see what we can do.

## ***Adding the Conditionals***

The next step in improving our program is to apply conditional statements. Ideally, we need to give the application the ability to measure the cloth in more than one way. For instance, if the length of the curtain is smaller than the width of the material roll, then we can flip it and simply use one width of material. But what if the curtain needs to be longer as well as wider than

the fabric roll allows? Let's say we need an amount of material that is equivalent to less than half of the width of a roll. This means we'd have to buy another roll of material with the same length. However, if we need more than half we'd have to buy two additional rolls. The situations we just described are conditions that we need to code into the program. As usual, let's start putting all of these ideas into pseudo code, but this time mimicking actual Python syntax:

```
if curtain width < roll width:
```

```
    total_length = curtain width
```

```
else:
```

```
    total_length = curtain length
```

```
    if (curtain width > roll width) and (curtain length > roll width):
```

```
        if extra material < (roll width / 2):
```

```
            width +=1
```

```
        if extra material > (roll width / 2):
```

```
            width +=2
```

The next step is to calculate how many roll widths there are in a curtain width and how much fabric we need to determine an entire width. Here's a simple way of performing the calculation:

```
widths = int(curtain_width/roll_width)
```

```
additional_material = curtain_width%roll_width
```

All we needed was an integer function in order to cut out the decimals and then figure out how much additional material we have with the help of the modulo operator. Now you can use all of the knowledge you gained so far to stitch all these pieces together and improve your program.

Furthermore, you should consider repeating this entire process by applying different values, however you might not want to write everything manually. The purpose of programming is to make everything as smooth and efficient as possible by automating most of the tasks and by avoiding repetition. All

of this can be achieved with the help of a loop, which means that a specific task is repeated and executed multiple times. In Python, as well as programming in general, there are two ways of creating a loop. You either use the “while” statement or the “for” statement.

## “While” Loops

A “while” loop will verify if a condition is true and then execute a statement for as long as that condition remains true. Here’s a simple example:

```
x = 1  
while x < 10:  
    print(x)  
    x += 1
```

This simple loop verifies whether x is smaller than ten. As long as this condition is true, the loop will keep being executed. During each iteration, the value of x is incremented by one. When x reaches a value of ten, the condition will no longer be met, which means the loop will stop being executed and the rest of the program will continue. On a side note, iteration is simply the repetition of a number of statements, such as those in this example.

When you construct “while” loops you need to take two things into consideration. Before the loop is executed, the variable used in the conditional statement needs to be first initialized. Furthermore, you need to make sure that the variable updates with every loop execution, otherwise you will end up creating an infinite loop. Keep in mind that infinite loops can be quite troublesome, especially because Python won’t alert you when one is created. When the program is stuck in one, you need to kill it, but depending on the complexity of the program, trying to simply close the application might not work. While they won’t damage your system or cause any such drastic issues, infinite loops should simply be avoided.

When you write a conditional statement you can use any type of variable. Let’s take the example of a program that needs to calculate the mean of multiple values that are introduced by the user. On what problem should

such an application focus? We need to take the user's input into consideration because we don't know how many values he'll use or whether they will be positive. The solution is simple. We need a loop that contains a so called sentinel value. A sentinel value is used to signal the application that it should break the loop when that special value is tested. The program needs to verify whether the user's input number holds a type of value that the script should accept. For instance, while the value is positive, the loop continues iterating, however when a negative value is introduced, the program will break the loop. Here's an example of this in code:

```
offset = 0
total = 0
x = 0
while x >= 0:
    x = int (input ("Enter a positive value: "))
    total += x
    offset += 1
mean = total / offset
print (mean)
```

As mentioned earlier, one of the issues is not knowing the type of value inserted by the user. To solve this, we automatically convert the user's input to an integer. If the user attempts to input a string, the program will be terminated due to a value error.

The next question regarding loops is how to quit them. There are a few methods of breaking out of a loop without causing any problems and they are with the help of the "break" and "continue" keywords. For instance, if you want to quit a loop at a certain time without going through more conditional statements you need to use "break". However, if you want to leave just one specific iteration you can use the "continue" keyword and the loop will go over the next iteration.

In another situation you might want the program to verify the validity of a condition, however you don't want it to act in any way. This is when you should use the "pass" keyword instead. What it does is essentially form a null statement. This means that the program will know to skip this loop and move on to the next block of code. Here's an example:

```
while True:  
    my_input = int (new_input ("$? : >> "))  
    if my_input > 0:  
        pass  
    else:  
        print ("This input is negative")
```

Now let's see an example of using break and continue:

```
offset = 0  
total = 0  
  
while True:  
    my_input = float (input ("$? : >> "))  
    if my_input < 0:  
        if offset == 0:  
            print ("You need to enter some values!")  
            continue  
        break  
    total += my_input  
    offset += 1  
    print (offset, ':', total)
```

It's worth mentioning that in Python you can nest multiple loops, as well as conditional statements. In fact, you can have an infinite structure of loop levels, however it's not recommended to go to that extreme. Nesting too many loops can quickly become confusing and can lead to a number of errors due to the programmer not being able to make heads or tails of a dozen loops within loops. Knowing which condition the program tries to follow out of a large number of options is no easy tasks. Remember, always keep it simple. Furthermore, indentation can lead to a readability problem. Each loop and conditional statement needs to be indented and a large number of complicated blocks will look horrendous to any programmer no matter his skill level. The general unwritten rule here is that if your program calls for more than two looping layers you should go over the design once more. There is no need for so much nesting. There is always a simpler solution that can improve your code drastically.

## **“For” Loops**

Another popular way of creating a loop is by using the “for” statement. Its structure resembles that of the “while” loop. In fact, in most cases you can write the same loop with either statement. For many programmers it's actually a matter of style, however “for” loops are best used when you have to iterate over a sequence (lists, dictionaries, strings, etc). During the first iteration the first element from the sequence is made available, and the second element is then taken during the second iteration, etc.

To understand how the “for” loop works, you should know a few basics about sequences. While they are reserved for a later chapter, you already worked with one type of sequence, which is the string. Strings are the simplest form because they are in fact a sequence of characters. At the same time we have lists, which are sequences of data that can be edited and manipulated in multiple ways. Furthermore we also have tuples, which are very similar to lists except that their data element cannot be edited. With that being said, here's an example of a basic “for” loop.

```
buildings = ["store", "house", "hospital", "school"]
```

```
for x in buildings:
```

```
    print (x)
```

Each element in the list will be printed. At the same time you can loop through a string as well:

```
for x in "store":
```

```
    print (x)
```

Every character in the word “store” will be printed individually.

Breaking from a “for” loop is done the same way as from a “while” loop. You can use the break statement like in this example:

```
buildings = ["store", "house", "hospital", "school"]
```

```
for x in buildings:
```

```
    print (x)
```

```
    if x == "hospital":
```

```
        break
```

This allows us to break from the loop once we reach the third iteration which includes printing the word “hospital”. Keep in mind that in this case, the break happens after the element we specified is printed. Here’s a slightly different case:

```
buildings = ["store", "house", "hospital", "school"]
```

```
for x in buildings:
```

```
    if x == "hospital":
```

```
        break
```

```
    print (x)
```

In this example, we will break out of the loop before the word “hospital” is printed. Next, we can use the “continue” statement to exit the current iteration and essentially skip it. Here’s an example:

```
buildings = ["store", "house", "hospital", "school"]
```

```
for x in buildings:
```

```
if x == "hospital":  
    continue  
    print (x)
```

The loop will cycle through every possible iteration, except for the one that includes the element “hospital”.

## Summary

In this chapter you learned how to make your applications more intelligent and actually perform various tasks themselves. Many basic operations require you to use logical operators, comparison operators, and assignment operators together with all aspects of decision making. Furthermore, you learned how to take advantage of loops and conditional statements to give the program the power of choice.

Knowing how to improve the design of your program with conditionals and loops is one of the most important fundamental aspects in programming. Improve your program’s design from now on by always considering these powerful tools. In addition, you also learned the power of trace tables and how they can help you find logic errors and solve them in order to improve the accuracy of the results. The development of an application is in fact a loop in which you design, program, test, and iterate until you have an efficient program that solves a problem.

In the next chapter you will progress to working with Python data structures which include complex data types. Learning how to manipulate sequences of data and edit them is another piece of the Python puzzle.

# **Chapter 5: Python Data Structures**

In this chapter you are going to learn how to process data in sequences. Data structures are one of the foundational elements of programming and you are going to explore the basics of working with lists, tuples, dictionaries, and strings. All of these data types can be manipulated in various ways and you are going to learn what you can do by working on a project while studying the theory.

In this chapter we are going to focus on creating a more complex program, namely a roleplaying game. A game is perfect for covering this topic because we need to use data structures in order to determine a character's statistics and inventory. Furthermore, we need to calculate the results from combat and then translate them into text data. For this project you will be using the knowledge you gained from previous chapters, so hopefully you spent some time practicing loops and conditionals because we will be using them a lot.

In addition, we will continue the discussion about design and development because complex programs come with their own issues. When creating something as challenging as a game, you will often need a lot of calculations and mathematical operations, as well as textual data, and Python is great at eliminating the repetitive obstacles that come along. For now, you focused mainly on examples of data that yield basic results. Normally, when you work on a program for the real world you will not be so lucky. Data comes in complex structures, grouped together, which would require a large number of statements from the programmer. However, if you know Python's data types well, you can simplify the entire process. This is what tuples, lists, dictionaries, and strings are for. They are data structures that will help you bring order to massive compilations of data elements.

## **Sequences**

As mentioned earlier, strings, lists, and tuples are types of sequences. There are multiple ways of accessing the data inside them, however, first we should analyze them from a different point of view. Can the sequences be

modified or not? This is called mutability. Strings and tuples cannot be modified, making them immutable, however they can still be used to form new tuples and strings. On the other side we have lists which are mutable sequences due to the fact that we can edit them by adding or deleting elements inside them.

Now the question is, how do we access the elements inside a sequence? The simplest way is by using the index of each item. The index represents the position of an element inside a sequence. It is an integer written between square brackets and after a variable. Here's an example:

```
building = "hospital"
```

```
building[0]
```

```
"b"
```

Here's another example using a list:

```
buildings = ["store", "house", "hospital", "school"]
```

```
buildings [2]
```

```
"hospital"
```

In both examples you'll notice that indexing starts from zero. In programming zero represents the first element instead of one. Everything starts with zero. Therefore, our index of two will access the third element inside the list. On a side note, you can also use negative integers to access the items. Here's an example:

```
buildings[-1]
```

```
"school"
```

A negative index will start the count from the end of the list or string.

But what if you want to access several elements at the same time? You can use what's referred to as slicing. You can slice a section of a sequence by writing two indexes separated by a colon. The first index will serve as the starting point and the second as the finishing point. Therefore a slice of [0:3] means that the elements from index 0 to index 3 will be accessed. Keep in mind, however, that the position of index three is not included, so

you will only access elements 0, 1, and 2. You can also access these elements backwards if you need to. Additionally, you can ignore writing a starting index and then the slice will start by default from the first element. The same thing happens if you leave out the end index.

Keep in mind that when you slice a sequence you actually create a new sequence. However, the data inside it will not change. You will still work with the same elements that contain the same information. This means that if you edit one of them in the parent sequence or in the sliced sequence, the data will change in both of them.

## ***Sequence Check***

Sometimes you will have to create a sequence check to verify whether an item is part of a sequence. This can be done with the help of the “in” keyword which is a Boolean that returns a true or false value regarding the membership of a piece of data. Here’s a simple example using the previous code:

“pear” in buildings

False

You can also add the “not” keyword if you want to achieve the opposite:

“pear” not in buildings

True

Additionally, you can use other operators in order to make a sequence combination:

“sun” + “flower”

“sunflower”

Furthermore you can use the join() method to combine strings. As you can see, we have different ways of combining different types of data. This also applies to other operations we perform. Depending on the data type the functions, methods, and operators needed have to be adapted. However, in most cases you either use the mathematical operators such as the plus and minus signs, or the keywords “in”, “or” and so on.

While we haven't covered methods yet, you should know that the way they are accessed is by following a certain variable separated by a period. Methods also contain arguments inside the parentheses. An example of such a method in our case would be `separator.join(seq)`. Join is the method itself and as you can see it is fairly similar to a function. However, they are bound to a specific object, such as a data type. We will discuss objects and methods in more detail in a later chapter. For now you should know that objects are various instances of data types.

In the case of a string, when using the join method, the sequence will be taken as an argument. The result will be a single string that contains all the items in the sequence combined with clones of the initial string. Here's how this looks in code:

```
“, “.join (buildings)
```

```
“store, house, hospital, school”
```

If this seems a bit confusing you can change it to improve the readability by adding “`sep = “,”`” and then call the join method.

Next, we could divide the string into individual elements in order to create a list. This can be done with the “split” method and it would look something like this:

```
string.split(separator, max).
```

As you can see, when splitting we have to specify a second argument, which represents how many times we can split the string.

```
buildings = "store, house, hospital, school, church "
```

```
buildings = buildings.split(", ")
```

```
buildings [ ‘store’, ‘house’, ‘hospital’, ‘school’, ‘church’]
```

Now that we split the string into elements, we can process or manipulate them however we need. We can even convert the sequence in other types of sequences. For instance, we can turn one into a list by using the list method. Here's an example where we convert a string to a list of characters:

```
list (‘house’)
```

```
[‘h’, ‘o’, ‘u’, ‘s’, ‘e’]
```

Now let's convert a sequence into a tuple instead:

```
tuple (buildings)
```

```
( ‘store’, ‘house’, ‘hospital’, ‘school’, ‘church’)
```

You can also convert a list into a string like in the following example:

```
str (buildings)
```

```
“[ ‘store’, ‘house’, ‘hospital’, ‘school’, ‘church’]”
```

Another useful instruction is “len”, which allows you to find out how many elements a certain sequence has. This might not be important for our short sequences, however imagine a list containing hundreds or even thousands of elements. You're not going to start counting them one by one. Here's how it works:

```
len (buildings)
```

```
5
```

Two other useful commands are “max” and “min” which will fetch the highest ordered element in the sequence, and the lowest. Here's how they work:

```
max (buildings)
```

```
‘store’
```

```
min (buildings)
```

```
church
```

You probably thought the max keyword will return the highest indexed item and min keyword would do the opposite, right? That's not the case. There's an order to sequences and you may notice now that they were chosen alphabetically. Even if we didn't sort the elements particularly, they have a default order. Because of this sorting order you can also use comparison operators to compare multiple sequences. Let's say we have a second sequence called “jobs” and it also contains five elements. If you compare

them, Python will determine that one is greater than the other. This is how it would look like in code:

```
jobs > buildings
```

```
False
```

```
buildings == jobs
```

```
False
```

Keep in mind that if you try to make the comparison to an empty sequence, the result will always show false.

## ***Tuples***

You learned earlier that tuples are one of the immutable data structures that contain a number of elements in a specific order. They are in fact nearly identical to lists, except that they cannot be modified. Essentially, they are data packs that contain information which is locked inside.

Tuples are lists of values separated by commas and contained between parentheses. Take note that the parentheses aren't always necessary, however you should apply them nonetheless to avoid any confusions and possible errors. Another characteristic of the tuple is that the values it contains don't have to be of the same data type. A tuple can even contain other tuples, or no elements at all. Here is an example of a blank tuple:

```
emptyTuple = ()
```

Here's a tuple containing one element:

```
lonelyTuple = ('element')
```

Tuples come in sequences just like lists. However, you cannot change their values. The purpose of the tuple isn't to offer you editable data. They are sealed because their purpose is to be used when a certain value needs to be passed onto something else without changing its state. If you need to manipulate any of the data, you should make a list instead.

## ***Lists***

Similarly to tuples, lists contain a number of elements separated by commas, however they are placed between square brackets instead of parentheses. Lists can also contain a variety of data types, including other lists. Furthermore, we can perform a number of operations on them, such as concatenating, indexing, slicing, and so on. Essentially you can do anything you can do to other types of sequences. However, unlike some of them, lists are mutable structure, meaning that you can edit and manipulate the elements contained within. You can also slice them up and then assign values or data to these sections and thus change the list itself drastically. Here's a simple example of writing a list in Python:

```
character_inventory = ['sword', ['more', 'lists'], 'bag of gold', '2 healing potions']
```

To edit the list and add new items you can do the following operation with the power of indexing:

```
character_inventory[1] = 'leather armor'
```

```
character_inventory ['sword', 'leather armor', 'bag of gold', '2 healing potions']
```

Next up, you can even replace a section of the list with another sequence. Here's how:

```
character_inventory[::-2] = ['spell scroll', 'healing herb', 'pet rock']
```

```
character_inventory ['spell scroll', 'leather armor', 'healing herb', 'bag of gold', 'pet rock']
```

When we defined the new slice, we didn't clarify where it should start or end. The only thing that was mentioned was that it should have a step size equal to two. This means that every second item from the list should be affected by the alteration.

Next, we have a number of handy methods which allow us to introduce new elements to the list. For instance we have “append” and “extend”. The first option will simply add new elements, while the second one adds items found in different lists. Here's an example with both methods in action:

```
character_inventory.append ('raw meat')
```

```
character_inventory ['spell scroll', 'leather armor', 'healing herb',
'bag of gold', 'pet rock', 'raw meat']
```

```
character_inventory.extend(['mace', 'flask'])
```

```
character_inventory ['spell scroll', 'leather armor', 'healing herb',
'bag of gold', 'pet rock', 'raw meat', 'mace', 'flask']
```

Now let's delete some of the elements. This is what the "del" keyword is for, however you need to keep in mind that this operation doesn't remove the data itself from the list. It only moves the reference to a certain item and if it's used in another data structure, nothing will change internally. Here's an example:

```
del character_inventory [4:]
```

```
character_inventory
```

```
['spell scroll', 'leather armor', 'healing herb', 'bag of gold']
```

As you can see, an entire section of the list was removed starting from the fourth element. Now, let's add an element to the list by typing the following command:

```
character_inventory.insert(2,'staff')
```

```
character_inventory
```

```
['spell scroll', 'leather armor', 'staff', 'healing herb', 'bag of gold']
```

In this case we have added a new element after the item with an index of 2. Next, let's delete a specific element from the list with the following line:

```
character_inventory.remove ('leather armor')
```

```
character_inventory
```

```
['spell scroll', 'staff', 'healing herb', 'bag of gold']
```

But what if you want to delete an item and return its value at the same time? You can issue two separate instructions of course, however this can be done with one command. Remember, if there's a way to keep your code simpler

and easier to read, you should go for it. This operation is done with the “pop” method, which takes the position of an element as the argument. Keep in mind that if you do not specify the index, the last element will be chosen by default. Here’s how it looks like in code:

```
character_inventory.pop(-2)  
'healing herb'  
character_inventory  
['spell scroll', 'staff', 'healing herb', 'bag of gold']
```

As you can see, we have chosen a negative index position, which led to the removal of the second to last element and its value was then returned.

Finally, let’s see how we can sort lists. There are two common ways. First, you can use the “sort” method which directly applies a modification to the list. Second, you can use the “sorted” method instead, which will return a new sorted list without modifying the source of it. The two methods are very familiar so try not to confuse them. Here’s an example:

```
sorted(character_inventory)  
['bag of gold', 'healing herb', 'staff', 'spell scroll']  
character_inventory.sort()
```

Now, let’s reverse the order of the original list with the help of the “reverse” method:

```
reversed (character_inventory)  
<listreverseiterator object at 0x81abf0c>  
character_inventory.reverse()  
character_inventory  
['spell scroll', 'staff', 'healing herb', 'bag of gold']
```

Take note that when we change sequences with certain methods and functions we sometimes don’t get a new sequence in response, but instead we return an object. For instance, in our example we have an iterator object.

This is used in “for” loops as if it were a sequence. As a result, when we used the “reversed” function we obtained a reversed iterator object. Here’s how it works:

```
for element in reversed(character_inventory):
    print item
    spell scroll
    staff
    healing herb
    bag of gold
```

## ***Enter the Matrix***

No, not the movie, but the multidimensional lists! This is a fancy way of saying that we can nest lists inside other lists, thus forming a matrix or a multidimensional list. This way we can use lists to hold data in the form of a table. Here’s an example of nested lists assigned to a variable:

```
my_matrix = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

This is a three by three matrix. If you remember your elementary school math classes it may look familiar.

When you iterate through a list you should probably use the “for” loop as it is considered a standard practice. Why? Because if you aren’t sure how many elements your list contains, your system does. Computers are amazing like that. However, if you are dealing with a matrix instead, you should use a “for” loop within a “for” loop. In other words, you need nested loops, but you need them for every single list dimension. Next, you can use the “enumerate” function in order to gain the index values. Let’s take a list example, followed by one with a matrix:

```
for x, value in enumerate (character_inventory):
    print x, value
    0 spell scroll
```

1 staff

2 healing herb

3 bag of gold

Now let's see the matrix version:

for row in matrix:

for x, value in enumerate(row):

print i, value,

print

0 11 1 21 2 31

0 21 1 22 2 32

0 31 1 32 2 33

## **Dictionaries**

This type of data sequence is similar to the yellow pages. As long as you have a name to search for, you are going to find his phone number and other information. In this example, the name of the person is called a key and the data comes in the form of values. The key itself isn't mutable, which means it can only be a string, a tuple, or a number, while the values can be any kind of data. Dictionaries, however, are mutable structures. This means that you can edit a number of values. This includes adding and removing them, as well as changing keys. On a side note, dictionaries are also known as mappings because their keys are mapped to certain objects. This makes them significantly different from the other sequences, and that is why they also carry an alternative name.

If you are familiar with other programming languages, you might notice that data types that have a similar structure to a dictionary are known as hashes or associative arrays. These are important differences because values that determine the keys are in fact hashed values. What does that mean? Hashed keys can be used to store larger keys because they are in fact small values that are defined from the key. All of this might sound extremely

confusing for now, however Python arranges the data inside a dictionary based on the hashed keys instead of following an alphabetical order. Therefore if you modify the value, the hash that results from it will also be modified and therefore the program will not be capable of finding the value stored within it. This is why keys are not mutable.

Now let's put the theory aside and see how dictionaries look in code. Keep in mind that they are defined by pairs of keys and values. They are separated by colons and contained within curly braces. Make sure not to confuse your braces with brackets and parentheses because if you mix them up, the application will not respond correctly. With that being said, here's an example:

```
character_profile = {'Name':'','Description':'', 'Race':'',
'Sex':'','Strength':0,'Intellect':0,'Dexterity':0,'Charisma':0}
character_profile['Name'] = "Player"
print character_profile['Name'], "has", character_profile['Intellect'],
"Intellect"
```

Player has 0 Intellect

You need to use square brackets when you pass a reference to one of the elements inside the dictionary. However, keep in mind that it doesn't work the same as with lists where we can use an index value that represents the element's position. Dictionary items don't have positions. Instead, we have the keys and the values that match them. This means that we need to call the key in order to refer to its value. The syntax looks something like this “[dictionary [key] = value]”. Keep in mind that the name of the key has to be between quotations marks, otherwise it will be seen as a variable. Another method of navigating through the elements of a dictionary is using the ‘for’ loop. As you can see, loops are extremely handy in a multitude of situations. Here's an example:

```
for key in character_profile:
```

```
    print key,
```

Intellect Name Sex Race Charisma Strength Dexterity Description

If you need to access the values, you need to type “profile [key]”, but if you need to check whether a certain key even exists, you need to look for the key in the profile. The result of the check will either be true or false. The only other possibility is an error. However, let’s say you aren’t sure whether a certain key already exists or not. In this case you can use the following method:

```
get (key, default)
```

Here we need to use arguments. The first represents the key you are interested in. The second argument represents the default value in the event that the key doesn’t exist. Now, let’s take a look at some code:

```
character_profile.get('Name')
```

```
'Player'
```

```
character_profile.get('Sex', 'unknown')
```

```
"
```

```
character_profile.get('Health', 'not found')
```

```
'not found'
```

You’ll notice that we have the key, however there’s only an empty value instead of a message. Now, let’s see how we can delete an element from a dictionary:

```
character_profile {'Name': 'Player', 'Sex': "", 'Race': "",
```

```
'Charisma': 0, 'Strength': 0, 'Dexterity': 0, 'Description': ""}
```

```
del character_profile['Sex']
```

```
character_profile
```

```
{'Name': 'Player', 'Race': "", 'Charisma': 0, 'Strength': 0, 'Dexterity': 0, 'Description': ""}
```

As you can see, like with other sequences, in order to delete an element we have to use the “del” method. Furthermore, if you need to delete a value

from the dictionary you can do so with the “pop” method as demonstrated earlier. Sorting works similarly as for lists by using the “sorted” method.

Essentially, dictionaries are an important data structure because they are needed in any program where we have to store a number of attributes and values. For instance, you can even define one to simply count the different states of a certain object. This works because every single key is in fact unique and therefore you won’t end up with identical values for one key. This means that the key can be used to store input data and then the value will store the results of any operations we perform. For example, if you want to learn how often a letter appears inside a certain sentence, you can iterate through all of the individual characters and then have every one of them assigned to a key.

## **Creating a Basic Game**

As mentioned in the beginning of the chapter, we are going to create a very basic game in which you will use everything you learned about sequences. This will essentially be a continuation to the simple character generator we worked on earlier. Now, before we get to coding, let’s start from the very beginning with the design.

The code will be split into three major blocks. You will need to be able to generate your character’s stats, purchase some gear, and go to battle. Let’s further develop these ideas:

1. First we are going to need to store character stats. Since we need a container for them, a dictionary is probably the ideal choice. The player will have the ability to determine his or her own character name, description, sex, and race. The character stats however, namely strength, dexterity, charisma, as well as gold, will be random. We will also introduce a number of secondary stats such as health, mana, and armor. In order to generate the numbers we are going to implement a random module. We didn’t discuss this topic yet, however it’s enough for you to know that a module is a compilation of functions that you call in order to perform certain tasks. Python modules need to be imported into the program before they can be used. However, once you import them they are used in the exact same way as normal methods

and functions. You can think of them as an extension to Python's abilities.

2. In the next section of the code we are going to define the “trader” where the player is going to spend his gold on a variety of items. Again, the ideal container would be a dictionary. The items offered by the trader will have their own keys to which we will connect the prices (values).
3. Last but not least, we will implement the combat system. Characters will be able to select a weapon and attack someone when it's their turn. Furthermore, we don't want to see only some numbers, so we are going to add text to describe the amount of damage that is dealt. To achieve this we need to use a number of tuples. The next problem is how we are going to deal with the weapon damage calculations. We are going to cheat a little here and introduce two more elements to our “trader”. One of them will be “attack damage” and the other one “attack speed”. In addition, we will have a tuple containing the price, attack damage, and attack speed. Keep in mind that the numbers for these elements will not change, therefore we are going to add them right at the start of the program so we get them out of our way as soon as possible. Lastly, we need the battle phase to go into a loop until one of the characters dies.

Now that we have the general design for the program, let's begin phase 2 of the development: pseudo code!

```
# declare the constant variables that don't change their values  
# define the trader = {}  
#1 while characters < 2  
#1.1 declare character profile dictionary  
#1.2 Ask the user to insert the following information about their character  
# (Name', 'Description', 'Sex', 'Race')
```

```
#1.3 Confirm data

#1.4 Implement random stats generation
#('Strength', 'Intellect', 'Dexterity', 'Charisma')

#1.5 Implement secondary character stats
#('health', 'mana', 'armor', 'gold')

## Allow the user to modify certain stats

#1.6 Confirm the character stats

#1.7 Display character stats document

#1.8 Ask the player to buy some gear from the trader

#1.9 loop while purchase not equal to 'no'.
#1.9.1 Display trader's item list with prices.

#1.9.2 Ask the user to buy items.

#1.9.3 If the player has enough gold and
#the trader has the item available, buy the item.

#1.10 Introduce another player

## Combat section

# Prepare the lists with descriptive text.

## hits = (), misses = (), damage_taken = (), health_update = ()

#2 Player enters battle.

## Player decides which weapon to use.

## As long as the weapon is in his inventory.

## If selected weapon doesn't exist, player will use his fists.

#3 Loop while opponent [health] > 0 and target[health] > 0:
#3.1 for player in battle:
```

#3.1.1 Calculate the attack speed

#3.1.2 Calculate the attack damage

#3.1.3 Print damage taken

#3.1.4 if opponent [health] > 0 and target [health] > 0: break

#3.2 Print player progress

#4 Print winner

As you can see, this program is somewhat more complex than what we designed so far. Before you continue, make sure to pay attention to the structure of the pseudo code and the indentation. The more complicated a design is, the more time you should invest in preparation. Don't look at an application as if it's a monstrous task that will require who knows how many man hours to develop. Instead, split it into smaller tasks, organize the blocks of code, and the problems you need to solve. Then you can start tackling them, one at a time.

With that being said, the pseudo code we have so far can be easily translated into Python code. First we have the constant variables and then the import random statement. We are going to use random methods. Next, we have the first section where we determine the items inside a dictionary named "trader". Each element contains three tuples that represent the price, attack damage, and attack speed. Then we have a set of elements which will be the items our character can purchase, such as armor and weapons, followed by an additional four tuples. They will hold the text which describes weapon hits, misses, the damage report, and the health updates. Because this is a game after all, you should dedicate a bit of time to how you design your descriptive text. It needs to fit the atmosphere of your game and whatever goals you set out for your characters.

Next up, we have a bare list that holds the player profiles and it's intuitively called "players". Each one of these profiles is then hosted temporarily inside a character\_profile dictionary. There we will have four separate fields in which the user will introduce some information about his character. Then we start using the random module "random.randint (start, end)" in order to automatically generate the player's stats. This is essentially a random

number generator that is the equivalent of rolling dice. The secondary stats are then generated from these numbers and verified to make sure that they come somewhere between a value of 1 and 100 to keep things simple and readable. If the numbers don't fit within this range, we throw the dice once again until we get a valid result. At the end of this block we print the player's character sheet.

Next, we have the item purchasing section which we can actually attach to the character generator. This way we simplify the coding to some degree and tighten the design a bit more. Additionally, we can use a "for" loop in order to loop through the list of items the player can buy. Then we access the price from the tuple with the help of indexing. At this point, the player will be able to make a decision, which is then added to his or her inventory. An amount of gold is removed in exchange for the item. Next, the player is asked to select a weapon, and if there isn't one in the character's inventory, fists are used as the default. What really happens behind the scenes is that the player accesses the tuple from the trader dictionary and translates it to a weapon value. The same thing happens with the character's armor values. Out of all of this data, a profile is built. Finally, the combat occurs inside a while loop that keeps being executed until one of the opponents loses all of his health. Inside the loop, we have another loop that swaps each player's turn. Keep in mind that we have a turn based combat mechanic in this example. The concept is simple. The nested loop accesses the player's index with the help of the "enumerate" function and then converts it into a Boolean. After one player's turn, the "not" operator reverses the value. This means that we have an attacker and a target index which is swapped between two players.

Next, we have the attack speed calculation. As long as the player rolls a positive number he will hit the target. Negative numbers are translated into misses. We set the value somewhere between zero and seven, however we do have to limit it otherwise we will experience some unintended behavior. Then this value is verified within the misses and hits tuple and depending on the result, a descriptive text is printed. The message depends on whether an attack has landed successfully against the opponent or it missed. The value is also used in the damage calculation. Keep in mind that these values are set within a somewhat random range, within limits, in order to provide a fun factor to the game instead of rigid results.

In the last section we define how much health is taken from the target's health pool based on the damage he takes. The target's health is then updated. As long as it's not equal to zero or a negative value, the game continues. When it reaches zero, we break out of the loop and the winner is decided. The program will finish executing once the end result is reached.

## **Coding Exercise**

At this point you've learned a great deal about Python and programming in general. We have laid out the plans for the game, we wrote the pseudo code, and then explained it in great detail. Now, the rest is up to you! You have everything you need to write the code yourself. You don't need to apply anything outside of what you have learned so far. Particularly, for this exercise you should focus on what you learned in this chapter.

You might feel intimidated by this exercise, however without independent practice, beginner programmers are tempted to just copy code and see if it works. That is not how you learn programming. You need to enter the trenches and push through on your own, or with some help. Even if you don't manage to code this game, you get errors or you're stuck somewhere, research is part of a programmer's toolkit. If you're stuck, take advantage of the many online Python programming communities and ask questions. Through every attempt you will learn something new.

## **Summary**

In this chapter you have hopefully accumulated a great deal of information, perhaps even overwhelming at times. You have continued building upon the foundation you built in the earlier chapters and now you are able to start taking the first steps on your own. You have learned to work with the most important Python data structures, such as dictionaries, tuples, and lists, and now you can put them to good use by creating more complex applications.

Additionally, you explored the differences between mutable and immutable sequences and learned how to access a number of elements by using indexes or through slicing. With the help of various methods and functions you can now edit lists, or iterate through various sequences by using for and while loops. In programming, avoiding the use of loops is nearly impossible. Take advantage of their power by making your programs more

intelligent and by building a more simplistic and logical design to process your information.

Finally, you have gone through the development of a somewhat sophisticated program using all the knowledge you accumulated so far. You took the first steps on your own by writing your own code and researching any problems you encountered along the way. This challenge is an important step on your journey to learning Python, as it's your first taste of real world programming that comes with its ups and downs and all those wonderful “aha!” moments.

# **Chapter 6: Functions**

So far you have focused on working with variables, performing basic arithmetic operations, comparisons, and giving your program a limited ability to make choices. You have enough programming knowledge in Python to take your skills to the next level.

At the moment, you can use what you learned to write basic scripts that are executed in a precise order from top to bottom, however there is far more you can do. The biggest problem you are facing right now is that if the program needs to repeat a certain section, it has to go through all of it. In turn, this leads to yet another even more severe problem. If you need to modify your code you will probably damage everything unless you review the entire script. This makes maintenance and updates a time consuming chore and it doesn't have to be this way. Ideally, you want to divide your code into different sections where each one of them handles only one task. This is what functions are for and in this chapter we are going to focus solely on them in order to write optimal code.

## **Creating Functions**

So far your programs were nothing more than a list of instructions that were executed one by one. They work just fine, however, the more they grow the less you will be able to keep up with the complexity of it all. That is why you need to learn to take advantage of functions and simplify your work.

You have already used a number of functions that are built into Python. However, the default functions are not enough when developing anything with some degree of complexity to it. Fortunately Python, like other programming languages, allows you to build your own functions. Once you create one, it is used in the exact same way as the ones you already have access to.

Defining new functions comes with a series of advantages:

1. By using them you can divide your code into any number of small blocks. This way you will be able to later modify your code without breaking the entire program. Updating and maintaining small blocks of

code is much easier to achieve than having to go through an entire application.

2. Custom functions will eliminate any repetition. While there's no written rule about repeating code, the general opinion is that if you have to write something more than twice, you should convert it into a function. You want your code to be as simple, efficient, and readable as possible. Therefore, you should write functions that include the code that you need to use multiple times throughout your program. When you need to execute it, instead of writing the section again, you simply call the function.
3. Another advantage is when working in a team of programmers. By splitting the application into manageable sections, each programmer can work on one independently. This way you can finish the project at a much faster rate because at the end you easily connect all the pieces.

Now, let's take a look at a new project that's been written with the help of functions.

As you already know, the best way to learn is by having a project in front of you. In this section we're going to work on a simple tic tac toe game where the user will play against the computer. We're going to start with the game instructions which are written as a separate program. We're going to analyze the code in order to gain a better understanding of user defined functions. Let's get to it! Here's the code:

```
# game guide

# illustrating user defined functions

def gameGuide():

    ““Display instructions. ””

    print (
        ““ Welcome to most epic game you will ever play!

        This will be a battle of wits between you, the fleshy creature

        Against I, the all-knowing computer!
```

You will start by entering a number from 0 to 8,  
which will correspond to the following positions on the game board:

0 | 1 | 2

3 | 4 | 5

6 | 7 | 8

Prepare for battle! \n" )

# main

print("This is the game guide for Tic Tac Toe: ")

gameGuide()

print("Review them once more:")

gameGuide () print("Surely you understand how it works.")

input("\n\n Press the enter key to quit.")

Now let's analyze the code in the following sections and see how functions work.

## ***Function Definition***

In order to create a function of your own you need to define it by using the "def" keyword. In the above example it was the first thing we did:

def gameGuide ():

So what do we mean by definition? The code we typed tells the computer that all the statements that follow will belong to this function. Essentially, we're declaring a block of code and giving it a name. This way we can always have access to the code within the function by simply calling the function. The information inside the function defines its purpose and what actions it takes. When Python sees the function definition, it knows that it's inside the program and that you can always call it wherever you need it. Keep in mind that this means the function will not perform any task by itself as it's defined. You need to call it at some point inside your application.

To call it you simply have to type the function's name together with its parentheses. In our example we did this a couple of times with the following line:

```
gameGuide()
```

Now the system knows that it needs to initiate and execute the function. In our example, whenever we call the function, the program will print the game guide.

As for the syntax, you can see it's fairly simple. All you need to do is start with the "def" keyword, type an intuitive / suggestive name that describes the function, followed by parentheses, which can hold parameters, and finally the colon. Then you can write any kind of statements you want depending on what the purpose of the function is.

## ***Function Documentation***

User defined functions allow you to document them with the help of a documentation string, also known as a docstring. Here's how it looks in our little game guide:

```
'''Display instructions.'''

```

As you can see, we need to use triple quotes just like we can do when we comment out a block of text. However, when you write a docstring, it has to be the very first line after defining the function. In our example we only wrote a simple sentence that describes the purpose of the function. Keep in mind that documentation isn't necessary, but if you create a large number of functions you might want to make sure you later know what they're for, especially if you work with other programmers. Docstrings can be later accessed inside IDLE and you can review the function you created the same way you can do with the default documented functions.

## ***Using Parameters***

As you may have noticed when working with default Python functions, you can set their values and have them return values to you. For instance, you can use the "len" function to set a sequence which will return its length value. You can do the same thing with the user defined functions.

Next we're going to take a look at another small program where we define three new functions. You will learn how you can form different combinations of obtaining and returning values. In other words, we will set one function to receive a value and another function to return one instead. Lastly, the third function will be able to do both. Here's the code:

```
# Parameters and Values

# Exploring the use of parameters

def show (message):
    print (message)

def return_me_ten():
    ten = 10

    return ten

def ask_yes_or_no (question):
    """Ask a yes or no question."""

    response = None

    while response not in ("y", "n"):
        response = input(question).lower()

    return response

# main

show("You have a message.\n")
number = return_me_ten()

print("This is what I received from return_me_ten():", number)
answer = ask_yes_or_no ("\nPlease enter 'y' or 'n': ")

print("Thank you:", answer)
input("\n\nPress the enter key to quit.")
```

In this example we define the “show” function which simply prints the value it receives through its parameter. If you’re still having some trouble grasping parameters, you should just look at them as variables inside the function. What a parameter does is simply obtain the values that are sent through the call of a function. This means that in our first function, we have the “message” parameter to which a value is assigned, namely the string “You have a message.\n”. This string is then received by the message parameter and then the function will run it as a variable inside itself. If the parameter wouldn’t have any values assigned to it we’d receive an error. Keep in mind that in this case, our “show” function has only one parameter, however that doesn’t mean that all functions need to have one. In fact, they can have quite a lot. All you need to do is list all the parameters by separating them with commas.

Our second function in this script is “return\_me\_ten” and it returns a value through the aptly named “return” statement. When the statement “return ten” is executed, the function will send the value to the section of the application where it was called from. Take note that functions are terminated when they reach the “return” statement. Keep in mind that a function can hold multiple values and they can all be returned by separating them with commas.

The third function aptly named “ask\_yes\_or\_no” will receive a question and return the answer given by the user as a simple yes (y) or no (n). The question is obtained through the parameter that is found within the function’s definition. The question then receives a value of an argument that is sent to the function. In this script we have the "\nPlease enter 'y' or 'n': " string. In the next section of the function we use this argument in order to ask the user to give us an answer:

```
response = None  
while response not in ("y", "n"):  
    response = input(question).lower()
```

Keep in mind that we need to have a loop repeatedly executing this section until the user provides the program with an answer. Once the response is inserted, an argument is sent back to the section of the application that

called it through the function. The answer is returned, the function terminated, and the program prints it.

## ***Reusing Functions***

Let's take a side step from all the theory and coding and take a note about the reusability of user-defined functions.

One of the biggest benefits of creating your own functions is the fact that you can save them and use them in other scripts and not just your current project. For instance, in our example we are prompting the user to give a positive or a negative answer to a question. This is a common task which is performed in many other programs even if it's not to achieve the exact same purpose. You can reuse this function in many other scenarios. This will save you time and make your coding a lot less demanding.

One of the simplest methods of reusing functions is to just copy them from an old script into a new one. However, this isn't the most efficient way of handling this process. Remember how we discussed earlier about Python modules which you can import in order to extend your possibilities? Well, you can develop your very own modules and use them to import your custom functions to other programs. It works exactly the same as using standard modules. We are going to discuss this in a later section in more detail. For now you can stick to the old fashioned copy paste method.

Now during your learning process you can do pretty much anything because there are no real consequences and optimization doesn't truly matter. However, when you are going to program in the real world you want to be as efficient as possible. Coding takes time and you don't want to waste it. This is why you shouldn't try to reinvent the wheel. Most software developing companies tend to focus a lot on reusable code and developing new techniques of implementing old things into new things. Here's why code reusability is so important if you choose to follow programming as a career:

1. Boosts your productivity, as well as your team's. Finish your projects at a much faster pace and create new building blocks than can further speed up any future projects as well.

- It allows you to be consistent and become comfortable with your development environment. Humans may be adaptable, but you lose precious time by switching the way you build a project. Even something as small as working with a different user interface is enough to slow you down significantly and even confuse you.

## Global Variables

The functions we have created and used so far are closed off from each other and completely independent. This is why they are so useful. You can always move them around and connect them to whatever components you want. However, in order to access the data inside them you need to go through their assigned parameters. To do that they need to return a value. While this is regularly the case, there's another way.

Sharing data between sections of your application can also be done with the help of global variables. However, in order to understand global variables you first need to gain an understanding of scopes.

### Scopes

As you already know from more complex examples, it is normal to divide a program into individual sections that can be accessed separately from each other. This is what we refer to when we say scopes. Every single function we create has a scope. This is the reason why one function cannot have direct access to another function's variables. A simple representation would look something like this:

```
def function_1 ():  
    variable_1  
  
    #End of the first function's scope  
  
def function_2 ():  
    variable2  
  
    #End of second function's scope  
  
#This represents the global scope for the entire application
```

## `variable()`

As you can see, in this case we have three scopes within one program. By default, all programs have a global scope and in our example we have two other scopes defined by the two functions. What does all of this even mean?

When you aren't operating inside any of the two functions, you are inside the global scope. The variables that we define here are known as global variables. Any other variables that we define within one of the function blocks will belong to them and be referred to as local variables. For instance, in our example we have "variable1" inside the first function. This means that it is a local variable and therefore we cannot gain access to it from another scope such as the second function. Furthermore, we can't change the variable from the global section either.

Imagine the scope as a car with tinted glass. If you're inside the car, you can see everything that's inside it. However, if you're on the outside, you can't see inside it. This is essentially how functions work as well. You can access a function's variables only while inside it, and if you're in the global space those variables will be invisible to you. In addition, if we have multiple variables with identical names but belonging to different functions they will be their own entities with no connections to each other. For instance, if we add "variable2" to the first function in our example, it will contain completely different data than "variable2" inside the second function.

## **Handling Global Variables**

We are going to write a short program that will demonstrate how you can use a function to read and edit a global variable. Let's create the application and then analyze each feature:

```
#Playing with global variables

def read_global():
    print ("From within read_global(), value is:", value)

def shadow_global():
    value = -10
    print ("From within shadow_global(), value is:", value)
```

```
def change_global ():  
    global value  
    value = -10  
    print ("From within change_global(), value is:", value)  
  
#main  
  
#From here on we are inside the global scope  
  
#Value becomes a global variable  
value = 10  
  
print ("Within the global scope, value is: ", value, "\n")  
read_global ()  
print ("Within the global scope, value is set to: ", value, "\n")  
shadow_global ()  
print ("Within the global scope, value is set to: ", value, "\n")  
change_global ()  
print ("Within the global scope, value is now: ", value, "\n")  
input("\n\nPress the enter key to quit.")
```

## ***Reading Global Variables***

We discussed local and global variables and how they can be accessed depending on their scope. Now is the perfect time to slightly confuse you. Reading global variables can be done no matter the scope. You can read a functions variable while you're in the global scope and vice versa. Let's go back to our car with tinted windows analogy. When you're inside the car, you can see everything that's in it, however you can also see outside through the windows. This means that when you're in a function you can read a global variable's value.

The short program we created earlier demonstrates the readability. We have defined a new function called “read\_global”, which prints the value of the global variable. The instructions works perfectly fine without any errors. However, keep in mind that while you can read outside of a function, you cannot make any kind of changes, at least not directly. Therefore if we would write the following line inside the read\_global function, we would see an error from Python:

```
value += 1
```

## ***Shadowing Global Variables***

You must’ve read the name of one of our functions in the above example and asked yourself “what shadow?” Shadowing refers specifically to naming a local variable the same way as the global variable. They can both have the same name, however don’t fall for the trap of thinking that you can edit the value of the global variable. The only change occurs to the local variable. The reason this is called “shadowing” is because we are essentially hiding the global variable with the new one. Here’s what we did in our example when assigning a value to the shadow\_global function:

```
value = -10
```

The global value didn’t change. However, a local version of the same value was defined inside the function and that version has the value of -10. You can observe a demonstration of how this works because we have instructed the program to print the global version of the value.

Keep in mind that shadowing a global variable within a function is rarely a good idea. Why? Well, you probably already know. Were you confused by our example and the explanations you just read? If you were and you needed some extra time to figure out what’s what, then you should probably avoid using them. You might implement shadowing and then when you want to use the value of the global variable you use the other one instead.

## ***Changing Global Variables***

You can have access to global variables by using the “global” keyword. You can see in our example how we used it to change the value to -10. When we execute the program, the new value will be printed.

Now the question is when to use global variables. You obviously can use them whenever you want, but would it be wise to do so? In programming, just because you can do something in a certain way, it doesn't mean you actually should. You need to weigh your options because not every possibility is appropriate for your project. Global variables are an example of a problematic decision. As mentioned earlier, they cause confusion because values change and then you no longer know what exactly you are dealing with. This is the main reason why you should take advantage of them only scarcely.

On the other hand, you have the option of turning global variables into constants, which aren't quite as hard to keep track of. For instance, if you develop a tax calculator, you'll naturally have a number of functions but all of them use the .45 as a value for whatever reason. This means that we could place this value inside a global constant instead. Then we no longer have to call the value itself in every function. It might not seem like a big change but when you work on a complex project, having clear variables instead of numbers that mean nothing to you make your programming life much easier.

## **Writing Tic Tac Toe**

Early in this chapter we discussed working on a new game, namely Tic Tac Toe. Now that you have learned a great deal about functions, you are ready for a more complex program. However, let's not jump straight to writing code just yet. Complex programs require more thorough planning than anything you worked on until now.

Always start with planning! Yes, you have already read this tip, advice, or even command, multiple times in this book. That's because the planning phase is half the work when it comes to any project. It may be boring, but you need a map with a pointer on it to show you the way. Without it, you will stumble around, spend a great deal of time and effort, and then give up when you get lost in the cyber woods.

With that being said, let's begin with the pseudo code. Keep in mind that the purpose of this program is to gain practice using the functions we already wrote, and a few new ones that you will write yourself. This means that the pseudo code doesn't have to be as detailed. See? This is another

benefit of working with functions. Even your pseudo code can be simplified and thus save more time. Now let's take a look at our outline:

```
# Print the game guide  
# Determine who starts  
# Build the game board and display it  
# While there's no winner or tie  
# If it's the player's turn,  
# get his move  
# Update the board and display it  
# Otherwise  
# Get the AI's move  
# Update the board and display it  
# Swap turns  
# Declare the winner
```

Now that we have an idea of what the game is about, we need to think of more concrete definitions. How are we going to present the game board? How will a piece move? Questions like these need answering, so let's think about the entire concept. First, we need to display the board, and we can represent a piece with a character. Then the board can be a list because we can edit it whenever the two players move. Since we know that a classic tic tac toe game needs 9 positions on the board we are going to introduce 9 elements into the list. Every element will represent a section of the board. Here's how we can design it:

```
0 | 1 | 2  
3 | 4 | 5  
6 | 7 | 8
```

Don't forget to start counting elements starting from position zero, not one. Each section of the board will have a designated number from zero to eight. The moves the players make will also be represented by a number because they can move only one square per turn. As for representing the two players, the human can be "X" and the computer can be "O". Then all we need is a variable that represents whose turn it is.

## ***Adding Functions***

So far we only have part of the design. This game is complex and the pseudo code we have only represents the general outline. What we need is to create a list of functions before we actually start writing them. This way we can brainstorm what we actually need and what parameters they should take. Here's what we need:

1. `game_guide ()`: Display the rules of the game and the instructions.
2. `ask_yes_or_no (question)`: Ask the player a question which can be answered with a yes or a no.
3. `ask_number (question, low, high)`: Ask the player for a number within a range. As you can see we have a question parameter which the user receives, and a number will be returned. This number will be within a range from low to high.
4. `game_piece ()`: This function will represent the two pieces on the board. One belongs to the human player and the other to the AI.
5. `empty_board()`: Create a new board.
6. `display_board (board)`: Display the board on the screen.
7. `allowed_moves (board)`: This will return a list of moves that are allowed once the board is returned.
8. `winner (board)`: Declaring the winner.
9. `player_move (board, player)`: Retrieves the move from the human player. The board and the player's piece are received and the move is returned.

10. computer\_move (board, computer, player): Calculates the AI's move, the board and the two pieces. Returns the AI's move.
11. next\_turn (turn): Swap turns.
12. announce\_winner (winner, computer, player): Declares the winner, as well as a tie.

Now that we have an idea about the structure of our game, let's start with the preparations.

## ***Setup and Exercise***

The first thing you need to do is define the global constant. They will be used by most functions and since their values won't change you can already declare them right in the beginning. This way your functions will be easier to write and manage because you have everything prepared. Here's the code to get you started:

```
# global constants
X = "X"
O = "O"
EMPTY = " "
TIE = "TIE"
NUM_POSITIONS = 9
```

X and O represent the two pieces that belong to the player and the computer. The “EMPTY” constant refers to an empty position on the board. Its value is an empty space because when we print it, it needs to look like an empty spot. The “TIE” constant represents a possible tie between the computer and the player and no winner is declared. Finally, we have “NUM\_POSITIONS” that represents all the squares on the board.

Now you have everything you need to build your tic tac toe game. Everything you learned in this chapter has prepared you for this project. You have a detailed plan, you know which functions you need to create and what their purposes should be, and you already have the game setup with all

the global constants it requires. Some of the functions you already have because we've used them as examples earlier in this section of the book. Use the skills you have to assemble all the components that are needed to create the program. This is the true purpose of functions. They make it easy for the programmer to write and work out individually and then connect them later to build the application.

## Summary

In this chapter you learned the basics of user-defined functions and how to write them. You learned what scopes are and the difference between global variables and local variables. Furthermore, you studied the importance of documenting your functions and limiting how often you use global constants. Finally, you learned how you can assemble all of your functions together to build an efficient program that is easy to read and maintain. Now you can create new projects using a component developing approach instead of writing long messy scripts.

# **Chapter 7: Introduction to Object Oriented Programming**

Python can be considered an object oriented programming language, but what does that mean?

OOP for short is a relatively new approach to programming. In recent years, it became the standard in the software and game development industries and most programs today are created using this methodology. What defines object oriented programming is the object itself, and in this chapter you will focus on learning the basic concepts behind it. Here are some of the things you will explore:

1. Defining objects and creating classes.
2. Writing methods and declaring an object's attributes.
3. Polymorphism.
4. Creating modules.

## **Classes**

So far you have gained some limited knowledge about working with basic methods. However, one of the most important concepts in OOP is that of the class. A class is essentially a template which allows you to build your own data type. In this case don't think about data types as numbers or text. A new data type could be for instance a car, and it would have a number of unique attributes such as brand, model, color, and so on.

Classes are the foundation of object oriented programming because they are templates with the purpose of shaping objects that you use in your projects. They are the next step in adapting your programming style. You started with basic top to bottom scripting, advanced to the use of functions, and now you have reached the pinnacle by focusing on classes. What classes do is allow you to place everything in one place, which is the object you create yourself. This means that it will contain data in the form of attributes, and commands in the shape of methods.

While all of these changes in your programming journey may seem overwhelming, you should eventually strive for object oriented programming. When you have to work on long, complex programs, object orientation will vastly decrease the amount of work you will need to do. OOP allows you to reuse a great deal of your code due to the fact that objects can inherit a number of attributes and methods from different classes. This means you can easily modify an object, or create a new one containing some data from other objects. This is possible thanks to the inheritance concept which is specific to OOP. It means that we can build an object from a generic one and then create an entire family of objects that stem from it. It would look something like a family tree.

For instance, let's say we have Flora, Fauna, and Minerals as the foundational classes. They will have a number of attributes that can be passed down to any object that belongs to either class. This means that you can create subclasses. For example, Fish and Bird are child classes of the Fauna parent class and they will inherit a number of attributes from it. In addition, they will have their own distinct attributes that aren't part of the parent class. Then you can create another round of subclasses such as Chicken and Raven which will inherit attributes from the Bird class. Then we can dig a little deeper. Let's say we have a program in which we want to use the Fauna class from which we create an instance of Raven that we will call Charlie. This particular raven will inherit the "caw()" method that all ravens would have. Therefore when we write the code we can make a call to the method by typing "charlie.caw()" and the result would be a print of a string "Caw! Caw!".

As you can see, when working with objects, classes and methods we use the exact same way of thinking as in the real world. This is one of the main reasons why object oriented programming is so popular in software development. It allows programmers to think and write code naturally without writing a large number of functions.

## **Python Namespaces**

Before we start exploring classes and how we define them, you need to learn about namespaces in order to understand the entire concept of object oriented programming.

Namespaces allow the mapping from a name to an object. By default they come as Python dictionaries and you may not even notice them. An example of a namespace is the collection of default names that contain various functions such as “abs”. It also includes module names or local function calls. Even a class’ attributes will form a namespace. What you should soak in from all of this is that identical names that belong to several namespaces don’t have anything to do with each other. Let’s take a module as an example. We can have two different modules that define a function with the same name, however they will not be confused with each other. They are unique and independent.

Python creates namespaces as soon as you run the interpreter, however some of them are enabled at different times and last for certain periods of time. For instance the namespaces that are built into Python will never be deleted. Furthermore, when a module is read in, a global namespace is defined for it, however when we close the interpreter it will be gone.

Keep in mind that while certain namespaces are pre-defined, we don’t necessarily have the ability to access them from anywhere. This is when scope comes into play once again. In this case, scope also represents the section of the application from which we have access to a namespace directly. While in the previous chapter we discussed global and local scopes, here we have three of them generally speaking. The first scope is that of a function and it has a local namespace. The second is the module which creates a global namespace. Finally, we have the third one, which is the general scope that contains all the pre-built, default namespaces.

## ***Defining a Class***

Let’s start with the class syntax. Here is an example of the most basic kind of class:

class Raven:

```
<myStatement1>  
<myStatement2>  
<myStatementN>
```

In a way, defining a class works nearly the same as defining a function. The class definitions are executed before we see the actual effect. However, keep in mind that the statements inside the definition will often be in the form of function definitions.

Regarding the syntax, as you can see we need to use the keyword “class” followed by the name of that class. In addition, you may have noticed that the name starts with a capitalized letter. The naming standard for classes doesn’t follow the same rules as the naming conventions for variables and functions. Keep in mind that this isn’t a rule set in stone that Python enforces in some way. You will not get an error if the name of the class starts with an underscore. However, this is the generally agreed upon convention among programmers and software developers and it is followed by most companies worldwide.

Now let’s use the class we created to build an object:

```
myObject = Raven()  
print (myObject.x)
```

We have created an object called “myObject” and printed its value “x”.

## ***Defining a Method***

Objects can contain methods, which in fact are functions that belong to them. Let’s look at an example:

```
class Man:
```

```
    def __init__(self, name, age)  
        self.name = name  
        self.age = age  
  
    def myfunction(self):  
        print ("Hi, I am " + self.name)  
  
x = Man ("Thomas", 25)  
x.myfunction()
```

Take note that we have a “self” parameter which refers to the current instance of our class. It is used to access any variable that belongs to it. Keep in mind that this doesn’t mean you can name it “self”, it is not a keyword. It is important to have it though, because we need it as the first parameter of a function that belongs to a class.

Another difference you may have noticed between this example and the basic syntax example is that we used a function called “`_init_`”. This type of function is known as a constructor. There are other constructors in Python and you can recognize them due to the underscores at the beginning and the end of the name.

Basic classes and objects aren’t that useful in real world applications, but with the help of this function we can give them a meaning. The “`init`” function is executed when the class is initialized and it is used to assign a number of values to an object. In addition, you can also assign any other operations that need to be performed with the creation of the object.

## Python Inheritance

Inheritance is one of the most powerful concepts in programming. It allows you to define a class that takes its functionality in order to define a new class. Keep in mind that the modifications you make to the new class, or subclass, don’t have to be extreme. You can even make a number of identical subclasses if you need to.

The new class that derives from the main class is called a child class. With that in mind, let’s take a look at the inheritance syntax:

```
class MainClass:
```

```
    #add functions and statements
```

```
class ChildClass (MainClass):
```

```
    #add functions and statements
```

This is how you can reuse code in Python. By following this structure, you can add or modify the features of your main class by creating a subclass out of it.

Take note that Python offers multiple types of inheritance:

1. Single inheritance: This is what we've worked with so far. When there's only one child class that inherits the features of its parent class, it's referred to as single inheritance.
2. Multiple inheritance: Unlike certain programming languages like C++, Python allows Multiple inheritance. This is the possibility of creating a child class which inherits the features of multiple parent classes. The syntax changes only slightly in the sense that you declare all parents by separating them with commas.
3. Multilevel inheritance: Let's say we have one child class that inherits the features of its parent class. Multilevel inheritance is when we have a second child class that inherits the features of the first child class. Sometimes this kind of class is referred to as a grandchild class.
4. Hierarchical inheritance: When we use one parent class to construct multiple subclasses we have hierarchical inheritance.
5. Hybrid inheritance: When we combine several types of inheritance we call it hybrid inheritance.

## Polymorphism

This translates to having multiple forms. In programming, polymorphism represents a language's ability to process objects based on their class or data type. Under this concept an object can redefine the methods for a derived class and therefore adapt its code to the type of data that it's processing.

Since polymorphism is often more difficult to grasp for a beginner because it's a somewhat advanced topic, let's go through a non-programming analogy and get a better understanding. Imagine you told someone to go and buy you something for lunch. If you say this to a 12 year old kid, he'll probably hop on his bicycle and spend his cash on a couple of hamburgers with fries. At the same time, if you say the exact same thing to your wife, she will probably drive her car to the store, buy you some veal and vegetables, and pay with her credit card. This is polymorphism in essence. You make a general specification such as go to the store and buy food. Then you have different objects implementing your request in different ways.

Keep in mind that this analogy is fairly simplistic and when you translate the specification to programming, it needs to be more detailed than in our example. However, the principle is the same.

Polymorphism is important in object oriented programming for two big reasons. Firstly, it allows one object to perform a varied implementation of a method depending on its parameters. Secondly, the code you write for one data type can be used on another related type. Polymorphism is the key to mastering Python programming.

Another important aspect of polymorphism that attempts to mimic the real world is the ability for a programming language to pass through the class hierarchy. In other words, it does something that it's used to performing on a regular basis because it learned it at some point in the past.

To describe this with another analogy, it's the same as you knowing how to read a book because it's something you already learned when you were younger. According to the rules of object oriented programming, you are an object and you can interact with a book, turn its page, and identify the meaning behind the letters. However, if you are capable of reading a book, you can also read web pages, which are the same as book pages but without the need for flipping the pages and physical interaction. You can also read street signs and perhaps texts in other languages as well. All of these examples are specialized versions of reading a book. If you can read a book, which in this case can be considered the base, or generic, object, then you can also read other things.

In programming languages, the above example shows that polymorphism allows code that is written for a certain type to also be used on different types that are derivative.

## **Creating Modules**

You already have some experience using Python modules, mainly importing them. However, one of Python's most powerful features is the ability to write your own modules.

Writing custom modules is another way of reusing or repurposing code and therefore save a great deal of time in the future. For example, you can use a number of common classes you already defined and reuse them. Let's say

you worked on some kind of card game and you had to create certain classes such as cards and deck. When you need to work on another card game, even if it's different, it will still require general classes that all of these games contain, such as cards and decks.

Another benefit of creating your own models is program management. When programs grow it's easy to get lost in them. Therefore you should break them up into small bits and pieces with the help of modules. Let's say you're working on a commercial project and you have thousands of lines of codes to deal with. This is to be expected when it comes to professional software, however, this is when you can take advantage of modules the most. Furthermore, if the program is divided this way, you can split various sections with other programmers and work on them simultaneously, thus saving a lot of time.

In addition to all of these advantages you will also be able to share your modules with your friends and fellow programmers. Whether you want to email your modules or upload them on a community forum, you can do so. Anyone can download and import your modules the same way they do with the default ones.

## ***Writing a Module***

In this section we are going to write a module that you can use in a simple game. Keep in mind that writing modules is no different than writing any other script or program. However, when you plan your module you should think about all the functions and classes you will need to define. Furthermore, once you have them, you need to place them in one file which can later be imported into your new projects.

With that in mind, let's create a simple module called "game". We will implement one class and two functions that fulfill common tasks you are likely to find in basic games. Let's take a look at the code:

```
# game  
# Example of writing a custom module  
class Player(object):  
    """ A basic player. """
```

```

def __init__(self, name, score = 0):
    self.name = name
    self.score = score
def __str__(self):
    rep = self.name + ":\t" + str(self.score)
    return rep
def ask_yes_or_no(question):
    """Ask a yes or no question."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
def ask_number(question, low, high):
    """Ask for a number within a range."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response
if __name__ == "__main__":
    print("This module was executed directly without importing.")
    input("\n\nPress the enter key to quit.")

```

This module we created will be called “games” because by default modules will carry the name of their file. If you look through the code itself you will already recognize pretty much everything, as we have implemented some of the functions we used in other examples. We have a “player” class that

defines an object. The object has two attributes and they are set in a constructor. However, there is one unique aspect to this module that we didn't explore before. We have an "if" statement which verifies whether the program is executed directly. This means that if the statement checks as true, the application will run directly, however if it's false, then it will be imported as a module.

## Summary

In this chapter you learned about one of Python's most important aspects, and that's object oriented programming. Keep in mind that the purpose of this chapter is to introduce you to OOP. This topic is fairly advanced for a beginner, so you shouldn't feel frustrated if you don't quite grasp the terminology or the theory.

In this short, introductory chapter to OOP you also learned about classes, polymorphism, and inheritance. All of these concepts are important when you plan to develop a complex program. Furthermore, you also learned more about modules and how to write your own. Don't forget that making your code reusable is one of your priorities as a programmer. If you wrote a good function or a module, save it because one day you will need it.

# **Chapter 8: Exceptions**

Projects never go as smooth as you expect no matter how skilled you are, how much you prepare, and how much you hope. As a programmer you need to expect the possibility of encountering exceptions.

In Python, exceptions are errors that come up when there's a problem with your code and the program cannot execute past the troublesome line. Fixing errors will be something that you will have to do quite often, and that is why in this chapter we are going to focus on learning how to deal with exceptions.

## **When Things Go Haywire**

Simple programs almost never display any erratic behavior due to the simplicity of code. Take a look at one of your earliest applications, such as “Hello World”. The programming behind it is so simple that if you run the program and it works, you can guarantee that it will always work no matter how many tests you perform. However, you've gathered enough knowledge to expand your programs and they are becoming more and more complex. You write many more lines of code and you are playing with more advanced concepts of programming, such as object oriented programming. With the complexity of your projects, your chances of encountering exceptions and errors increases dramatically. It is no longer enough for you to just look at your code and conclude that it will work as intended every single execution cycle.

Here's a simple example of a common issue. What happens if your application's configuration file doesn't have the right settings? Or perhaps it's missing altogether. Keep in mind that complicated programs with hundreds, if not thousands of lines of code will at some point contain corrupted data that is no longer accurate, or even usable. Furthermore, what if the code and the data are perfect, but your application is designed to connect to a web server in order to gather certain information, but the connection is unavailable? In some cases the error itself isn't that important and can even be ignored, however when it does matter, you need to handle it.

When Python encounters an abnormal situation that prevents the normal flow of instructions, you will see an exception. For the moment, if you do encounter one, you might be tempted to dig deep into your code and start writing some additional lines that would handle that event. However, this involves tedious work and it is a poor use of your time. Fortunately, Python offers you an exception handling methodology that is easy to write and maintain and doesn't take much of your time. It is a simple concept, and while there are other ways of handling exceptions, you will soon realize that you rarely need anything else besides exception handling.

## ***Basic Exceptions***

To demonstrate exception handling we're going to start with the most basic error, which is the attempt to divide by zero. This is a simple error that you probably wouldn't make because it's so obvious, however it is enough to trigger Python's exception handling system. Simply type the following line and then analyze the result:

```
1 / 0
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: int division or modulo by zero
```

If you recall, in an earlier chapter you encountered an error that generated a report that was quite similar to this one. Let's analyze the arithmetic operation first. As you should already know, a division requires a dividend and a divisor. It is one of the most basic operations, however, deep inside of it Python goes through a number of internal verifications. It checks whether the two arguments are numbers and whether the divisor is anything but zero. When the conditions are met, we get the result we expect. However, when the verification fails due to one of the checks, the operator relies on Python's exception handling. The result of a divisor being zero is the class called "ZeroDivisionError", which generates an exception object. In programming terms, this means that the operation raised an exception.

Pay attention to the meaning of the word "raised" because it refers to a certain issue going through a hierarchical order. What happens is that the

first function calls the second function which contains the division by zero and therefore the operator will raise the exception inside it. Since the second function can't deal with the exception on its own, it raises an exception with the first function. In the end, the exception can't be handled in any case so the interpreter raises the exception with you. Fortunately, the exception is as easy to read and understand as any Python code. The report you see at the end of our example is referred to as a traceback.

Keep in mind that in certain programming languages there is no exception raising. Instead, they are throwing or catching the exceptions. This means that the exception is thrown to the second function, but if it fails to catch it then it takes it to the first function. Basically, the process is the same but the terminology differs.

When you encounter an exception one of the easiest ways of dealing with it is implementing a “try” statement in your script. What the “try” block does is look for errors and then an “except” block will handle it. Here's a basic example of a statement that causes an error:

```
print(x)
```

Since we didn't define “x”, this will cause an error and without a “try” statement to raise it, the program will simply crash. Now let's handle the exception with the following lines:

```
try:
```

```
    print (x)
```

```
except:
```

```
    print (“An exception occurred”)
```

You can also define multiple exception statements if you are looking to catch multiple errors. Here's an example:

```
try:
```

```
    print (x)
```

```
except NameError:
```

```
    print (“There is no definition for variable x”)
```

```
except:
```

```
    print ("Something else went wrong")
```

If our try statement raises a `NameError` exception, the program will print the first message, but if any other error occurs, we will see the second message.

When handling exceptions through a “try” statement you can also use the “else” keyword to specify that you want the code to be executed if there aren’t any errors. Here’s an example:

```
try:
```

```
    print ("Hello World")
```

```
except:
```

```
    print ("An error occurred")
```

```
else:
```

```
    print ("Everything is fine")
```

## ***Exceptions Classes***

In Python we have a certain number of exception classes which inform the programmer what exactly went wrong during the execution of the code. Take note that these built-in classes are in fact subclasses of a main exception class “`BaseException`”. This means that every single exception you will work through is based on that class.

In the next example we are going to raise a number of exceptions in order to find out from which classes they inherit their features. To do this we are going to handle the exceptions by using an array in order to store them and then investigate their main classes. Here’s how the code looks:

```
store = []
# Create a number of different exceptions and handle them
try: {}["foo"]
except KeyError as e: store.append(e)
try: 1 / 0
```

```
except ZeroDivisionError as e: store.append(e)
try: """.bar()
except AttributeError as e: store.append(e)
# Loop through the errors and print the class ladder
for exception_object in store:
    ec = exception_object.__class__
    print(ec.__name__)
    indent = " +"
    while ec.__bases__:
        # Assign ec's superclass to itself and increase
        ec = ec.__bases__[0]
        print(indent + ec.__name__)
        indent = " " + indent
```

The result should look something like this:

```
KeyError
+-LookupError
+-Exception
+-BaseException
+-object
ZeroDivisionError
+-ArithmeticError
+-Exception
+-BaseException
+-object
AttributeError
+-Exception
+-BaseException
+-object
```

Now we have every exception's class hierarchy spawning from the main class. All of these exceptions you can find in Python's documentation library if you want to read more about each one of them. However, you will rarely need to work with a list of exceptions next to you. Typically, you will try to catch a certain exception and handle it individually or you create a general catch for every single one of them.

While each exception class that Python has can be found in this hierarchy, you need to take note that this also includes any custom exceptions. Yes,

you can even create your own exception classes that can inherit various features from their parent classes. We will not expand on this topic since a beginner programmer will not have a use for them in most scenarios.

## **Assertion Error**

The purpose of this type of error is to tell the programmer that there's some kind of error from which the program can't recover. They are not supposed to show a report like "variable is not defined", which would normally allow you to take an action. Assertions are essentially a self-check inside the code. They verify whether your program contains any conditions that are impossible. If any of these are encountered during the execution of the program, the application will crash with an assertion error which will tell you which impossible condition was the cause of it.

Assertion errors are great tools that let you know you have a bug and it needs fixing as soon as possible. In order to place an assertion you need to use the "assert" keyword. Normally, programmers place the statement at the beginning of a function to check the input, or at the end of a function to check the output. When Python detects an assert statement it will analyze the expression that's attached to it. If the evaluation turns a "true" value, then the function is good to go. However, if you get a "false" reading, you will find an assertion error exception.

Here's what the syntax looks like:

```
assert Expression [, Arguments]
```

Keep in mind that assertion errors can be handled exactly the same way as any other exception. Simply use the "try / except" statement. Just make sure you catch the assertion because otherwise the program will simply crash.

## **Summary**

In this chapter you learned about basic errors, exceptions, and the most common technique used to handle them. Add exception handling to your project development stage and take advantage of the "try / except" statement. Furthermore, you also learned about the assertion error which is a slightly more complex issue, but it can be resolved in fairly the same manner as the other exceptions.

# Bibliography

- Barry, P. (2016). Head first Python: a brain-friendly guide. Beijing: O'Reilly.
- Beazley, D., & Jones, B. K. (2013). *Python cookbook: recipes for mastering Python 3*. Beijing: O'Reilly.
- Gray, W. (2019). Learn Python programming: write code from scratch in a clear & concise way, with a complete basic course. From beginners to intermediate, an hands-on project with examples, to follow step by step.
- Rees, J. (2019). Python programming: a practical introduction to Python programming for total beginners. The Pragmatic Bookshelf.
- Ramalho, L. (2016). Fluent Python: Clear, Concise, and Effective Programming. Beijing: O'Reilly.
- Lutz, M. (2018). Learning Python: Powerful Object-Oriented Programming. Beijing: O'Reilly.

# **SQL**

THE ULTIMATE BEGINNER'S GUIDE  
TO LEARN SQL PROGRAMMING STEP BY  
STEP

**© Copyright 2018 - All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

**Legal Notice:**

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

# TABLE OF CONTENTS

## INTRODUCTION

1. Advantages of Databases
2. Types of SQL Queries

## CHAPTER 1: The Data Definition Language (DDL)

3. DDL for Database and Table Creation
4. Alter DDL for Foreign Key Addition
5. Foreign Key DDL in Tables
6. Unique Constraint DDL in Tables
7. Delete and Drop DDL in Tables
8. DDL to Create Views

## CHAPTER 2: SQL Joins and Union

1. SQL INNER JOIN
2. SQL RIGHT JOIN
3. SQL LEFT JOIN
4. SQL UNION Command
5. SQL UNION ALL Command

## CHAPTER 3: How to Ensure Data Integrity

- Integrity Constraints – The Basics
- The Not Null Constraint
- The Unique Constraint
- The PRIMARY KEY Constraint
- The FOREIGN KEY Constraints
- The CHECK Constraint

## CHAPTER 4: How to Create an SQL View

- How to Add a View to a Database
- How to Create an Updateable View
- How to Drop a View
- Database Security
- The Security Model of SQL
- Creating and Deleting a Role
- Granting and Revoking a Privilege

## CHAPTER 5: Database Creation

[Creating a Database](#)  
[Removing a Database](#)  
[Schema Creation](#)  
[Creating Tables and Inserting Data Into Tables](#)  
[How to Create a Table](#)  
[Creating a New Table Based on Existing Tables](#)  
[Inserting Data into Table](#)  
[Populating a Table with New Data](#)  
[Inserting Data into Specific Columns](#)  
[Inserting NULL Values](#)  
[Query Structure and SELECT Statement](#)  
[The WHERE Clause](#)  
[Using ORDER BY](#)  
[Data Definition Language \(DDL\)](#)  
[Applying DDL Statements](#)  
[Running the DDL Script](#)  
[Data Manipulation Language \(DML\)](#)  
[Running the DML Script](#)

## [CHAPTER 6: Database Administration](#)

[Recovery Models](#)  
[Database Backup Methods](#)  
[Database Restores](#)  
[Preparing to Restore the Database](#)  
[Database Restore Types](#)  
[Attaching and Detaching Databases](#)  
[Attaching/Detaching the AdventureWorks2012 Database](#)  
[Detaching the Database](#)  
[Attaching Databases](#)  
[Query Structure and SELECT Statement](#)  
[The WHERE Clause](#)  
[Using ORDER BY](#)

## [CHAPTER 7: SQL Transaction](#)

[What is an SQL Transaction?](#)  
[Commit](#)  
[Rollback](#)  
[Savepoint](#)  
[Release savepoint](#)

[Set Transaction](#)

[Start Transaction](#)

[Set Constraint](#)

## [CHAPTER 8: Logins, Users and Roles](#)

[Server Logins](#)

[Server Level Roles](#)

[Assigning Server Roles](#)

[Database Users](#)

[Database Level Roles](#)

[Assigning Database Roles and Creating Users](#)

[LIKE Clause](#)

[SQL Functions](#)

[SQL AVG Function](#)

[SQL ROUND Function](#)

[SQL SUM Function](#)

[SQL MAX\(\) Function](#)

[SQL MIN\(\) Function](#)

## [CHAPTER 9: Modifying and Controlling Tables](#)

[Modifying Column Attributes](#)

[Renaming Columns](#)

[Deleting a Column](#)

[Adding a New Column](#)

[Modifying an Existing Column without Changing its Name](#)

[Rules to Remember when Using ALTER TABLE](#)

[Deleting Tables](#)

[Combining and joining tables](#)

[Using Outer Join](#)

[Here's the result:](#)

[SQL Character Functions](#)

[Note:](#)

[SQL Constraints](#)

[NOT NULL Constraint](#)

[Default Constraint](#)

[Unique Constraint](#)

[Primary Key](#)

[Foreign Key](#)

[CHECK Constraint](#)

[INDEX Constraint](#)

[ALTER TABLE Command](#)

## [CONCLUSION AND NEXT STEPS](#)

[Product Review and Feedback](#)

[More Database Samples](#)

[Keep Learning](#)

[More References](#)

## [REFERENCES](#)

# INTRODUCTION

*“Never write when you can talk. Never talk when you can nod. And never put anything in an e-mail.” – Eliot Spitzer*

On a hard disk, data can be stored in different file formats. It can be stored in the form of text files, word files, mp4 files, etc. However, a uniform interface that can provide access to different types of data under one umbrella in a robust and efficient manner is required. Here, the role of databases emerges.

The definition of a database is “a collection of information stored in computer in a way that it can easily be accessed, managed and manipulated.”

Databases store data in the form of a collection of tables where each table stores data about a particular entity. The information that we want to store about students will be represented in the columns of the table. Each row of the table will contain the record of a particular student. Each record will be distinguished by a particular column, which will contain a unique value for each row.

Suppose you want to store the ID, name, age, gender, and department of a student. The table in the database that will contain data for this student will look like this:

| SID | SName | SAge | SGender | SDepartment |
|-----|-------|------|---------|-------------|
| 1   | Tom   | 14   | Male    | Computer    |
| 2   | Mike  | 12   | Male    | Electrical  |
| 3   | Sandy | 13   | Female  | Electrical  |
| 4   | Jack  | 10   | Male    | Computer    |
| 5   | Sara  | 11   | Female  | Computer    |

Student Table

Here, the letter “S” has been prefixed with the name of each column. This is just one of the conventions used to denote column names. You can give any name to the columns. (We will look at how to create tables and columns within it in the coming chapters.) It is much easier to access, manipulate,

and manage data stored in this form. SQL queries can be executed on the data stored in the form of tables that have relationships with other tables.

A database doesn't contain a single table. Rather, it contains multiple related tables. Relationships maintain database integrity and prevent data redundancy. For instance, if the school decides to rename the Computer department from "Computer" to "Comp & Soft," you will have to update the records of all students in the Computer department. You will have to update the 1st, 4th, and 5th records of the student table.

It is easy to update three records; however, in real life scenarios, there are thousands of students and it is an uphill task to update the records of all of them. In such scenarios, relationships between data tables become important. For instance, to solve the aforementioned redundancy problem, we can create another table named Department and store the records of all the departments in that table. The table will look like this:

| DID | DName      | DCapacity |
|-----|------------|-----------|
| 101 | Electrical | 800       |
| 102 | Computer   | 500       |
| 103 | Mechanical | 500       |

Department Table

Now, in the student table, instead of storing the department name, the department ID will be stored. The student table will be updated like this:

| SID | SName | SAge | SGender | DID |
|-----|-------|------|---------|-----|
| 1   | Tom   | 14   | Male    | 102 |
| 2   | Mike  | 12   | Male    | 101 |
| 3   | Sandy | 13   | Female  | 101 |
| 4   | Jack  | 10   | Male    | 102 |
| 5   | Sara  | 11   | Female  | 102 |

Table Student

You can see that the department name column has been replaced by the department id column, represented by "DID". The 1st, 4th, and 5th rows that were previously assigned the "Computer" department now contain the id of the department, which is 102. Now, if the name of the department is changed from "Computer" to "Comp & Soft", this change has to be made

only in one record of the department table and all the associated students will be automatically referred to the updated department name.

## **Advantages of Databases**

The following are some of the major advantages of databases:

- Databases maintain data integrity. This means that data changes are carried out at a single place and all the entities accessing the data get the latest version of the data.
- Through complex queries, databases can be efficiently accessed, modified, and manipulated. SQL is designed for this purpose.
- Databases avoid data redundancy. Through tables and relationships, databases avoid data redundancy and data belonging to particular entities resides in a single place in a database.
- Databases offer better and more controlled security. For example, usernames and passwords can be stored in tables with excessive security levels.

## **Types of SQL Queries**

On the basis of functionality, SQL queries can be broadly classified into a couple of major categories as follows:

- **Data Definition Language (DDL)**

Data Definition Language (DDL) queries are used to create and define schemas of databases. The following are some of the queries that fall in this category:

1. CREATE – to create tables and other objects in a database
2. ALTER – to alter database structures, mainly tables.
3. DROP - delete objects, mainly tables from the database

- **Data Manipulation Language**

Data Manipulation Language (DML) queries are used to manipulate data within databases. The following are some examples of DML queries.

1. SELECT – select data from tables of a database
2. UPDATE - updates the existing data within a table
3. DELETE - deletes all rows from a table, but the space for the record remains

# CHAPTER 1:

## The Data Definition Language (DDL)

*“Every man has a right to his opinion, but no man has a right to be wrong in his facts. “— Bernard Mannes Baruch*

SQL data definition language is used to define new databases, data tables, delete databases, delete data tables, and alter data table structures with the following key words; create, alter and drop. In this chapter, we will have a detailed discussion about the SQL Data Definition Language in a practical style.

### DDL for Database and Table Creation

The database creation language is used to create databases in a database management system. The language syntax is as described below:

*CREATE DATABASE my\_Database*

For example, to create a customer\_details database in your database management system, use the following SQL DDL statement:

*CREATE DATABASE customer\_details*

Please remember that SQL statement is case insensitive. Next, we need to create the various customer tables that will hold the related customers records, in the earlier created ‘customer\_details’ database. This is why the system is called a relational database management system, as all tables are related for easy record retrieval and information processing. To create the different but related customer tables in the customer\_details database, we apply the following DDL syntax:

```
CREATE TABLE my_table  
(  
    table_column-1  data_type,  
    table_column-2  data_type,  
    table_column-3  data_type,  
    table_column-n  data_type  
)
```

```
CREATE TABLE customer_accounts
(
    acct_no INTEGER, PRIMARY KEY,
    acct_bal  DOUBLE,
    acct_type INTEGER,
    acct_opening_date DATE
)1
```

The attribute “PRIMARY KEY” ensures the column named ‘acct\_no’ has unique values throughout, with no null values. Every table should have a primary key column to uniquely identify each record of the table. Other column attributes are ‘NOT NULL’ which ensures that a null value is not accepted into the column, and ‘FOREIGN KEY’ which ensures that a related record in another table is not mistakenly or unintentionally deleted. A column with a ‘FOREIGN KEY’ attribute is a copy of a primary key column in another related table. For example, we can create another table ‘customer\_personal\_info’ in our ‘customer\_details’ database like below:

```
CREATE TABLE customer_personal_info
(
    cust_id      INTEGER PRIMARY KEY,
    first_name   VARCHAR(100) NOT NULL,
    second_name  VARCHAR(100),
    lastname     VARCHAR(100) NOT NULL,
    sex          VARCHAR(5),
    date_of_birth DATE,
    address      VARCHAR(200)
)2
```

The newly created ‘customer\_personal\_info’ table has a primary key column named ‘cust\_id’. The ‘customer\_accounts’ table needs to include a column named ‘cust\_id’ in its field to link to the ‘customer\_personal\_info’ table in order to access the table for more information about the customer with a given account number.

Therefore, the best way to ensure data integrity between the two tables, so that an active record in a table is never deleted is to insert a key named ‘cust\_id’ in the ‘customer\_accounts’ table as a foreign key. This ensures that a related record in ‘customer\_personal\_info’ to another in ‘customer\_accounts’ table is never accidentally deleted. We will discuss how to go about this in the next section.

## Alter DDL for Foreign Key Addition

Since ‘customer\_accounts’ table is already created, we need to alter the table to accommodate the new foreign key. To achieve this, we use the SQL Data Definition Language syntax described below:

```
ALTER TABLE mytable
```

```
ADD FOREIGN KEY (targeted_column)
```

```
REFERENCES related_table(related_column)
```

Now, to add the foreign key to the ‘customer\_accounts’ table and make it reference the key column ‘cust\_id’ of table ‘customer\_personal\_info’, we use the following SQL statements:

```
ALTER TABLE customer_accounts
```

```
ADD FOREIGN KEY (cust_id)
```

```
REFERENCES customer_personal_info(cust_id)
```

## Foreign Key DDL in Tables

In situations where we need to create foreign keys as we create new tables, we make use of the following DDL syntax:

```
CREATE TABLE my_table
```

```
(
```

```
Column-1 data_type FOREIGN KEY, REFERENCES (related column)
```

```
)
```

## Unique Constraint DDL in Tables

The unique constraint can be placed on a table column to ensure that the values stored in the column are unique, just like the primary key column. The only difference is that you can only have one primary key column in a

table but as many unique columns as you like. The DDL syntax for the creation of a unique table column or field is as described below:

```
CREATE TABLE my_table  
(  
Column-1 data_type UNIQUE  
)
```

## Delete and Drop DDL in Tables

*DROP TABLE my\_table*

It must be noted that this action is irreversible and plenty of care must be taken before doing this. Also, a database can be deleted with the following DDL syntax:

*DROP DATABASE my\_database*

This action is also irreversible. At the atomic level, you may decide to delete a column from a data table. To achieve this, we use the ‘Delete’ DDL rather than ‘DROP’. The syntax is as below:

*DELETE column\_name FROM data\_table*

## DDL to Create Views

```
CREATE VIEW virtual_table AS  
SELECT column-1, column-2, ..., column-n  
FROM data_table  
WHERE column-n operator value3
```

For example, given a table (*customer\_personal\_info*) like the one below:

| cust_id | first_nm | second_nm | lastname | sex    | Date_of_birth |
|---------|----------|-----------|----------|--------|---------------|
| 03051   | Mary     | Ellen     | Brown    | Female | 1980-10-19    |
| 03231   | Juan     | John      | Maslow   | Female | 1978-11-18    |
| 03146   | John     | Ken       | Pascal   | Male   | 1983-07-12    |
| 03347   | Alan     | Lucas     | Basal    | Male   | 1975-10-09    |

Table 2.1 customer\_personal\_info

To create a view of female customers from the table, we use the following SQL Create View statement:

```
CREATE VIEW [Female Depositors] AS  
SELECT cust_id, first_nm, second_nm, lastname, sex, date_of_birth  
FROM customer_personal_info  
WHERE sex = 'Female'
```

To query and display the records of the created view, use the following select statement:

```
SELECT * FROM [Female Depositors]
```

The execution of the above select statement would generate the following view:

| <code>cust_id</code> | <code>first_nm</code> | <code>second_nm</code> | <code>lastname</code> | <code>sex</code> | <code>Date_of_birth</code> |
|----------------------|-----------------------|------------------------|-----------------------|------------------|----------------------------|
| 03051                | Mary                  | Ellen                  | Brown                 | Female           | 1980-10-19                 |
| 03231                | Juan                  |                        | Maslow                | Female           | 1978-11-18                 |

However, it must be noted that a view is never stored in memory but recreated when needed. A view can be processed just like you would process a real table.

# CHAPTER 2:

## SQL Joins and Union

*“I have noted that persons with bad judgment are most insistent that we do what they think best”. – Lionel Abe*

The select statement can be made to process more than one table at the same time in a single select statement with the use of the logical operators ('OR' and 'AND'). It can sometimes be more efficient to use the Left, right and the inner join operator for more efficient processing.

### 1. SQL INNER JOIN

The SQL ‘INNER JOIN’ can also be used to process more than one data table in an SQL query or statement. However, the data tables must have relating columns (the primary and its associated foreign key columns). The “INNER JOIN” keyword returns records from two data tables if a match is found between columns in the affected tables. The syntax of usage is as described below:

```
SELECT column-1, column-2... column-n  
FROM data_table-1  
INNER JOIN data_table-2  
ON data_table-1.keycolumn = data_table-2.foreign_keycolumn
```

For example, to select customer acct\_no, first\_name, surname, sex and account\_bal data from across the two tables ‘customer\_accounts’ (table 4.1) and ‘customer\_personal\_info’ (table 4.2) based on matching columns ‘cust\_id’, we use the following SQL INNER JOIN query:

| acct_no | cust_id | acct_bal | acct_type | acct_opening_date |
|---------|---------|----------|-----------|-------------------|
| 0411003 | 03231   | 2540.33  | 100       | 2000-11-04        |
| 0412007 | 03146   | 10350.02 | 200       | 2000-09-13        |
| 0412010 | 03347   | 7500.00  | 200       | 2002-12-05        |

Table 4.1 customer\_accounts

| cust_id | first_nm | second_nm | lastname | sex    | Date_of_birth | addr     |
|---------|----------|-----------|----------|--------|---------------|----------|
| 03051   | Mary     | Ellen     | Brown    | Female | 1980-10-19    | Coventry |
| 03231   | Juan     |           | Maslow   | Female | 1978-11-18    | York     |

|       |      |     |        |      |            |           |
|-------|------|-----|--------|------|------------|-----------|
| 03146 | John | Ken | Pascal | Male | 1983-07-12 | Liverpool |
| 03347 | Alan |     | Basal  | Male | 1975-10-09 | Easton    |

Table 4.2 customer\_personal\_info

```

SELECT a.acct_no, b.first_nm AS first_name, b.lastname AS
surname, b.sex, a.acct_bal
FROM customer_accounts AS a
INNER JOIN customer_personal_info AS b
ON b.cust_id = a.cust_id5

```

The above SQL query would produce the result set in table 4.3 below:

| acct_no | first_name | surname | sex    | acct_bal |
|---------|------------|---------|--------|----------|
| 0411003 | Juan       | Maslow  | Female | 2540.33  |
| 0412007 | John       | Pascal  | Male   | 10350.02 |
| 0412010 | Alan       | Basal   | Male   | 7500.00  |

Table 4.3

## SQL RIGHT JOIN

When used with the SQL select statement, The “RIGHT JOIN” keyword includes all records in the right data table even when no matching records are found in the left data table. The syntax of usage is as described below:

```
SELECT column-1, column-2... column-n
```

```
FROM left-data_table
```

```
RIGHT JOIN right-data_table
```

```
ON left-data_table.keyColumn = right-data_table.foreign_keycolumn
```

For example, to select customers’ acct\_no, first\_name, surname, sex and account\_bal details across the two tables, customer\_accounts and customer\_personal\_info display customer personal information whether they have an active account or not. We use the following SQL ‘RIGHT JOIN’ query:

```

SELECT a.acct_no, b.first_nm AS first_name, b.lastname AS surname,
b.sex, a.acct_bal

```

```
FROM customer_accounts AS a
```

```
RIGHT JOIN customer_personal_info AS b
```

ON b.cust\_id = a.cust\_id<sup>6</sup>

The output of execution of the above SQL RIGHT JOIN query is presented in table 4.4 below:

| acct_no | first_name | surname | sex    | acct_bal |
|---------|------------|---------|--------|----------|
| 0411003 | Juan       | Maslow  | Female | 2540.33  |
| 0412007 | John       | Pascal  | Male   | 10350.02 |
| 0412010 | Alan       | Basal   | Male   | 7500.00  |
|         | Mary       | Brown   | Female |          |

Table 4.4

## SQL LEFT JOIN

The ‘LEFT JOIN’ is used with the SQL select statement to return all records in the left data table (first table) even if there were no matches found in the relating right table (second table). The syntax of usage is as described below:

SELECT column-1, column-2... column-n

FROM left-data\_table

LEFT JOIN right-data\_table

ON left-data\_table.keyColumn = right-data\_table.foreign\_keycolumn

For example, to display all active accounts holders’ details across the ‘customer\_accounts’ and ‘customer\_personal\_info’ tables, we use the following SQL ‘LEFT JOIN’ query:

SELECT a.acct\_no, b.first\_nm AS first\_name, b.lastname AS surname, b.sex, a.acct\_bal

FROM customer\_accounts AS a

LEFT JOIN customer\_personal\_info AS b

ON b.cust\_id = a.cust\_id<sup>7</sup>

The above SQL LEFT JOIN query would produce the result set in table 4.5 below:

| acct_no | first_name | surname | sex    | acct_bal |
|---------|------------|---------|--------|----------|
| 0411003 | Juan       | Maslow  | Female | 2540.33  |

|         |      |        |      |          |
|---------|------|--------|------|----------|
| 0412007 | John | Pascal | Male | 10350.02 |
| 0412010 | Alan | Basal  | Male | 7500.00  |

Table 4.5

## SQL UNION Command

Two or more SQL select statements' result sets can be joined with the 'UNION' command. The syntax of usage is as described below:

SELECT column-1, column-2... column-n

FROM data\_table-1

UNION

SELECT column-1, column-2... column-n

FROM data\_table-2

Source [8](#)

For example, to display a list of our phantom bank's customers in UK and US branches, you may display one record for customers that have accounts in both branches according to the following tables 4.6 and 4.7:

| acct_no | first_name | surname | sex    | acct_bal |
|---------|------------|---------|--------|----------|
| 0411003 | Juan       | Maslow  | Female | 2540.33  |
| 0412007 | John       | Pascal  | Male   | 10350.02 |
| 0412010 | Alan       | Basal   | Male   | 7500.00  |

Table 4.6 London\_Customers

| acct_no | first_name | surname | sex    | acct_bal  |
|---------|------------|---------|--------|-----------|
| 0413112 | Deborah    | Johnson | Female | 4500.33   |
| 0414304 | John       | Pascal  | Male   | 13360.53  |
| 0414019 | Rick       | Bright  | Male   | 5500.70   |
| 0413014 | Authur     | Warren  | Male   | 220118.02 |

Table 4.7 Washington\_Customers

We use the following SQL query:

```
SELECT first_name, surname, sex, acct_bal
FROM London_Customers
```

## UNION

```
SELECT first_name, surname, sex, acct_bal  
FROM Washington_Customers9
```

The following is the result set from the execution of the above SQL query:

| first_name | surname | sex    | acct_bal  |
|------------|---------|--------|-----------|
| Juan       | Maslow  | Female | 2540.33   |
| John       | Pascal  | Male   | 10350.02  |
| Alan       | Basal   | Male   | 7500.00   |
| Deborah    | Johnson | Female | 4500.33   |
| Rick       | Bright  | Male   | 5500.70   |
| Authur     | Warren  | Male   | 220118.02 |

Table 4.8 SQL UNION

Note that the record for a customer (John Pascal) is only listed once, even if he is a customer of the two branches. This is because the UNION command lists only distinct records across tables. So, to display all records across associated tables, use ‘UNION ALL’ instead.

## SQL UNION ALL Command

The UNION ALL command is basically the same as the UNION command, except that it displays all records across unionized tables, as explained earlier. The syntax of usage is as described below:

```
SELECT column-1, column-2... column-n
```

```
FROM data_table-1
```

```
UNION
```

```
SELECT column-1, column-2... column-n
```

```
FROM data_table-2
```

Apply the ‘UNION ALL’ command on the data of the two tables ‘London\_Customers’ and ‘Washington\_Customers’ with the SQL query below:

```
SELECT first_name, surname, sex, acct_bal  
FROM London_Customers
```

UNION ALL

```
SELECT first_name, surname, sex, acct_bal  
FROM Washington_Customers10
```

The records would then be displayed in two tables, as shown below:

| first_name | surname | sex    | acct_bal  |
|------------|---------|--------|-----------|
| Juan       | Maslow  | Female | 2540.33   |
| John       | Pascal  | Male   | 10350.02  |
| Alan       | Basal   | Male   | 7500.00   |
| Deborah    | Johnson | Female | 4500.33   |
| John       | Pascal  | Male   | 13360.53  |
| Rick       | Bright  | Male   | 5500.70   |
| Authur     | Warren  | Male   | 220118.02 |

Table 4.9

# CHAPTER 3:

## How to Ensure Data Integrity

*“The possession of facts is knowledge, the use of them is wisdom.” – Thomas Jefferson*

SQL databases don't just store information. If the information's integrity has been compromised, its reliability becomes questionable. If the data is unreliable, the database that contains it also becomes unreliable.

To secure data integrity, SQL offers a wide range of rules that can limit the values a table can hold. These rules, known as "integrity constraints," work on columns and tables. This chapter will explain each kind of constraint.

### Integrity Constraints – The Basics

SQL users divide integrity constraints into the following categories:

The Assertions – You need to define this constraint inside a separate definition (which is called the “assertion definition”). This means that you don't indicate an assertion in your table's definition. In SQL, you may apply an assertion to multiple tables.

The Table-Related Constraints – This is a constraint that you need to define inside a table's definition. You may define a constraint as a component of a table or column's definition.

The Domain Constraints – Similar to the assertions, you need to create domain constraints in a separate definition. This kind of constraint works on the column/s that you declared inside the domain involved.

Table-related constraints offer various constraint options. Consequently, these days, it is the most popular category of integrity constraints. You can divide this category into two: column constraints and table constraints. The former belong to the definition of a column. The latter, on the other hand, act as elements of a table.

The table and column constraints work with different kinds of constraints. The domain constraints and assertions, however, can only work with one constraint type.

### The Not Null Constraint

In the previous chapter, you learned that “null” represents an unknown/undefined value. Keep in mind that undefined/unknown is different from zeroes, blanks, default values, and empty strings. Rather, it signifies the absence of a value. You may consider this value as a “flag” (i.e. a bit, number, or character that expresses some data regarding a column). If you leave a column empty, and the value is therefore null, the system will place the “flag” to indicate that it’s an unknown value.

Columns have an attribute called “nullability.” This attribute shows whether the columns can take unknown values or not. In SQL, columns are set to take null values. However, you may change this attribute according to your needs. To disable the nullability of a column, you just have to use the NOT NULL constraint. This constraint informs SQL that the column won’t accept any null value.

In this language, you need to use NOT NULL on a column. That means you can't use this constraint on an assertion, domain constraint, or table-based constraint. Using NOT NULL is a simple process. Just add the syntax below to your column definition:

(name of column) [ (domain) | (data type) ] NOT NULL

As an example, let's assume that you need to generate a table called FICTION\_NOVEL\_AUTHORS. This table needs to have three columns: AUTHOR\_ID, AUTHOR\_NAME, and AUTHOR\_DOB. You need to ensure that each entry you add has values for AUTHOR\_ID and AUTHOR\_NAME. To accomplish this, you must insert the NOT NULL constraint into the definition of both columns. Here's the code:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
( AUTHOR_ID      INT      NOT NULL ,  
  AUTHOR_NAME     CHARACTER(50) NOT NULL ,  
  AUTHOR_DOB      CHARACTER(50) );11
```

As you can see, this code did not set NOT NULL for the AUTHOR\_DOB column. Consequently, if a new entry doesn't have any value for AUTHOR\_DOB, the system will insert a null value in that column.

## The Unique Constraint

Table and column constraints accept unique constraints. In SQL, unique constraints belong to one of these two types:

1. UNIQUE
2. PRIMARY KEY

Important Note: This part of the book will concentrate on the first type. You'll learn about the second one later.

Basically, you can use UNIQUE to make sure that a column does not accept duplicate values. This constraint will stop you from entering a value that already exists in the column.

Let's assume that you want to apply this constraint on the AUTHOR\_DOB column. This way, you can make sure that the values inside that column are all unique. Now, let's say you realized that requiring dates of birth to be unique is a bad idea since people may be born on the same date. You may adjust your approach by placing the UNIQUE constraint on AUTHOR\_NAME and AUTHOR\_DOB. Here, the table will stop you from repeating an AUTHOR\_NAME/AUTHOR\_DOB pair. You may repeat values in the AUTHOR\_NAME and AUTHOR\_DOB columns. However, you can't reenter an exact pair that already exists in the table.

Keep in mind that you may tag UNIQUE constraints as table constraints or column constraints. To generate column constraints, add them to the definition of a column. Here is the syntax:

(name of column) [ (domain) | (data type) ] UNIQUE

If you need to use the UNIQUE constraint on a table, you must insert it into the table definition as an element. Here is the code:

{ CONSTRAINT (name of constraint) }

UNIQUE < (name of column) { [, (name of column) ] ... } >

As the syntax above shows, using UNIQUE on a table is more complicated than using the constraint on a column. However, you cannot apply UNIQUE on multiple columns. Regardless of how you use this constraint (i.e. either as a table constraint or a column constraint), you may define any number of UNIQUE constraints within each table definition.

Let's apply this constraint on a columnar level:

```
CREATE TABLE BOOK_LIBRARY  
( AUTHOR_NAME    CHARACTER (50) ,  
  BOOK_TITLE     CHARACTER (70) UNIQUE,  
  PUBLISHED_DATE INT )12;
```

You may also use UNIQUE on other columns. However, its result would be different than if we had used a table constraint on multiple columns. The following code will illustrate this idea:

```
CREATE TABLE BOOK_LIBRARY  
( AUTHOR_NAME    CHARACTER (50) ,  
  BOOK_TITLE     CHARACTER (70) ,  
  PUBLISHED_DATE INT,  
  CONSTRAINT UN_AUTHOR_BOOK UNIQUE ( AUTHOR_NAME,  
  BOOK_TITLE ))13;
```

Now, for the table to accept a new entry, the AUTHOR\_NAME and BOOK\_TITLE columns must have unique values.

As you've read earlier, the UNIQUE constraint ensures that one or more columns do not have duplicate values. That is an important rule to remember. However, you should also know that UNIQUE doesn't work on "null." Thus, a column will accept any number of null values even if you have set a UNIQUE constraint on it.

If you want to set your columns not to accept a null value, you must use NOT NULL. Let's apply NOT NULL on the column definition of BOOK\_TITLE:

```
CREATE TABLE BOOK_LIBRARY  
( AUTHOR_NAME    CHARACTER (50) ,  
  BOOK_TITLE     CHARACTER (70)   UNIQUE NOT NULL,  
  PUBLISHED_DATE INT )14;
```

In SQL, you may also insert NOT NULL into column definitions that a table-level constraint is pointing to:

```
CREATE TABLE BOOK_LIBRARY
```

```
( AUTHOR_NAME    CHARACTER (50) ,  
BOOK_TITLE     CHARACTER (70) NOT NULL,  
PUBLISHED_DATE INT,  
CONSTRAINT UN_AUTHOR_BOOK UNIQUE (BOOK_TITLE) ) ;
```

Source [https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp)

In both cases, the BOOK\_TITLE column gets the constraint. That means BOOK\_TITLE won't accept null or duplicate values.

## The PRIMARY KEY Constraint

The PRIMARY KEY constraint is almost identical to the UNIQUE constraint. You may use a PRIMARY KEY to prevent duplicate entries. In addition, you may apply it to multiple columns and use it as a table constraint or a column constraint. The only difference is that PRIMARY KEY has two distinct restrictions. These restrictions are:

If you apply PRIMARY key on a column, that column won't accept any null value. Basically, you won't have to use the NOT NULL constraint on a column that has PRIMARY KEY.

A table can't have multiple PRIMARY KEY constraints.

These restrictions exist because primary keys (also known as “unique identifiers”) play an important role in each table. As discussed in the first chapter, tables cannot have duplicate rows. This rule is crucial since the SQL language cannot identify redundant rows. If you change a row, all of its duplicates will also be affected.

You need to choose a primary key from the candidate keys of your database. Basically, candidate keys are groups of columns that identify rows in a unique manner. You may enforce a candidate key's uniqueness using UNIQUE or PRIMARY KEY. However, you must place one primary key on each table even if you did not define any unique constraint. This requirement ensures the uniqueness of each data row.

To define a primary key, you need to indicate the column/s you want to use. You can complete this task through PRIMARY KEY (i.e. the SQL keyword). This process is similar to the one discussed in the previous section. To apply PRIMARY KEY on a new column, use the following syntax:

(name of column) [ (domain) | (data type) ] PRIMARY KEY

To use PRIMARY key on a table, you must enter it as an element of the table you're working on. Check the syntax below:

```
{ CONSTRAINT (name of constraint) }
```

```
PRIMARY KEY <(name of column) {, (name of column) } ... >
```

SQL allows you to define primary keys using column constraints. However, you can only use this feature on a single column. Analyze the following example:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(AUTHOR_ID INT,  
AUTHOR_NAME CHARACTER (50) PRIMARY KEY ,  
PUBLISHER_ID INT )15;
```

If you want to apply PRIMARY KEY on multiple columns (or store it as another definition), you may use it on the tabular level:

```
CREATE TABLE FICTION_NOVEL_AUTHORS  
(AUTHOR_ID INT,  
AUTHOR_NAME CHARACTER (50) ,  
PUBLISHER_ID INT,  
CONSTRAINT PK_AUTHOR_ID PRIMARY KEY (AUTHOR_ID,  
AUTHOR_NAME ));
```

This approach places a primary key on two columns (i.e. AUTHOR\_ID and AUTHOR\_NAME). That means the paired values of the two columns need to be unique. However, duplicate values may exist inside any of the columns. Experienced database users refer to this kind of primary key as a “superkey.” The term “superkey” means that the primary key exceeds the number of required columns.

In most cases, you need to set both UNIQUE and PRIMARY KEY constraints on a table. To achieve this, you just have to define the involved constraints, as usual. For instance, the code given below applies both of these constraints:

```
CREATE TABLE FICTION_NOVEL_AUTHORS
```

```

(AUTHOR_ID INT,
AUTHOR_NAME CHARACTER (50) PRIMARY KEY ,
PUBLISHER_ID INT,
CONSTRAINT UN_AUTHOR_NAME UNIQUE (AUTHOR_NAME) );

```

The following code will give you the same result:

```

CREATE TABLE FICTION_NOVEL_AUTHORS
(AUTHOR_ID INT,
AUTHOR_NAME CHARACTER (50) -> UNIQUE,
PUBLISHER_ID INT,
CONSTRAINT PK_PUBLISHER_ID PRIMARY KEY (PUBLISHER_ID)
);16

```

## The FOREIGN KEY Constraints

The constraints discussed so far focus on securing the data integrity of a table. NOT NULL stops columns from taking null values. PRIMARY KEY and UNIQUE, on the other hand, guarantee that the values of one or more columns are unique. In this regard, FOREIGN KEY (i.e. another SQL constraint) is different. FOREIGN KEY, also called "referential constraint," focuses on how information inside a table works with the information within another table.

This connection ensures information integrity throughout the database. In addition, the connection between different tables results to "referential integrity." This kind of integrity makes sure that data manipulation done on one table doesn't affect the data inside other tables. The tables given below will help you understand this topic. Each of these tables, named PRODUCT\_NAMES and PRODUCT\_MANUFACTURERS, have one primary key:

|                  | PRODUCT_NAMES  |                   |
|------------------|----------------|-------------------|
| PRODUCT_NAME_ID: | PRODUCT_NAME:  | MANUFACTU RER_ID: |
| INT              | CHARACTER (50) | INT               |
| 1001             | X Pen          | 91                |
| 1002             | Y Eraser       | 92                |

### PRODUCT\_MANUFACTURERS

| MANUFACTURER_ID: INT | BUSINESS_NAME: CHARACTER (50) |
|----------------------|-------------------------------|
| 91                   | THE PEN MAKERS INC.           |
| 92                   | THE ERASER MAKERS INC.        |
| 93                   | THE NOTEBOOK MAKERS INC.      |

The PRODUCT\_NAME\_ID column of the PRODUCT\_NAMES table has a PRIMARY KEY. The MANUFACTURER\_ID of the PRODUCT\_MANUFACTURERS table has the same constraint. These columns are in yellow (see the tables above).

As you can see, the PRODUCT\_NAMES table has a column called MANUFACTURER\_ID. That column has the values of a column in the PRODUCT\_MANUFACTURERS table. Actually, the MANUFACTURER\_ID column of the PRODUCT\_NAMES table can only accept values that come from the MANUFACTURER\_ID column of the PRODUCT\_MANUFACTURERS table.

Additionally, the changes that you'll make on the PRODUCT\_NAMES table may affect the data stored in the PRODUCT\_MANUFACTURERS table. If you remove a manufacturer, you also need to remove the entry from the MANUFACTURER\_ID column of the PRODUCT\_NAMES table. You can achieve this by using FOREIGN KEY. This constraint ensures the referential integrity of your database by preventing actions on any table from affecting the protected information.

Important Note: If a table has a foreign key, it is called “referencing table.” The table a foreign key points to is called “referenced table.”

When creating this kind of constraint, you need to obey the following guidelines:

You must define a referenced column by using PRIMARY KEY or UNIQUE. Most SQL programmers choose PRIMARY KEY for this purpose.

You may tag FOREIGN KEY constraints as column constraints or table constraints. You may work with any number of columns if you are using

FOREIGN KEY as a table constraint. On the other hand, if you use this constraint at the column-level, you can only work on a single column.

A referencing table's foreign key should cover all of the columns you are trying to reference. In addition, the columns of the referencing table should match the data type of their counterparts (i.e. the columns being referenced). However, you don't have to use the same names for your referencing and referenced columns.

You don't need to indicate reference columns manually. If you don't specify any column for the constraint, SQL will consider the columns of the referenced table's primary key as the referenced columns. This process happens automatically.

You will understand these guidelines once you have analyzed the examples given below. For now, let's analyze the syntax of this constraint. Here's the format that you must use to apply FOREIGN KEY at the columnar level:

```
(name of column) [ (domain) | (data type) ] { NOT NULL }  
REFERENCES (name of the referenced table) { < (the referenced columns)  
> }  
{ MATCH [ SIMPLE | FULL | PARTIAL ] }  
{ (the referential action) }17
```

To use this FOREIGN KEY as a tabular constraint, you need to insert it as a table element. Here's the syntax:

```
{ CONSTRAINT (name of constraint) }  
FOREIGN KEY < (the referencing column) { [, (the referencing column) ]  
... } >  
REFERENCES (the referenced table) { < (the referenced column/s) > }  
{ MATCH [ SIMPLE | FULL | PARTIAL ] }  
{ (the referential action) }18
```

You've probably noticed that FOREIGN KEY is more complex than the constraints you've seen so far. This complexity results from the constraint's option-filled syntax. However, generating this kind of constraint is easy and simple. Let's first analyze a basic example:

```
CREATE TABLE PRODUCT_NAMES
```

```

( PRODUCT_NAME_ID -> INT,
PRODUCT_NAME -> CHARACTER (50) ,
MANUFACTURER_ID      ->      INT      ->      REFERENCES
PRODUCT_MANUFACTURERS ) ;

```

This code applies the constraint on the MANUFACTURER\_ID column. To apply this constraint on a table, you just have to type REFERENCES and indicate the referenced table's name. In addition, the columns of this foreign key are equal to that of the referenced table's primary key. If you don't want to reference your target's primary key, you need to specify the column/s you want to use. For instance, REFERENCES PRODUCT\_MANUFACTURERS (MANUFACTURER\_ID).

Important Note: The FOREIGN KEY constraint requires an existing referenced table. In addition, that table must have a PRIMARY KEY or UNIQUE constraint.

For the second example, you will use FOREIGN KEY as a tabular constraint. The code that you see below specifies the referenced column's name, even if that information is not required.

```

CREATE TABLE PRODUCT_NAMES
( PRODUCT_NAME_ID   INT,
PRODUCT_NAME   CHARACTER (50) ,
MANUFACTURER_ID   INT,
CONSTRAINT      TS_MANUFACTURER_ID      FOREIGN      KEY
(MANUFACTURER_ID)
REFERENCES          PRODUCT_MANUFACTURERS
(MANUFACTURER_ID) )19 ;

```

You may consider the two lines at the bottom as the constraint's definition. The constraint's name, TS\_MANUFACTURER\_ID, comes after the keyword CONSTRAINT. You don't need to specify a name for your constraints since SQL will generate one for you in case this information is missing. On the other hand, you may want to set the name of your constraint manually since that value appears in errors (i.e. when SQL commands violate an existing constraint). In addition, the names you will provide are more recognizable than system-generated ones.

Next, you should set the kind of constraint you want to use. Then, enter the name of your referencing column (MANUFACTURER\_ID for the current example). You will then place the constraint on that column. If you are dealing with multiple columns, you must separate the names using commas. Afterward, type REFERENCES as well as the referenced table's name. Finally, enter the name of your referenced column.

That's it. Once you have defined this constraint, the MANUFACTURER\_ID column of PRODUCT\_NAMES won't take values except those that are already listed in the PRODUCT\_MANUFACTURERS table's primary key. As you can see, a foreign key doesn't need to hold unique values. You may repeat the values inside your foreign keys as many times as you want, unless you placed the UNIQUE constraint on the column you're working on.

Now, let's apply this constraint on multiple columns. You should master this technique before studying the remaining elements of the constraint's syntax. For this example, let's use two tables: BOOK\_AUTHORS and BOOK\_GENRES.

The table named BOOK\_AUTHORS has a primary key defined in the AUTHOR\_NAME and AUTHOR\_DOB columns. The SQL statement found below generates a table called BOOK\_GENRES. This table has a foreign key consisting of the AUTHOR\_DOB and DATE\_OF\_BIRTH columns.

```
CREATE TABLE BOOK_GENRES
(AUTHOR_NAME  CHARACTER (50),
DATE_OF_BIRTH  DATE,
GENRE_ID  INT,
CONSTRAINT  TS_BOOK_AUTHORS  FOREIGN  KEY  (
AUTHOR_NAME,      DATE_OF_BIRTH      )  REFERENCES
BOOK_AUTHORS (AUTHOR_NAME, AUTHOR_DOB) );20
```

This code has a pair of referenced columns (i.e. AUTHOR\_NAME, AUTHOR\_DOB) and a pair of referencing columns (i.e. AUTHOR\_NAME and DATE\_OF\_BIRTH). The columns named AUTHOR\_NAME inside the data tables contain the same type of data. The data type of the DATE\_OF\_BIRTH column is the same as that of AUTHOR\_DOB. As this

example shows, the name of a referenced column doesn't need to match that of its referencing counterpart.

## The MATCH Part

Now, let's discuss another part of the constraint's syntax:

{ MATCH [ SIMPLE | FULL | PARTIAL ] }

The curly brackets show that this clause is optional. The main function of this clause is to let you choose how to handle null values inside a foreign key column, considering the values that you may add to a referencing column. This clause won't work on columns that don't accept null values.

This part of the syntax offers three choices:

**SIMPLE** – If you choose this option, and at least one of your referencing columns has a null value, you may place any value on the rest of the referencing columns. The system will automatically trigger this option if you don't specify the MATCH section of your FOREIGN KEY's definition.

**FULL** – This option requires all of your referencing columns to accept null values; otherwise, none of them can accept a null value.

**PARTIAL** – With this option, you may place null values on your referencing columns if other referencing columns contain values that match their respective referenced columns.

## The (referential action) Part

This is the final section of the FOREIGN KEY syntax. Just like the MATCH part, “referential action” is completely optional. You can use this clause to specify which actions to take when updating or removing information from one or more referenced columns.

For example, let's assume that you want to remove an entry from the primary key of a table. If a foreign key references the primary key you're working on, your desired action will violate the constraint. So, you should always include the data of your referencing columns inside your referenced columns.

When using this clause, you will set a specific action to the referencing table's definition. This action will occur once your referenced table gets changed.

ON UPDATE (the referential action) { ON DELETE (the referential action)  
} | ON DELETE (the referential action) { ON UPDATE (the referential  
action) } (the referential action) ::=

RESTRICT | SET NULL | CASCADE | NO ACTION | SET DEFAULT

According to this syntax, you may set ON DELETE, ON UPDATE, or both. These clauses can accept one of the following actions:

**RESTRICT** – This referential action prevents you from performing updates or deletions that can violate the FOREIGN KEY constraint. The information inside a referencing column cannot violate FOREIGN KEY.

**SET NULL** – This action changes the values of a referencing column to "null" if its corresponding referenced column gets removed or updated.

**CASCADE** – With this referential action, the changes you'll apply on a referenced column will also be applied to its referencing column.

**NO ACTION** – Just like RESTRICT, NO ACTION stops you from performing actions that will violate FOREIGN KEY. The main difference is that NO ACTION allows data violations while you are executing an SQL command. However, the information within your foreign key will not be violated once the command has been executed.

**SET DEFAULT** – With this option, you may set a referencing column to its default value by updating or deleting the data inside the corresponding referenced column. This referential action won't work if your referencing columns don't have default values.

To use this clause, you just have to insert it to the last part of a FOREIGN KEY's definition. Here's an example:

```
CREATE TABLE AUTHORS_GENRES
( AUTHOR_NAME    CHARACTER (50) ,
  DATE_OF_BIRTH   DATE,
  GENRE_ID        INT,
  CONSTRAINT      TS_BOOK_AUTHORS   FOREIGN KEY (
    AUTHOR_NAME,      DATE_OF_BIRTH      ) REFERENCES
    BOOK_AUTHORS ON DELETE RESTRICT ON UPDATE RESTRICT )
;
```

## The CHECK Constraint

You can apply this constraint on a table, column, domain, or inside an assertion. This constraint lets you set which values to place inside your columns. You may use different conditions (e.g. value ranges) that define which values your columns may hold.

According to SQL programmers, the CHECK constraint is the most complex and flexible constraint currently available. However, this constraint has a simple syntax. To use CHECK as a column constraint, add the syntax below to your column definition:

```
(name of column) [ (domain) | (data type) ] CHECK <(the search condition)>
```

If you want to use this constraint on a table, insert the syntax below to your table's definition:

```
{ CONSTRAINT (name of constraint) } CHECK <(the search condition)>
```

Important Note: You'll later learn how to use this constraint on assertions and domains.

As this syntax shows, CHECK is easy to understand. However, its search condition may involve complex and extensive values. This constraint tests the assigned search condition for the SQL commands that try to alter the information inside a column protected by CHECK. If the result of the test is TRUE, the commands will run; if the result is false, the system will cancel the commands and display error messages.

You need to analyze examples in order to master this clause. However, almost all components of the search condition involve predicates. Predicates are expressions that work on values. In SQL, you may use a predicate to compare different values (e.g. COLUMN\_3 < 5). The “less than” predicate checks whether the values inside COLUMN\_3 are less than 5.

Most components of the search condition also utilize subqueries. Basically, subqueries are expressions that act as components of other expressions. You use a subquery if an expression needs to access or compute different layers of information. For instance, an expression might need to access TABLE\_X to insert information to TABLE\_Z.

In the example below, CHECK defines the highest and lowest values that you may enter in a column. This table definition generates a CHECK constraint and three columns:

```
CREATE TABLE BOOK_TITLES
( BOOK_ID      INT,
BOOK_TITLE     CHARACTER (50) NOT NULL,
STOCK_AVAILABILITY  INT,
CONSTRAINT TS_STOCK_AVAILABILITY ( STOCK_AVAILABILITY
< 50 AND STOCK_AVAILABILITY > 1 ) );21
```

The resulting table will reject values that are outside the 1-50 range. Here's another way to write the table:

```
CREATE TABLE BOOK_TITLES
( BOOK_ID      INT,
BOOK_TITLE     CHARACTER (50) NOT NULL,
STOCK_AVAILABILITY      INT CHECK (
STOCK_AVAILABILITY < 50 AND STOCK_AVAILABILITY > 1 ));
```

Now, let's analyze the condition clause of these statements. This clause tells SQL that all of the values added to the STOCK\_AVAILABILITY column must be lower than 50. The keyword AND informs SQL that there's another condition that must be applied. Finally, the clause tells SQL that each value added to the said column should be higher than 1. To put it simply, each value should be lower than 50 and higher than 1.

This constraint also allows you to simply list your “acceptable values.” SQL users consider this a powerful option when it comes to values that won't be changed regularly. In the next example, you will use the CHECK constraint to define a book's genre:

```
CREATE TABLE BOOK_TITLES
( BOOK_ID      INT,
BOOK_TITLE     CHARACTER (50) ,
GENRE        CHAR (10) ,
```

```
CONSTRAINT TS_GENRE CHECK ( GENRE IN ( ' DRAMA ' , ' HORROR ' , ' SELF HELP ' , ' ACTION ' , ' MYSTERY ' , ' ROMANCE ' )  
));22
```

Each value inside the GENRE column should be included in the listed genres of the condition. As you can see, this statement uses IN (i.e. an SQL operator). Basically, IN makes sure that the values within GENRE are included in the listed entries.

This constraint can be extremely confusing since it involves a lot of parentheses. You may simplify your SQL codes by dividing them into multiple lines. As an example, let's rewrite the code given above:

```
CREATE TABLE BOOK_TITLES  
(  
    BOOK_ID      INT,  
    BOOK_TITLE   CHAR (50) ,  
    GENRE        CHAR (10) ,  
    CONSTRAINT TS_GENRE CHECK  
    (  
        GENRE IN  
        ( 'DRAMA ' , ' HORROR ' , ' SELF HELP ' , ' ACTION ' , ' MYSTERY ' , ' ROMANCE '  
    )  
);
```

This style of writing SQL commands ensures code readability. Here, you need to indent the parentheses and their content so that they clearly show their position in the different layers of the SQL statement. By using this style, you can quickly identify the clauses placed in each pair of parentheses. Additionally, this statement works like the previous one. The only drawback of this style is that you need to use lots of spaces.

Let's analyze another example:

```
CREATE TABLE BOOK_TITLES
```

```
( BOOK_ID      INT,  
  BOOK_TITLE    CHAR (50) ,  
  STOCK_AVAILABILITY INT,  
  CONSTRAINT TS_STOCK_AVAILABILITY CHECK ( (  
    STOCK_AVAILABILITY BETWEEN 1 AND 50 ) OR (   
    STOCK_AVAILABILITY BETWEEN 79 AND 90 ));
```

This code uses BETWEEN (i.e. another SQL operator) to set a range that includes the lowest and highest points. Because it has two ranges, it separates the range specifications using parentheses. The OR keyword connects the range specifications. Basically, OR tells SQL that one of the conditions need to be satisfied. Consequently, the values you enter in the column named STOCK\_AVAILABILITY should be from 1 through 50 or from 79 through 90.

## How to Define an Assertion

Assertions are CHECK constraints that you can apply on multiple tables. Due to this, you can't create assertions while defining a table. Here's the syntax that you must use while creating an assertion:

```
CREATE ASSERTION (name of constraint) CHECK (the search  
conditions)
```

Defining an assertion is similar to defining a table-level CHECK constraint. After typing CHECK, you need to specify the search condition/s.

Let's analyze a new example. Assume that the BOOK\_TITLES table has a column that holds the quantity of books in stock. The total for this table should always be lower than your desired inventory. This example uses an assertion to check whether the total of the STOCK\_AVAILABILITY column is lower than 3000.

```
CREATE ASSERTION LIMIT_STOCK_AVAILABILITY CHECK ( (  
  SELECT SUM (STOCK_AVAILABILITY) FROM BOOK_TITLES ) <  
  3000 );
```

This statement uses a subquery (i.e. “SELECT SUM (STOCK\_AVAILABILITY) FROM BOOK\_TITLES”) and compares it with 3000. The subquery starts with an SQL keyword, SELECT, which queries information from any table. The SQL function called SUM adds up

all of the values inside STOCK\_AVAILABILITY. The keyword FROM, on the other hand, sets the column that holds the table. The system will then compare the subquery's result to 3000. You will get an error message if you add an entry to the STOCK\_AVAILABILITY column that makes the total exceed 3000.

## How to Create a Domain and a Domain Constraint

As mentioned earlier, you may also insert the CHECK constraint into your domain definitions. This kind of constraint is similar to the ones you've seen earlier. The only difference is that you don't attach a domain constraint to a particular table or column. Actually, a domain constraint uses VALUE, another SQL keyword, while referring to a value inside a column specified for that domain. Now, let's discuss the syntax you need to use to generate new domains:

```
CREATE DOMAIN (name of domain) {AS } (type of data)
{ DEFAULT (the default value) }
{ CONSTRAINT (name of constraint) } CHECK <(the search condition)>
```

This syntax has elements you've seen before, as you've seen default clauses and data types in the third chapter. The definition of the constraint, on the other hand, has some similarities with the ones discussed in the last couple of sections.

In the example below, you will generate an INT-type domain. This domain can only accept values between 1 and 50:

```
CREATE DOMAIN BOOK_QUANTITY AS INT CONSTRAINT
TS_BOOK_QUANTITY CHECK (VALUE BETWEEN 1 and 50 );
```

This example involves one new item, which is the VALUE keyword. As mentioned earlier, this keyword refers to a column's specified value using the BOOK\_QUANTITY domain. Consequently, you will get an error message if you will enter a value that doesn't satisfy the assigned condition (i.e. each value must be between 1 and 50).

# CHAPTER 4:

## How to Create an SQL View

*“You can tell whether a man is clever by his answers... You can tell whether a man is wise by his questions.” – Naguib Mahfouz*

Your database stores SQL information using “persistent” (i.e. permanent) tables. However, persistent tables can be impractical if you just want to check particular entries from one or more tables. Because of this, the SQL language allows you to use “views” (also called “viewed tables”).

Views are virtual tables whose definitions act as schema objects. The main difference between views and persistent tables is that the former doesn't store any data. Actually, viewed tables don't really exist – only their definition does. This definition lets you choose specific data from a table or a group of tables, according to the definition's query statements. To invoke a view, you just have to include its name in your query, as if it was an ordinary table.

### How to Add a View to a Database

Views are extremely useful when you're trying to access various kinds of information. If you use a view, you may define complicated queries and save them inside a view definition. Rather than typing queries each time you use them, you may just call the view. In addition, views allow you to present data to other people without showing any unnecessary or confidential information.

For instance, you might need to allow some users to access certain parts of employee records. However, you don't want the said users to access the SSN (i.e. social security number) or pay rates of the listed employees. Here, you may generate views that show only the data needed by users.

### How to Define an SQL View

In SQL, the most basic view that you can create is one which points to a single table and collects information from columns without changing anything. Here is the basic syntax of a view:

```
CREATE VIEW (name of view) { < (name of the view's columns) > }  
AS (the query)
```

{ WITH CHECK OPTION }

Important Note: This part of the book focuses on the first and second lines of the format. You'll learn about the third line later.

You need to set the view's name in the first part of the definition. Additionally, you should name the view's columns if you are facing any of the following circumstances:

If you need to perform an operation to get the column's values, instead of just copying them from a table.

If you are working with duplicate column names. This situation happens when you combine tables.

You may set names for your columns even if you don't need to. For instance, you may assign logical names to your columns so that even an inexperienced user can understand them.

The second part of the format has a mandatory keyword (i.e. AS) and a placeholder for the query. Despite its apparent simplicity, the query placeholder may involve a complicated structure of SQL statements that perform different operations.

Let's analyze a basic example:

```
CREATE VIEW BOOKS_IN_STOCK  
( BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY ) AS  
SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY23
```

This sample is one of the simplest views that you can create. It gets three columns from a table. Remember that SQL isn't strict when it comes to line breaks and spaces. For instance, while creating a view, you may list the column names (if applicable) on a separate line. Database management systems won't care which coding technique you use. However, you can ensure the readability of your codes by adopting a coding style.

Now, let's dissect the sample code given above. The first part sets BOOKS\_IN\_STOCK as the view's name. The second part sets the name of the columns and contains the SQL keyword AS.

If you don't specify the names you want to use, the view's columns will just copy the names of the table's columns. The last two lines hold the search expression, which is a SELECT statement. Here it is:

```
SELECT BOOK_TITLE, AUTHOR, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY
```

SELECT is flexible and extensive: it allows you to write complex queries that give you the exact kind of information you need.

The SELECT statement of this example is basic. It only has two clauses: SELECT and FROM. The first clause sets the column to be returned. The second clause, however, sets the table where the information will be pulled from. Once you call the BOOKS\_IN\_STOCKS view, you will actually call the embedded SELECT command of the view. This action gets the information from the correct table/s.

For the second example, let's create a view that has an extra clause:

```
CREATE VIEW BOOKS_IN_STOCK_80s  
( BOOK_TITLE, YEAR_PUBLISHED, STOCK_AVAILABILITY ) AS  
SELECT BOOK_TITLE, YEAR_PUBLISHED, STOCK_AVAILABILITY  
FROM BOOK_INVENTORY  
WHERE YEAR_PUBLISHED >1979 AND YEAR_PUBLISHED < 1990;
```

The last clause sets a criterion that should be satisfied for the system to retrieve data. The only difference is that, rather than pulling the authors' information, it filters search results based on the year each book was published.

Important Note: The contents of the last clause don't affect the source table in any way. They work only on the information returned by the view.

You may use WHERE in your SELECT statements to set different types of criteria. For instance, you can use this clause to combine tables. Check the following code:

```
CREATE VIEW BOOK_PUBLISHERS  
(BOOK_TITLE, PUBLISHER_NAME ) AS  
SELECT      BOOK_INVENTORY      .BOOK_TITLE,      TAGS  
.PUBLISHER_NAME
```

```
FROM BOOK_INVENTORY, TAGS  
WHERE BOOK_INVENTORY .TAG_ID = TAGS .TAG_ID;24
```

This code creates a view named BOOK\_PUBLISHERS. The BOOK\_PUBLISHERS view contains two columns: BOOK\_TITLE and PUBLISHER\_NAME. With this view, you'll get data from two different sources: (1) the BOOK\_TITLE column of the BOOK\_INVENTORY table and (2) the PUBLISHER\_NAME column of the TABS table.

For now, let's focus on the third clause (i.e. the SELECT statement). This clause qualifies the columns based on the name of their respective tables (e.g. BOOK\_INVENTORY .BOOK\_TITLE). If you are joining tables, you need to specify the name of each table to avoid confusion. Obviously, columns can be highly confusing if they have duplicate names. However, if you're dealing with simple column names, you may omit the name of your tables. For instance, your SELECT clause might look like this:

```
SELECT BOOK_TITLE, PUBLISHER_NAME
```

Now, let's discuss the statement's FROM section. When combining tables, you need to name all of the tables you want to use and separate the entries using commas. Aside from the concern regarding duplicate names, this clause is identical to that of previous examples.

WHERE, the last clause of this statement, matches data rows together. This clause is important since, if you don't use it, you won't be able to match values you've gathered from different tables. In the current example, the values inside the TAG\_ID column of BOOK\_INVENTORY should match the values inside the TAG\_ID column of the table named TAGS.

SQL allows you to qualify a query by expanding the latter's WHERE clause. In the next example, WHERE restricts the returned rows to those that hold "999" in the BOOK\_INVENTORY table's TAG\_ID column:

```
CREATE VIEW BOOK_PUBLISHERS  
(BOOK_TITLE, BOOK_PUBLISHER ) AS  
SELECT      BOOK_INVENTORY      .BOOK_TITLE,      TAGS  
.BOOK_PUBLISHER  
FROM BOOK_INVENTORY, TAGS  
WHERE BOOK_INVENTORY .TAG_ID = TAGS .TAG_ID
```

AND BOOK\_INVENTORY .TAG\_ID = 999;<sup>25</sup>

Let's work on another example. Similar to the examples you've seen earlier, this view collects information from a single table. This view, however, performs computations that return the modified information. Here is the statement:

```
CREATE VIEW BOOK_DISCOUNTS  
(BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE ) AS  
SELECT BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE * 0.8  
FROM BOOK_INVENTORY;
```

This statement creates a view that has three columns: BOOK\_TITLE, ORIGINAL\_PRICE, and REDUCED\_PRICE. Here, SELECT indicates the columns that hold the needed information. The statement defines BOOK\_TITLE and ORIGINAL\_PRICE using the methods discussed in the previous examples. The system will copy the data inside the BOOK\_INVENTORY table's BOOK\_TITLE and ORIGINAL\_PRICE columns. Then, the system will paste the data to the columns of the same name inside the BOOK\_DISCOUNTS view.

However, the last column is different,. Aside from taking values from its corresponding column, it multiplies the collected values by 0.8 (i.e. 80%). This way, the system will determine the correct values to display in the view's REDUCED\_PRICE column.

SQL also allows you to insert the WHERE clause to your SELECT statements. Here's an example:

```
CREATE VIEW BOOK_DISCOUNTS  
( BOOK_TITLE, ORIGINAL_PRICE, REDUCED_PRICE ) AS  
SELECT BOOK_TITLE, ORIGINAL_PRICE, REDUCED PRICE * 0.8  
FROM BOOK_INVENTORY  
WHERE STOCK_AVAILABILITY > 20;26
```

This WHERE clause limits the search to entries whose STOCK\_AVAILABILITY value is higher than 20. As this example shows, you may perform comparisons on columns that are included in the view.

## How to Create an Updateable View

In the SQL language, some views allow you to perform updates. Simply put, you may use a view to alter the information (i.e. add new rows and/or alter existing information) inside the table you're working on. The “updateability” of a view depends on its SELECT statement. Usually, views that involve simple SELECT statements have higher chances of becoming updateable.

Remember that SQL doesn't have syntax to create updateable views. Rather, you need to write a SELECT statement that adheres to certain standards. This is the only way for you to create an updateable view.

The examples you've seen in this chapter imply that the SELECT statement serves as the search expression of a CREATE VIEW command. To be precise, query expressions may belong to different kinds of expressions. As an SQL user, most of the time, you'll be dealing with query specifications. Query expressions are SQL expressions that start with SELECT and contain different elements. To keep it simple, let's assume that SELECT is a query specification. Database products also use this assumption, so it is certainly effective.

You can't summarize, combine, or automatically delete the information inside the view.

The table you're working with should have at least one updateable column.

Every column inside the view should point to a single column in a table.

Every row inside the view should point to a single row in a table.

## How to Drop a View

In some cases, you need to delete a view from a database. To do that, you need to use the following syntax:

DROP VIEW (name of the view);

The system will delete the view as soon as you run this statement. However, the process won't affect the underlying information (i.e. the data stored inside the actual tables). After dropping a view, you may recreate it or use its name to generate another view. Let's analyze a basic example:

DROP VIEW BOOK\_PUBLISHERS;

This command will delete the BOOK\_PUBLISHERS view from the database. However, the underlying information will be unaffected.

## **Database Security**

Security is an important element of every database. You need to make sure that your database is safe from unauthorized users who may view or alter data. Meanwhile, you also need to ensure that authorized users can access and/or change data without any problems. The best solution for this problem is to provide each user with the privileges they need to do their job.

To protect databases, SQL has a security scheme that lets you specify which database users can view specific information from. This scheme also allows you to set what actions each user can perform. This security scheme (or model) relies on authorization identifiers. As you've learned in the second chapter, authorization identifiers are objects that represent one or more users that can access/modify the information inside the database.

## **The Security Model of SQL**

The security of your database relies on authorization identifiers. You can use these identifiers to allow other people to access and/or alter your database entries. If an authorization identifier lacks the right privileges to alter a certain object, the user won't be able to change the information inside that object. Additionally, you may configure each identifier with various kinds of privileges.

In SQL, an authorization identifier can be a user identifier (i.e. "user") or a role name (i.e. "role"). A "user" is a security profile that may represent a program, a person, or a service. SQL doesn't have specific rules regarding the creation of a user. You may tie the identifier to the OS (i.e. operating system) where the database system runs. Alternatively, you may create user identifiers inside the database system itself.

A role is a group of access rights that you may assign to users or other roles. If a certain role has access to an object, all users you've assigned that role to can access the said object.

SQL users often utilize roles to provide uniform sets of access rights to other authorization identifiers. One of the main benefits offered by a role is that it can exist without any user identifier. That means you can create a role before creating a user. In addition, a role will stay in the database even if

you have deleted all of your user identifiers. This functionality allows you to implement a flexible process to administer access rights.

The SQL language has a special authorization identifier called PUBLIC. This identifier covers all of the database users. Similar to other identifiers, you may assign access rights to a PUBLIC profile.

Important Note: You need to be careful when assigning access rights to the PUBLIC identifier, as users might use that identifier for unauthorized purposes.

## **Creating and Deleting a Role**

Generating new roles is a simple process. The syntax has two clauses: an optional clause and a mandatory clause.

`CREATE ROLE (name of role)`

`{ WITH ADMIN [ CURRENT_ROLE | CURRENT_USER ] }`

As you can see, CREATE ROLE is the only mandatory section of this statement. You don't need to set the statement's WITH ADMIN part. Actually, SQL users rarely set that clause. WITH ADMIN becomes important only if your current role name/user identifier pair doesn't have any null values.

Let's use the syntax to create a role:

`CREATE ROLE READERS;`

That's it. After creating this role, you will be able to grant it to users or other roles.

To drop (or delete) a role, you just have to use the following syntax:

`DROP ROLE (name of role)`

This syntax has a single requirement: the name of the role you want to delete.

## **Granting and Revoking a Privilege**

Whenever you grant a privilege, you are actually linking a privilege to an authorization identifier. You will place this privilege/authorization identifier pair on an object, allowing the former to access the latter based on the defined privileges.

```
GRANT [ (list of privileges) | ALL PRIVILEGES ]  
ON (type of object) (name of object)  
TO [ (list of authorization identifiers) | PUBLIC ] { WITH GRANT  
OPTION }  
{ GRANTED BY [ CURRENT_ROLE | CURRENT_USER ] }
```

This syntax has three mandatory clauses, namely: ON, TO and GRANT. The last two clauses, GRANTED BY and WITH GRANT OPTION, are completely optional.

The process of revoking privileges is simple and easy. You just have to use the syntax given below:

```
REVOKE { GRANT OPTION FOR } [ (list of privileges) | ALL  
PRIVILEGES ]  
ON (type of object) name of object  
FROM [ { list of authorization identifiers) | PUBLIC } ]
```

# CHAPTER 5:

## Database Creation

*“To improve is to change; to be perfect is to change often.” – Sir Winston Churchill*

Data is stored in tables and indexed to make queries more efficient. Before you can create tables, you need a database to hold the table. If you're starting from zero, you will have to learn to create and use a database.

### **Creating a Database**

*“We are what we pretend to be, so we must be careful about what we pretend to be.” – Kurt Vonnegut*

To create a database, you will use the CREATE command with the name of the database.

The following is the syntax:

```
CREATE DATABASE database_name;
```

To demonstrate, assume you wanted to create a database and name it xyzcompany:

```
CREATE DATABASE xyzcompany;
```

With that statement, you have just created the xyzcompany database. Before you can use this database, you need to designate it as the active database. You have to run the USE command with the database name to activate your new database.

Here's the statement:

```
USE xyzcompany;
```

In following sessions, you can just type the statement ‘USE xyzcompany’ to access the database.

### **Removing a Database**

If you need to remove an existing database, you can easily do so with this syntax:

```
DROPDATABASE databasename;
```

Therefore, you must exercise caution when using the DROP command to remove a database. You will also need admin privileges to drop a database.

### **Schema Creation**

The CREATE SCHEMA statement is used to define a schema. On the same statement, you can also create objects and grant privileges on these objects.

The CREATE SCHEMA command can be embedded within an application program. Likewise, it can be issued using dynamic SQL statements. For example, if you have database admin privileges, you can issue this statement which creates a schema called USER1 with the USER1 as its owner:

```
CREATE SCHEMA USER1 AUTHORIZATION USER1
```

The following statement creates a schema with an inventory table. It also grants authority on the inventory table to USER2:

```
CREATE SCHEMA INVENTORY
```

```
CREATE TABLE ITEMS (IDNO INT(6) NOT NULL,
```

```
    SNAME VARCHAR(40),
```

```
    CLASS INTEGER)
```

```
GRANT ALL ON ITEMS TO USER2
```

MySQL 5.7

If you're using MySQL 5.7, CREATE SCHEMA is synonymous to the CREATE DATABASE command.

Here's the syntax:<sup>[27](#)</sup>

```
CREATE {DATABASE | SCHEM A}[IF NOT EXISTS] db_name  
[create_specification]...
```

create\_specification:

```
[DEFAULT] CHARACTER SET [=] charset_name | [DEFAULT] COLLATE  
[=] collation_name
```

## Oracle 11g

In Oracle 11g, you can create several views and tables and perform several grants in one transaction within the CREATE SCHEMA statement. To successfully execute the CREATE SCHEMA command, Oracle runs each statement within the block and commits the transaction if no errors are encountered. If a statement returns an error, all statements are rolled back.

The statements CREATE VIEW, CREATE TABLE, and GRANT may be included within the CREATE SCHEMA statement. Hence, you must not only have the privilege to create a schema, you must also have the privileges needed to issue the statements within it.

The syntax is the following:<sup>[28](#)</sup>

```
{ create_table_statement  
| create_view_statement  
| grant_statement  
}...;
```

## SQL Server 2014

In SQL Server 2014, the CREATE SCHEMA statement is used to create a schema in the current database. This transaction may also create views and tables within the newly-created schema and set GRANT, REVOKE, or DENY permission on these objects.

This statement creates a schema and sets the specifications for each argument:<sup>29</sup>

```
CREATE SCHEM A schem a_name_clause [ <schem a_element> [ ...n ] ]  
<schem a_name_clause> ::=  
{  
    schem a_name  
    | AUTHORIZATION owner_name  
    | scheme a_name AUTHORIZATION owner_name  
}  
<schem a_element> ::=  
{  
    table_definition | view_definition | grant_statement |  
    revoke_statement | deny_statement  
}
```

## PostgreSQL 9.3.13

The CREATE SCHEMA statement is used to enter a new schema into a database. The schema name should be unique within the current database.<sup>30</sup>

Here's the syntax:

```
CREATE SCHEM A schem a_name [AUTHORIZATION user_name][schem a_element[...]]
```

```
CREATE SCHEM A AUTHORIZATION user_name [schem a_element[...]]
```

```
CREATE SCHEM A IF NOT EXISTS schem a_name [AUTHORIZATION user_name]
```

```
CREATE SCHEM A IF NOT EXISTS AUTHORIZATION user_name
```

## Creating Tables and Inserting Data Into Tables

Tables are the main storage of information in databases. Creating a table means specifying a name for a table, defining its columns, as well as the data type of each column.

### How to Create a Table

The keyword CREATE TABLE is used to create a new table. It is followed by a unique identifier and a list that defines each column and the type of data it will hold.

```
CREATE TABLE table_name  
(  
    colum n1 datatype [NULL | NOT NULL].  
    colum n2 datatype [NULL | NOT NULL].  
    ...  
);
```

This is the basic syntax to create a table:

#### Parameters

table\_name: This is the identifier for the table.

column1, column2: These are the columns that you want the table to have. All columns should have a data type. A column is defined as either NULL or NOT NULL. If this is not specified, the database will assume that it is NULL.

The following set of questions can serve as a guide when creating a new table:

- What is the most appropriate name for this table?
- What data types will I be working with?
- What is the most appropriate name for each column?
- Which column(s) should be used as the main key(s)?
- What type of data can be assigned to each column?
- What is the maximum width for each column?
- Which columns can be empty and which columns should not be empty?

The following example creates a new table with the xyzcompany database. It will be named EMPLOYEES:

```
CREATE TABLE EMPLOYEES(
    ID INT(6) auto_increment, NOT NULL,
    FIRST_NAME VARCHAR(35) NOT NULL,
    LAST_NAME VARCHAR(35) NOT NULL,
    POSITION VARCHAR(35),
    SALARY DECIMAL(9,2),
    ADDRESS VARCHAR(50),
    PRIMARY KEY (id)
);
```

The code creates a table with 6 columns. The ID field was specified as its primary key. The first column is an INT data type with a precision of 6. It does not accept a NULL value. The second column, FIRST\_NAME, is a VARCHAR type with a maximum range of 35 characters. The third column, LAST\_NAME, is another VARCHAR type with a maximum of 35 characters. The fourth column, POSITION, is a VARCHAR type which is set at a maximum of 35 characters. The fifth column, SALARY, is a DECIMAL type with a precision of 9 and scale of 2. Finally, the fifth column, ADDRESS, is a VARCHAR type with a maximum of 50 characters. The id column was designated as the primary key.

## **Creating a New Table Based on Existing Tables**

You can create a new table based on an existing table by using the CREATE TABLE keyword with SELECT.

Here's the syntax.<sup>[31](#)</sup>

```
CREATE TABLE new_table_name AS
(
    SELECT [column n1.column, column n, 2'''column nN]
    FROM existing_table_name
    [WHERE]
);
```

Executing this code will create a new table with column definitions that are identical to the original table. You may copy all columns or select specific

columns for the new table. The new table will be populated by the values of the original table.

To demonstrate, create a duplicate table named STOCKHOLDERS from the existing EMPLOYEES table within the xyzcomppany. Here's the code:

```
CREATE TABLE STOCKHOLDERS AS  
SELECT ID, FIRST_NAME, LAST_NAME, POSITION, SALARY, ADDRESS  
FROM EMPLOYEES;
```

## Inserting Data into Table

SQL's Data Manipulation Language (DML) is used to perform changes to databases. You can use DML clauses to fill a table with fresh data and update an existing table.

## Populating a Table with New Data

There are two ways to fill a table with new information: manual entry or automated entry through a computer program.

Populating data manually involves data entry using a keyboard, while automated entry involves loading data from an external source. It may also include transferring data from one database to a target database.

Unlike SQL keywords or clauses that are case-insensitive, data is case-sensitive. Hence, you have to ensure consistency when using or referring to data. For instance, if you store an employee's first name as 'Martin', you should always refer to it in the future as 'Martin' and never 'MARTIN' or 'martin'.

### The INSERT Keyword

The INSERT keyword is used to add records to a table. It inserts new rows of data to an existing table.

There are two ways to add data with the INSERT keyword. In the first format, you simply provide the values for each field and they will be assigned sequentially to the table's columns. This form is generally used if you need to add data to all columns.

You will use the following syntax for the first form:

```
INSERT INTO table_name  
VALUES ('value1', 'value2', [NULL];
```

In the second form, you'll have to include the column names. The values will be assigned based on the order of the columns' appearance. This form is typically used if you want to add records to specific columns.

Here's the syntax:

```
INSERT INTO table_name (column n1, column n2, column n3)  
VALUES ('value1', 'value2', 'value3');
```

Notice that in both forms, a comma is used to separate the columns and the values. In addition, you have to enclose character/string and date/time data within quotation marks.

Assuming that you have the following record for an employee:

First Name Robert

Last Name Page

Position Clerk

Salary 5,000.00

282 Patterson Avenue Illinois

To insert this data into the EMPLOYEES table, you can use the following statement:

```
INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, POSITION,  
SALARY, ADDRESS)  
VALUES ('Robert', 'Page', 'Clerk', 5000.00, '282 Patterson Avenue, Illinois');
```

To view the updated table, here's the syntax:

```
SELECT * FROM table_name;
```

To display the data stored in the EMPLOYEES' table, use the following statement:

```
SELECT * FROM EMPLOYEES;
```

The wildcard (\*) character tells the database system to select all fields on the table.

Here's a screenshot of the result:<sup>32</sup>

```
mysql> SELECT * FROM EMPLOYEES;
+----+-----+-----+-----+-----+-----+
| ID | FIRST_NAME | LAST_NAME | POSITION | SALARY | ADDRESS |
+----+-----+-----+-----+-----+-----+
| 1  | Robert     | Page      | Clerk    | 5000.00 | 282 Patterson Avenue, Illinois |
| 015 |             |            |          |          |          |
+----+-----+-----+-----+-----+-----+
```

Now, try to encode the following data for another set of employees:

| First Name | Last Name | Position   | Salary   | Address                     |
|------------|-----------|------------|----------|-----------------------------|
| John       | Malley    | Supervisor | 7,000.00 | 5 Lake View, New York       |
| Kristen    | Johnston  | Clerk      | 4,500.00 | 25 Jump Road, Florida       |
| Jack       | Burns     | Agent      | 5,000.00 | 5 Green Meadows, California |

You will have to repeatedly use the INSERT INTO keyword to enter each employee data to the database. Here's how the statements would look:

```
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION,  
SALARY, ADDRESS)  
VALUES('John', 'Malley', 'Supervisor', 7000.00, '5 Lake View New York');  
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION,  
SALARY, ADDRESS)  
VALUES('Kristen', 'Johnston', 'Clerk', 4000.00, '25 Jump Road, Florida');  
INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, POSITION,  
SALARY, ADDRESS)  
VALUES('Jack', 'Burns', 'Agent', 5000.00, '5 Green Meadows, California');
```

To fetch the updated EMPLOYEES table, use the SELECT command with the wild card character.

```
SELECT * FROM EMPLOYEES;
```

Here's a screenshot of the result:

| mysql> SELECT * FROM EMPLOYEES; |            |           |            |         |                                |
|---------------------------------|------------|-----------|------------|---------|--------------------------------|
| ID                              | FIRST_NAME | LAST_NAME | POSITION   | SALARY  | ADDRESS                        |
| 1                               | Robert     | Page      | Clerk      | 5000.00 | 282 Patterson Avenue, Illinois |
| 2                               | John       | Malley    | Supervisor | 7000.00 | 3 Lake View New York           |
| 3                               | Kristen    | Johnston  | Clerk      | 4000.00 | 25 Lump Road Florida           |
| 4                               | Jack       | Burns     | Agent      | 6000.00 | 5 Green Meadows California     |

4 rows in set (0.00 sec)

Notice that SQL assigned an ID number for each set of data you entered. This is because you have defined the ID column with the auto\_increment attribute. This property will prevent you from using the first form when inserting data. So, you have to specify the rest of the columns in the INSERT INTO table\_name line.<sup>33</sup>

## Inserting Data into Specific Columns

You may also insert data into specific column(s). You can do this by specifying the column name inside the column list and the corresponding values inside the VALUES list of the INSERT INTO statement. For example, if you just want to enter an employee's full name and position, you will need to specify the column names FIRST\_NAME, LAST\_NAME, and SALARY in the columns list and the values for the first name, last name, and salary inside the VALUES list.

To see how this works, try entering the following data into the EMPLOYEES table:

|            |           |          |          |         |
|------------|-----------|----------|----------|---------|
| First Name | Last Name | Position | Salary   | Address |
| James      | Hunt      |          | 7,500.00 |         |

Here's a screenshot of the updated EMPLOYEES table:<sup>34</sup>

| ID | FIRST_NAME | LAST_NAME | POSITION   | SALARY  | ADDRESS                        |
|----|------------|-----------|------------|---------|--------------------------------|
| 1  | Robert     | Page      | Clerk      | 5000.00 | 282 Patterson Avenue, Illinois |
| 2  | John       | Malley    | Supervisor | 7000.00 | 5 Lake View New York           |
| 3  | Kristen    | Johnston  | Clerk      | 4000.00 | 25 Lump Road Florida           |
| 4  | Jack       | Burns     | Agent      | 5000.00 | 5 Green Meadows California     |
| 5  | James      | Hunt      | NULL       | 7500.00 | NULL                           |

5 rows in set (0.00 sec)

## Inserting NULL Values

In some instances, you may have to enter NULL values into a column. For example, you may not have data on hand to enter a new employee's salary. It may be misleading to provide just about any salary figure.

Here's the syntax:

```
INSERT INTO schem a.table_name
VALUES ('column n1', NULL, 'column n3');
```

You'll need this answer sheet to be able to check your syntax in each exercise to ensure that it is correct. You are more than welcome to use it if you're stuck on an exercise as well. At the end of each exercise, I encourage you to check your answers.

Each exercise gives an overview, as well as examples, before you start applying what you learned in each section.

Below is a sample of the Product table that you will be using in the next few exercises.

| Product ID | Name        | Product Number | Color | Standard Cost | List Price | Size |
|------------|-------------|----------------|-------|---------------|------------|------|
| 317        | LL Crankarm | CA-5965        | Black | 0             | 0          | NULL |

|     |                 |         |        |   |   |      |
|-----|-----------------|---------|--------|---|---|------|
| 318 | ML Crankarm     | CA-6738 | Black  | 0 | 0 | NULL |
| 319 | HL Crankarm     | CA-7457 | Black  | 0 | 0 | NULL |
| 320 | Chainring Bolts | CB-2903 | Silver | 0 | 0 | NULL |
| 321 | Chainring Nut   | CN-6137 | Silver | 0 | 0 | NULL |

## Query Structure and SELECT Statement

Understanding the syntax and its structure will be extremely helpful for you in the future. Let's delve into the basics of one of the most common statements, the SELECT statement, which is used solely to retrieve data.

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
```

In the above query, you're selecting two columns and all rows from the entire table. Since you're selecting two columns, the query will perform much faster than if you had selected all of the columns like this:

```
SELECT *
FROM Table_Name
```

Select all of the columns from the Production.Product table.

Using the same table, select only the ProductID, Name, Product Number, Color and Safety Stock Level columns.

*(Hint: The column names do not contain spaces in the table!)*

## The WHERE Clause

The WHERE clause is used to filter the amount of rows returned by the query. This clause works by using a condition, such as if a column is equal to, greater than, less than, a certain value.

When writing your syntax, it's important to remember that the WHERE condition comes after the FROM statement. The types of operators that can be used vary based on the type of data that you're filtering.

EQUALS – This is used to find an exact match. The syntax below uses the WHERE clause for a column that contains the exact string of 'Value'. Note that strings need single quotes around them, but numerical values do not.

```
SELECT Column_Name1, Column_Name2
FROM Table_Name
```

WHERE Column\_Name3 = 'Value'

BETWEEN – Typically used to find values between a certain range of numbers or date ranges. It's important to note that the first value in the BETWEEN comparison operator should be lower than the value on the right. Note the example below comparing between 10 and 100.

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 BETWEEN 100 AND 1000
```

GREATER THAN, LESS THAN, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO – SQL Server has comparison operators that you can use to compare certain values.

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 = < 1000 --Note that you can also use <, >, >= or even  
<> for not equal to
```

LIKE – This searches for a value or string that is contained within the column. You would still use single quotes, but include the percent symbols indicating if the string is in the beginning, at the end or anywhere between ranges of strings.

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 LIKE '%Value%' --Searches for the word 'value' in any field
```

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name  
WHERE Column_Name3 LIKE 'Value%' --Searches for the word 'value' at the beginning of the field
```

```
SELECT Column_Name1, Column_Name2  
FROM Table_Name35  
WHERE Column_Name3 LIKE '%Value' --Searches for the word 'value' at the end of the field
```

IS NULL and IS NOT NULL – As previously discussed, NULL is an empty cell that contains no data. Eventually, you'll work with a table that does contain NULL values, which you can handle in a few ways.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 IS NULL --Filters any value that is NULL in that column, but don't put NULL in single quotes since it's not considered a string.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 IS NOT NULL --Filters any value that is not NULL in that column.

Using the Production.Product table again, select the Product ID, Name, Product Number and Color. Filter the products that are silver colored.

### ***Using ORDER BY***

The ORDER BY clause is simply used to sort data in ascending or descending order, and is specified by which column you want to sort. This command also sorts in ascending order by default, so if you want to sort in descending order, use DESC in your command.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = 'Value'
```

ORDER BY Column\_Name1, Column\_Name2 --Sorts in ascending order, but you can also include ASC to sort in ascending order.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = 'Value'
```

ORDER BY Column\_Name1, Column\_Name2 DESC --This sorts the data in descending order by specifying DESC after the column list.

Select the Product ID, Name, Product Number and List Price from the Production.Product table where the List Price is between \$100.00 and \$400.00. Then, sort the results by the list price from smallest to largest.

(*Hint: You won't use the \$ in your query and also, you may refer back to the sorting options you just reviewed if you need to.*)<sup>36</sup>

### **Data Definition Language (DDL)**

*"There is nothing worse than aggressive stupidity." – Johann Wolfgang von Goethe*

DDL is the SQL syntax that is used to create, alter or remove objects within the instance or database itself. Below are some examples to help you get started. DDL is split up into three types of commands:

**CREATE** - This will create objects within the database or instance, such as other databases, tables, views, etc.

**--Creates a Database**

**CREATE DATABASE** DatabaseName

**--Creates a schema (or container) for tables in the current database**

**CREATE SCHEMA** SchemaName

**--Creates a table within the specified schema**

**CREATE TABLE** SchemaName.TableName

(

Column1 datatype PRIMARY KEY,

Column2 datatype(n),

Column3 datatype

)<sup>37</sup>

**--Creates a View**

**CREATE VIEW** ViewName

AS

SELECT

Column1,

Column2

FROM TableName

**ALTER** - This command will allow you to alter existing objects, like adding an additional column to a table or changing the name of the database, for example.

**--Alters the name of the database**

**ALTER DATABASE** DatabaseName **MODIFY NAME** = NewDatabaseName

**--Alters a table by adding a column**

**ALTER TABLE** TableName

**ADD** ColumnName datatype(n)<sup>38</sup>

**DROP** - This command will allow you to drop objects within the database or the database itself. This can be used to drop tables, triggers, views, stored procedures, etc. Please note that these items will not exist within your database anymore – or the database will cease to exist if you drop it.

**--Drops a Database - use the master db first**

```
USE master
```

```
GO
```

**--Then drop the desired database**

```
DROP DATABASE DatabaseName
```

```
GO
```

**--Drops a table; should be performed in the database where the specified table exists**

```
DROP TABLE Table_Name
```

**--Drops a view from the current database**

```
DROP VIEW ViewName
```

### ***Applying DDL Statements***

Open a new query window on your own machine by clicking the “New Query” button and ensure that you’re using the ‘master’ database.

You can tell which database you’re using and switch between databases by checking below and using the drop-down menu.



Back in the [Guidelines for Writing T-SQL Syntax](#) section, you learned about the ‘USE’ statement and splitting up batches of code with ‘GO’.

You’ll be using this in the next exercise. So, feel free to refer back to that section before or during the exercise.

Create a database called Company. Don’t forget to use the GO statement between using the master database and creating your new database!

After you’ve performed this, you’ll now have a database that you can begin adding tables to. Below are a series of exercises in which you’ll begin creating your first schema and table within this database.

Create a schema called Dept. The table that you’ll be creating in the next exercise will be associated to this schema.

Create a table called Department. The table should have two columns: Department\_Id, which should be an integer data type and the primary key. The other column should be Department\_Name, which should be a data type of VARCHAR of 30 bytes.

Alter the name of your database from Company to Company\_Db.

## **Running the DDL Script**

Now, it's time for you to download the following DDL script to create the rest of the tables that you'll be working with. Don't worry, I've written the script so that it will successfully run by clicking the 'Execute' button. Just make sure that you have completed the exercises correctly, otherwise it will not work.

Once you've performed this, you'll start using your knowledge of DML to add and retrieve data.

## **Data Manipulation Language (DML)**

DML or Data Manipulation Language is exactly what it sounds like; it manipulates the data. The statements that are part of DML allow you to modify existing data, create new data, or even delete data.

Take note that DML is not specific to objects like modifying a table's structure or a database's settings, but it works with the data within such objects.

**SELECT** - This statement is the most common SQL statement, and it allows you to retrieve data. It is also used in many reporting scenarios.

### **--Selects two columns from a table**

```
SELECT
```

```
Column_Name1,
```

```
Column_Name2
```

```
FROM Table_Name
```

**INSERT** - This statement allows you to insert data into the tables of a particular database.

### **--Inserts three rows into the table**

```
INSERT INTO Table_Name
```

```
(Column_Name1, Column_Name2) --These two items are the columns
```

```
VALUES
```

**('Value1', 1), --Here, I'm inserting Value1 into Column\_Name1 and the number 1 in Column\_Name2.**

```
('Value2', 2),
```

```
('Value3', 3)
```

Now, it is time to insert data into your Department table. Insert the values as shown below. For example, the Department Name of Marketing must have a Department ID of 1, and so on.

Department\_Id: 1, Department\_Name: Marketing  
Department\_Id: 2, Department\_Name: Development  
Department\_Id: 3, Department\_Name: Sales  
Department\_Id: 4, Department\_Name: Human Resources  
Department\_Id: 5, Department\_Name: Customer Support  
Department\_Id: 6, Department\_Name: Project Management  
Department\_Id: 7, Department\_Name: Information Technology  
Department\_Id: 8, Department\_Name: Finance/Payroll  
Department\_Id: 9, Department\_Name: Research and Development

UPDATE - Allows you to update existing data. For instance, you may change an existing piece of data in a table to another value.

It's recommended to use caution when updating values in a table. You could give all rows within one column the same value if you don't specify a condition in the WHERE clause.

**--Updating a value or set of values in one column of a table**

UPDATE Table\_Name

SET

**Column\_Name1 = 'New Value 1', --Specify what your new value should be here**

Column\_Name2 = 'New Value 2'

WHERE

**Column\_Name1 = 'Old Value' --Using the WHERE clause as a condition**

**--Updates all rows in the table with the same value for one specific column**

UPDATE Table\_Name

SET Column\_Name1 = 'Value'<sup>39</sup>

Now, you will update one of the department names in your table. Let's say that you feel like the name 'Finance/Payroll' won't work because payroll is handled by employees via an online web portal. So, that leaves finance, but it's best to call this department 'Accounting' instead, due to the responsibilities of those in the department.

Update the Department table and change the value from 'Finance/Payroll' to 'Accounting' based on its ID.

**DELETE** - This action is self-explanatory; it deletes data from a specified object, like a table. It's best to use caution when running the DELETE statement.

If you don't use the WHERE clause in your query, you'll end up deleting all of the data within the table. So, it's always best to use a WHERE clause in a DELETE statement.

\*Note: Much like the UPDATE statement, you can also swap out DELETE with SELECT in your statement to see what data you will be deleting, as long as you're using the WHERE clause in your query. Like the UPDATE statement, when you delete one value, it's best to use the primary key as a condition in your WHERE clause!

**--Uses a condition in the DELETE statement to only delete a certain value or set of values**

```
DELETE FROM Table_Name  
WHERE Column_Name1 = 'Some Value'
```

**--Deletes all of the data within a table since the WHERE clause isn't used**

```
DELETE FROM Table_Name
```

Let's say that this company doesn't actually have a Research and Development (R&D) department. Perhaps they haven't grown enough yet or don't require that department.

For this exercise, delete the 'Research and Development' department from the Department table based on its ID.

## Running the DML Script

Now, it's time for you to download the following DML script in order to populate the Company\_Db database with data. Again, make sure that you have completed the exercises correctly, otherwise it will not work.

This data will be used in the following exercises in the 'Transforming Data' section. So, please keep in mind that each exercise refers to the Company\_Db database, unless specified otherwise.

# CHAPTER 6:

## Database Administration

*“Laws too gentle are seldom obeyed; too severe, seldom executed.” – Benjamin Franklin*

In order to start on the path of a Database Administrator, Database Developer or even a Data Analyst, you’ll need to know how to back up, restore and administer databases. These are essential components to maintaining a database and are definitely important responsibilities.

### ***Recovery Models***

There are several different recovery models that can be used for each database in the SQL Server. A recovery model is an option or model for the database that determines the type of information that can be backed up and restored.

Depending on your situation, like if you cannot afford to lose critical data or you want to mirror a Production environment, you can set the recovery model to what you need. Also, keep in mind that the recovery model that you choose could also affect the time it takes to back up or restore the database and logs, depending on their size.

Below are the recovery models and a brief explanation of each. The SQL statement example at the end of each is what you would use in order to set the recovery model.

#### **Full**

This model covers all of the data within the database, as well as the transaction log history.

When using this model and performing a restore of the database, it offers “point in time” recovery. This means that if there was a point where data was lost from the database during a restore for example, it allows you to roll back and recover that data.

This is the desired recovery model if you cannot afford to lose any data.

It may take more time to back up and restore, depending on the size of the data.

- Can perform full, differential and transaction log backups.

`ALTER DATABASE DatabaseName SET RECOVERY FULL`

### Simple

This model covers all of the data within the database too, but recycles the transaction log. When the database is restored, the transaction log will be empty and will log new activity moving forward.

Unlike the “Full” recovery model, the transaction log will be reused for new transactions and therefore cannot be rolled back to recover data if it has accidentally been lost or deleted.

It’s a great option if you have a blank server and need to restore a database fairly quickly and easily, or to just mirror another environment and use it as a test instance.

- Can perform full and differential backups only, since the transaction log is recycled per this recovery model.

`ALTER DATABASE DatabaseName SET RECOVERY SIMPLE`

### Bulk Logged

This particular model works by forgoing the bulk operations in SQL Server, like BULK INSERT, SELECT INTO, etc. and does not store these in the transaction logs. This will help free up your transaction logs and make it easier and quicker during the backup and restore process.

Using this method, you have the ability to use “point in time” recovery, as it works much like the “Full” recovery model.

If your data is critical to you and your database processes often use bulk statements, this recovery model is ideal. However, if your database does not often use bulk statements, the “Full” recovery model is recommended.

- Can perform full, differential and transaction log backups.

`ALTER DATABASE DatabaseName SET RECOVERY BULK_LOGGED`

Let’s say you’re in the middle of a large database project and need to back up the database. Use SQL syntax (and the examples above) to set the

recovery model for Company\_Db to “Full” to prepare for the backup process.

## **Database Backup Methods**

There are a few main backup methods that can be used to back up databases in SQL Server, and each one is dependent on the recovery model being used. In the previous section regarding recovery models, the last bullet point in each model discussed the types of backups that can be performed, i.e. full, differential and transaction log.

*Note: if you don't remember the types of backups that can be performed for each recovery model, just give yourself time! Eventually, you will remember them!*

Each backup method will perform the following action:

Full:

Backs up the database in its entirety, as well as the transaction log (if applicable.)

Ideal if the same database needs to be added to a new server for testing.

It may take a longer period of time to back up if the database is large in size, in addition to it backing up the entire database and transaction log.

- However, this doesn't need to be performed nearly as often as a differential since you're backing up the entire database.

```
BACKUP      DATABASE      DatabaseName      TO      DISK      =
'C:\SQLBackups\DatabaseName.BAK'
```

Differential:

It is dependent upon a full back up, so the full backup must be performed first. However, this will back up any data changes between the last full back up and the time that the differential backup takes place.

It is ideal in any situation, but must be done more frequently since there is new data being added to the database frequently.

It is much faster to back up than a full backup and takes up less storage space.

- However, this doesn't capture the database entirely, so this is the reason it must be performed periodically.

```
BACKUP      DATABASE      DatabaseName      TO      DISK      =
'C:\SQLBackups\DatabaseName.BAK' WITH DIFFERENTIAL
```

### **Transaction Log:**

Like a differential backup, it's dependent upon a full backup.

Backs up all of the activity that has happened in the database.

Useful and necessary in some cases, like when restoring a database in full.

Since they hold the activity within the database's history, it's wise to back this up frequently so that the logs do not grow large in size.

Some logs are initially larger in size, but can become smaller as more frequent backups are done (depends on the activity within the database).

- Therefore, the log could be quick to back up and restore, if the database has minimal activity and is backed up frequently.

```
BACKUP      LOG      DatabaseName      TO      DISK      =
'C:\SQLBackups\DatabaseName.TRN'
```

In the last exercise, you set the recovery model of the Company\_Db to "Full". Now for this exercise, you'll be backing up the database.

First, back up the database in full using the full backup method.

Next, delete all of the data from the Sales.Product\_Sales table (intended to be a simulation of losing data.)

Finally, query the table that you just deleted data from to ensure that the data doesn't exist. Later on, you'll be restoring the database and ensuring that the data is the Sales.Product\_Sales table.

## ***Database Restores***

In the database world, you will probably hear that the most important part of restoring a database is using a solid backup. That's entirely true! This is why it's important to ensure that you're using the proper recovery model.

### ***Preparing to Restore the Database***

An important thing to note is that the .bak file typically contains file groups or files within them. Each file could either be a full or differential backup. It's important to know which one you're restoring, but you can also specify which file(s) you'd like to restore. However, if you don't specify, SQL Server will pick the default file, which is 1. This is a typical full back up.

First, use RESTORE HEADERONLY to identify the files within your .BAK file. This is something you should do prior to restoring a database.

#### --View database backup files

```
RESTORE      HEADERONLY      FROM      DISK      =
'C:\SQLBackups\DatabaseName.BAK'
```

Once you run the above query, look for the number in the “BackupType” column. The 1 indicates a full backup, whereas a 5 indicates a differential backup.

Also, the file number that you need to use is in the “Position” column. In the example below, note that the file number 8 is a full backup and file number 9 is a differential backup.

Results Messages

|   | BackupName | BackupDescription | BackupType | ExpirationDate | Compressed | Position |
|---|------------|-------------------|------------|----------------|------------|----------|
| 1 | NULL       | NULL              | 1          | NULL           | 0          | 1        |
| 2 | NULL       | NULL              | 1          | NULL           | 0          | 2        |
| 3 | NULL       | NULL              | 1          | NULL           | 0          | 3        |
| 4 | NULL       | NULL              | 1          | NULL           | 0          | 4        |
| 5 | NULL       | NULL              | 1          | NULL           | 0          | 5        |
| 6 | NULL       | NULL              | 1          | NULL           | 0          | 6        |
| 7 | NULL       | NULL              | 5          | NULL           | 0          | 7        |
| 8 | NULL       | NULL              | 1          | NULL           | 0          | 8        |
| 9 | NULL       | NULL              | 5          | NULL           | 0          | 9        |

Additionally, you must first use the ‘master’ database and then set the desired database to a non-active state (SINGLE\_USER) to only allow one open connection to the database prior to restoring.

In the below syntax, the WITH ROLLBACK IMMEDIATE means that any incomplete transactions will be rolled back (not completed) in order to set the database to a single, open connection state.

**--Sets the database to allow only one open connection**

**--All other connections to the database must be closed, otherwise the restore fails**

```
ALTER DATABASE DatabaseName SET SINGLE_USER WITH
ROLLBACK IMMEDIATE
```

Once the restore has completed, you can set the database back to an active state (MULTI\_USER), as shown below. This state allows multiple users to connect to the database, rather than just one.

**--Sets the database to allow multiple connections**

```
ALTER DATABASE DatabaseName SET MULTI_USER
```

***Database Restore Types***

Below is an overview of the types of restores that you can perform depending on your situation/needs.

**Full Restore**

Restores the entire database, including its files.

Overwrites the database if it already exists using the WITH REPLACE option.

If the recovery model is set to “Full”, you need to use the WITH REPLACE option.

Use the FILE = parameter to choose which file you’d like to restore (can be full or differential).

If the recovery model is set to “Simple”, you don’t need the WITH REPLACE option.

If the database does not exist, it will create the database, including its files and data.

Restores from a .BAK file.

**--If the recovery model is set to Full - also choosing the file # 1 in the backup set**

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK' WITH FILE = 1, REPLACE
```

**--If recovery model is set to Simple**

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK'
```

**Differential Restore**

Use RESTORE HEADERONLY in order to see the backup files you have available, i.e. the full and differential backup file types.

You must perform a full restore of the .BAK file first with the NORECOVERY OPTION, as the NORECOVERY option indicates other files need to be restored as well.

Once you've specified the full backup file to be restored and use WITH NORECOVERY, you can restore the differential.

Finally, you should include the RECOVERY option in your last differential file restore in order to indicate that there are no further files to be restored.

As a reference, the syntax below uses the files from the previous screenshot of RESTORE HEADERONLY.

**--Performs a restore of the full database backup file first**

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK' WITH FILE = 8, NORECOVERY
```

**--Now performs a restore of the differential backup file**

```
RESTORE DATABASE DatabaseName FROM DISK =  
'C:\SQLBackups\DatabaseName.BAK' WITH FILE = 9, RECOVERY
```

**Log Restore**

This is the last step to perform a thorough database restore.

You must restore a “Full” or “Differential” copy of your database first, then restore the log.

- Finally, you must use the NORECOVERY option when restoring your database backup(s) so that your database stays in a restoring state and allows you to restore the transaction log.

Since you already performed a backup of your database and “lost” data in the Sales.Product\_Sales table, it's now time to restore the database in full.

In order to do this, you'll first need to use the ‘master’ database, then set your database to a single user state, perform the restore, and finally set it back to a multi-user state.

Once you've restored the database, retrieve all data from the Sales.Product\_Sales table to verify that the data exists again.

***Attaching and Detaching Databases***

As you know, because you already went through this portion, the methodology of attaching and detaching databases is similar to backups and restores.

Essentially, here are the details of this method:

Allows you to copy the .MDF file and .LDF file to a new disk or server.

Performs like a backup and restore process, but can be faster at times, depending on the situation.

- The database is taken offline and cannot be accessed by any users or applications. It will remain offline until it's been reattached.

So, which one should you choose? Though a backup is the ideal option, there are cases where an attachment/detachment of the database may be your only choice.

Consider the following scenarios:

Your database contains many file groups. Attaching that can be quite cumbersome.

The best solution would be to back up the database and then restore it to the desired destination, as it will group all of the files together in the backup process.

Based on the size of the database, the backup/restore process takes a long time. However, the attaching/detaching of the database could be much quicker if it's needed as soon as possible.

- In this scenario, you can take the database offline, detach it, and reattach to the new destination.

As you know, there are two main file groups when following the method of attaching databases. These files are .MDF and .LDF. The .MDF file is the database's primary data file, which holds its structure and data. The .LDF file holds the transactional logging activity and history.

However, a .BAK file that's created when backing up a database groups all of the files together and you restore different file versions from a single backup set.

Consider your situation before using either option, but also consider a backup and restore to be your first option, then look into the attach/detach method as your next option. Also, be sure to test it before you move forward with live data!

## ***Attaching/Detaching the AdventureWorks2012 Database***

Since you already attached this database, let's now detach it from the server. After that, you'll attach it again using SQL syntax.

### ***Detaching the Database***

In SQL Server, there's a stored procedure that will detach the database for you. This particular stored procedure resides in the 'master' database. Under the hood, you can see the complexity of the stored procedure by doing the following:

Click to expand the Databases folder

Click on System Databases, then the Master database

Click on Programmability

Click on Stored Procedures, then System Stored Procedures

- Find sys.sp\_detach\_db, right-click it and select 'Modify' in SSMS. You'll then see its syntax.

For this, simply execute the stored procedure as is.

The syntax is the following:

USE master

GO

```
ALTER DATABASE DatabaseName SET SINGLE_USER WITH  
ROLLBACK IMMEDIATE
```

GO

```
EXEC master.dbo.sp_detach_db @dbname = N'DatabaseName',  
@skipchecks = 'false'
```

GO

To expand a little on what is happening, you want to use the ‘master’ database to alter the database you’ll be detaching and set it to single user instead of multi-user.

Finally, the value after @dbname allows you to specify the name of the database to be detached, and the @skipchecks being set to false means that the database engine will update the statistics information, identifying that the database has been detached. It’s ideal to set this as @false whenever detaching a database so that the system holds current information about all databases.

Now, detach the AdventureWorks2012 database from your server instance. Use the above SQL example for some guidance.

## **Attaching Databases**

Once you have detached your database, if you navigate to where your data directory is, you’ll see that the AdventureWorks2012\_Data.MDF file still exists – and it should since you only detached it and haven’t deleted it.

Next, take the file path of the .MDF file and copy and paste it in a place that you can easily access, like on the notepad.

*My location is C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA.*

Once you’ve connected to your instance and opened up a new query session, you will just need to use the path where the data file is stored. Once you have that, you can enter that value in the following SQL syntax examples in order to attach your database.

Below is the syntax used to attach database files and log files. So, in the following exercise, you’ll skip attaching the log file completely, since you’re not attaching this to a new server. So, you may omit the statement to attach the log file.

```
CREATE DATABASE DatabaseName ON (FILENAME = 'C:\SQL Data  
Files\DatabaseName.mdf'), (FILENAME      =      'C:\SQL      Data  
Files\DatabaseName_log.ldf') FOR ATTACH
```

In the above example, I am calling out the statement to attach the log file if one is available. However, if you happen not to have the .LDF file but only the .MDF file, that’s alright. You can just attach the .MDF file and the

database engine will create a new log file and start writing activity to that particular log.

For this exercise, imagine that you've had to detach this database from an old server and that you're going to attach it to a new server. Go ahead and use the syntax above as an example in order to attach the AdventureWorks2012 database.

Each exercise gives an overview, as well as examples, before you start applying what you learn in each section.

Below is a sample of the Production.Product table that you'll be using in the next couple of exercises.

| Product ID | Name            | Product Number | Color  | Standard Cost | List Price | Size |
|------------|-----------------|----------------|--------|---------------|------------|------|
| 317        | LL Crankarm     | CA-5965        | Black  | 0             | 0          | NULL |
| 318        | ML Crankarm     | CA-6738        | Black  | 0             | 0          | NULL |
| 319        | HL Crankarm     | CA-7457        | Black  | 0             | 0          | NULL |
| 320        | Chainring Bolts | CB-2903        | Silver | 0             | 0          | NULL |
| 321        | Chainring Nut   | CN-6137        | Silver | 0             | 0          | NULL |

## ***Query Structure and SELECT Statement***

Understanding the syntax and its structure will be extremely helpful for you in the future. Let's delve into the basics of one of the most common statements, the SELECT statement, which is used solely to retrieve data.

`SELECT Column_Name1, Column_Name2`

`FROM Table_Name`

In the above query, you're selecting two columns and all rows from the entire table. Since you're selecting two columns, the query performs much faster than if you had selected all of the columns like this:

`SELECT *`

`FROM Table_Name`

Select all of the columns from the table Production.Product.

Using the same table, select only the ProductID, Name, Product Number, Color and Safety Stock Level columns.

(*Hint: The column names do not contain spaces in the table!*)

### **The WHERE Clause**

The WHERE clause is used to filter the amount of rows returned by the query. This clause works by using a condition, such as if a column is equal to, greater than, less than, between, or like a certain value.

When writing your syntax, it's important to remember that the WHERE condition comes after the FROM statement.

**EQUALS** – This is used to find an exact match. The syntax below uses the WHERE clause for a column that contains the exact string of ‘Value’. Note that strings need single quotes around them, while numerical values do not.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = 'Value'
```

**BETWEEN** – Typically used to find values between a certain range of numbers or date ranges. It's important to note that the first value in the BETWEEN comparison operator should be lower than the value on the right. Note the example below comparing between 10 and 100.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 BETWEEN 10 AND 100
```

**GREATER THAN, LESS THAN, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO** – SQL Server has comparison operators that you can use to compare certain values.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = < 1000 --Note that you can also use <, >, >= or even <> for not equal to
```

**LIKE** – This searches for a value or string that is contained within the column. You still use single quotes but include the percent symbols

indicating if the string is in the beginning, end or anywhere between ranges of strings.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 LIKE '%Value%' --Searches for the word 'value' in any field

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 LIKE 'Value%' --Searches for the word 'value' at the beginning of the field

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 LIKE '%Value' --Searches for the word 'value' at the end of the field

IS NULL and IS NOT NULL – As previously discussed, NULL is an empty cell that contains no data. Eventually, you'll work with a table that does contain NULL values, which you can handle in a few ways.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 IS NULL --Filters any value that is NULL in that column, but don't put NULL in single quotes since it's not considered a string.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

WHERE Column\_Name3 IS NOT NULL --Filters any value that is not NULL in that column.<sup>40</sup>

Using the Production.Product table again, select the Product ID, Name, Product Number and Color. Filter the products that are silver colored.

## **Using ORDER BY**

The ORDER BY clause is simply used to sort data in ascending or descending order, specific to which column you want to start. This command

also sorts in ascending order by default, so if you want to sort in descending order, use DESC in your command.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = 'Value'
```

ORDER BY Column\_Name1, Column\_Name2 --Sorts in descending order, but you can also include ASC to sort in ascending order.

```
SELECT Column_Name1, Column_Name2
```

```
FROM Table_Name
```

```
WHERE Column_Name3 = 'Value'
```

ORDER BY Column\_Name1, Column\_Name2 DESC --This sorts the data in descending order by specifying DESC after the column list.

Then, sort the results in the list price from smallest to largest.

*(Hint: You won't use the \$ in your query and if needed, you can refer back to the sorting options you just reviewed.)*

# CHAPTER 7:

## SQL Transaction

*“Just because something doesn’t do what you planned it to do doesn’t mean it’s useless.” – Thomas Edison*

### **What is an SQL Transaction?**

Transactions are a sequence of tasks performed in a logical order against a database. We consider each transaction as a single unit of work. Transactions give you more control over your SQL behavior, which becomes essential if you are continuing to maintain your data integrity, or planning to avoid database errors.

In Chapter 3, we talked at length about data integrity and the various methods one can adopt to ensure the presence of such integrity. However, that takes into account the fact that somehow, multiple users will not have access to the same data.

In reality, this may not be the case.

In a situation where many users are trying to modify the data, there are chance occurrences of actions overlapping each other. In many instances, one user takes specific operations on data that may not be valid. She or he does not realize that the data has been compromised because another user took action at the same time. As neither user will be immediately aware of any form of change taking place, the data might continue to remain inconsistent for quite a while.

With this in mind, all users will continue to assume the data is intact and that there are no problems for them to oversee.

Let us try to understand the above scenario with an example. Assume that there are two users, User A, and User B, who are both working on a customer’s data at the same time.

User A might notice that the customer’s last name is incorrect, and will set about changing the name, immediately saving the data after making the change. User B might be working on another section of the customer’s data, such as the customer’s address or telephone number. However, User A’s changes might not reflect on User B’s side. User B might still be working

on the customer's data using the old, and wrong, last time, inadvertently changing the last name to the incorrect entry when saving the data.

Now both users are unaware of the inconsistencies in their work, and this problem might go unnoticed for a long while. It might seem somewhat trivial when contemplating the fact that only a single error took place. However, imagine the occurrences repeating themselves overtime, creating inconsistencies on numerous data. When the time comes, you will be forced to look through all the data you have. If your database is a large one, this might take a considerable amount of your time.

With SQL transactions, consecutive actions or changes will not affect the validity of data that is seen by another user.

SQL transactions give you the power to commit SQL statements, which will apply said statements into the database. Alternatively, you have the ability to rollback, which will undo statements from the database.

In an SQL transaction, an action will convert into a transaction if it successfully passes four characteristics, denoted by the acronym ACID.

### **Atomic**

This property focuses on the “everything or nothing” principle of a transaction. It ensures the performance of all the operations in a transaction. If an operation fails any point, the entire transaction they will be aborted. Additionally, if only a few statements are executed, the transactions are rolled back to their previous state. To complete a transaction successfully and to apply it to the database, all operations should be correct and none should be missing.

### **Consistent**

With this property, the database must not merely be consistent at the beginning of the transaction, but after its completion as well. Hence, if a set of operations result in actions that change the consistency at any point of their performance, then the transaction will be reverted to its original state.

### **Isolated**

When a particular transaction is taking place, the data might be inconsistent. This inconsistency has the potential to affect the database and hence until the transaction is completed, the data will not be available for other operations. It will be free from any outside effect. In other others, it will be

isolated. Additionally, no other transaction can influence or affect the isolated transaction in any way, ensuring that there is no compromise in integrity.

## Durable

When all transactions are completed, any changes or modifications must be preserved and maintained. In the event of hardware or application error, there should be no loss of data; it should be available reliably and consistently. This characteristic ensures that your data is available in the event of such

One has to note that at any time, whether operations are completed or reverted, the transaction maintains the integrity of the database.

There are seven commands that you can use to manage transactions. These are:

**COMMIT:** Saves changes and commits them to the database.

**ROLLBACK:** Rolls back changes to a specific SAVEPOINT, or the beginning of a transaction. Changes are not applied to the database.

**SAVEPOINT:** Creates a savepoint – a form of marker – within a transaction. When you ROLLBACK, you can reach this point instead of the beginning of the transaction.

**RELEASE SAVEPOINT:** Removes or releases a savepoint. After this command, any ROLLBACK will revert the transaction to its original state.

**SET TRANSACTION:** Creates characteristics for the execution of a particular transaction.

**START TRANSACTION:** Sets the characteristics of the transactions and begins the transaction.

**SET CONSTRAINTS:** Establishes the constraint mode within a transaction. A constraint creates a condition where a decision is passed to either apply the constraint as soon as modifications on the data begin to occur or delay the application of the constraint until the completion of the transaction.

Time for another example.

It is important to remember that when you are about to start a transaction, the database is in its original state. At this state, you can expect the data to

be consistent. Once you execute the transactions, the SQL statements within those transactions begin processing. If the process is successful, then one uses the COMMIT command. This command causes the SQL statements to update the database and dismiss the transaction. If the process of updating is not successful, then the ROLLBACK command comes into play. This process gives you a brief overview of commands. However, we will be going in-depth into their functions.

## ***Commit***

When all the statements are accomplished in a specific transaction, then the next step would be to terminate the transaction. One of the recommended ways to terminate a transaction is to commit all the changes to the database made so far. When you are confident of the changes you have performed so far, you can then utilize the COMMIT command. See the below syntax statements that you can use to commit a statement:

COMMIT [WORK] [AND CHAIN]

COMMIT [WORK] [AND NO CHAIN]

You must remember that the usage of the WORK entry is not mandatory. Only the COMMIT keyword will suffice to process the COMMIT command. In actuality, the COMMIT and COMMIT WORK entries have the same purpose. Earlier versions of SQL used to have the inclusion of the WORK keyword and some users might be more familiar with it. However, if you are not used to its usage, then you do not have to include it to execute the COMMIT command.

The AND CHAIN article is another optional inclusion in the COMMIT statement. It is not a commonly used statement in SQL implementations. The AND CHAIN clause simply tells the system to begin with a new transaction as soon as the current one ends. With this clause, you will have no need to use the SET TRANSACTION or START TRANSACTION statements, leaving the work to the system to continue to the next step automatically.

However, this might prove to be a disadvantage rather than a convenience. As you will need complete control over your transaction, it is best to use an alternative. Rather than using AND CHAIN in your COMMIT statement, try to utilize the AND NO CHAIN parameter. This essentially lets your

system know that it should not begin a new transaction with the same settings are the current one. In the case where there is no mention of either of the parameters (AND CHAIN or AND NO CHAIN), the AND NO CHAIN will be the default clause the system will adopt.

Your COMMIT statement might look like the below:

COMMIT;

We have not included any of the two clauses mentioned. However, know that if you would like the system to start a new transaction automatically, use the below statement:

COMMIT AND CHAIN;

Assume you have a table named EMPLOYEES with “names” and “salaries” columns.

## **NAME SALARIES**

John        5,000

Mary        7,000

Jennifer    9,000

Mark        10,000

Adam        7,000

If you would like to remove salaries that are 7,000, then you simply have to use the below:

SQL > DELETE FROM EMPLOYEES

            WHERE SALARIES = 7,000

SQL > COMMIT;

Now, as mentioned before, if you would like the next transaction to begin after the execution of the current one, then your statement will look something like this:

SQL > DELETE FROM EMPLOYEES

WHERE SALARIES = 7,000

SQL > COMMIT AND CHAIN;

### ***Rollback***

Human error is a permanent fixture in any process. For this reason, if you find a situation that calls for a roll back, then this statement gives you the power to commit such an action.

By using ROLLBACK, you will undo the statements you have done so far, either bringing them back to a specific savepoint – if you had established such a savepoint – or sending them to the beginning of the transaction.

Let us look at some of the parameters or inclusions of a ROLLBACK statement:

ROLLBACK [WORK] [AND CHAIN]

ROLLBACK [WORK] [AND NO CHAIN]

When a user makes changes to data but does not use the COMMIT statement, then those changes are stored in a temporary format called a transaction log. Users can look at this unsaved data, analyzing it to check if it meets particular requirements. If the changes do not express the desired result users are aiming for, they can hit ROLLBACK, ensuring that the data is not saved, and they can work on it again.

Users can also use the ROLLBACK feature to send the database to a savepoint or the beginning of a transaction in the event of a hardware malfunction or application crash. If a power loss interrupted your work, then as soon as the system restarts, the ROLLBACK feature will review all pending transactions and will roll back all statements.

Unlike the COMMIT statement, ROLLBACK has the option to include a TO SAVEPOINT parameter. When using the TO SAVEPOINT clause, the system will not cancel the transaction. You will merely be taken to a specific point from where you can continue your work.

Should you wish to cancel the transaction simply remove the TO STATEMENT parameter from your statement.

Here is an example of the most fundamental form of a ROLLBACK statement:

```
ROLLBACK;
```

Notice that we have not used the AND CHAIN parameter. This is because, as mentioned before in the section for COMMIT command, the system uses the AND NO CHAIN command by default. Alternatively, you can use the AND CHAIN command should you wish to initiate a new transaction after the conclusion of the current one. However, bear in mind that you cannot use the TO SAVEPOINT and AND CHAIN parameters at the same time. This is because a TO SAVEPOINT command requires the termination of the current transaction.

When including the TO SAVEPOINT parameter, ensure that you add the name of the savepoint. See an example of this process below:

```
ROLLBACK TO SAVEPOINT;
```

The SAVEPOINT value is replaced by the name of the savepoint as shown in the example below:

```
BEGIN
```

```
    SELECT customer_number, customer_lastname, purchases  
    DELETE FROM customer_number WHERE purchases < 10000;  
    SAVEPOINT section_1;  
    ROLLBACK TO section_1;
```

```
END;
```

While the above example is a rather simple form of the command, its purpose is to give you clarity on SAVEPOINT usages and naming. Do note that you can enter as many parameters between the SAVEPOINT command and the ROLLBACK clause.

If you specify a savepoint name, then the rollback command will take you back to that particular savepoint, irrespective of the fact that there could be other savepoints in between.

```
BEGIN
```

```
SELECT customer_number, customer_lastname, purchases  
DELETE FROM customer_number WHERE purchases < 10000;  
SAVEPOINT section_1;  
SAVEPOINT section_2;  
SAVEPOINT section_3;  
SAVEPOINT section_4;  
ROLLBACK TO section_1;  
END;
```

Notice that even though you have SAVEPOINTS named from section\_2 to section\_4, you have asked the system to roll back to section\_1. By doing so, the system will ignore all of the other savepoints.

## ***Savepoint***

While we are on the subject of savepoints, let us look at why these commands are important, and how to work with them.

Essentially, you will be working with a complex set of transactions. To make it easier to understand the transaction, you ill group different sections into units. Breaking down transactions in this matter will allow you to identify problems easily and know which section to reach in order to solve the issue.

But the question arises: how can you reach these sections without having to change the entire transaction?

Why, you use the SAVEPOINT command, of course!

At this point, you should be aware of an important point. MySQL and Oracle support the SAVEPOINT parameter, but if you are working with SQL Server, you might be using the SAVE TRANSACTION query instead.

However, the functions of both SAVEPOINT and SAVE TRANSACTION parameters are the same.

Let us look at an example to understand how a savepoint function works.

STEP 1: Start Transaction

STEP 2: SQL Statements

STEP 3: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to the beginning of the transaction

STEP 4: Enter SAVEPOINT 1

STEP 5: SQL Statements

STEP 6: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to SAVEPOINT 1

STEP 7: Enter SAVEPOINT 2

STEP 8: SQL Statements

STEP 9: If the SQL statement is successful, the commence execution. If the SQL statement is not successful, then roll back to SAVEPOINT 2

STEP 10: COMMIT

In the above example, we set savepoints after working on SQL statements. When we perform this routine, we begin to work on specific SQL statements before jumping on to the next one. This ensures that we execute all statements properly, and if there is an error in the statements, we can rework them. Once the system deems statements fully successful, we can use a savepoint and move on to the next set of operations. By doing this, we save the integrity of the first group of operations. After the savepoint, we can continue our progress, knowing that there will not be any changes occurring to the first statements.

This becomes a convenient method of dealing with a set of actions without worrying about the integrity of previous actions.

The process of creating a savepoint is rather simple. You just have to use the following command:

**SAVEPOINT <name of the savepoint>**

The savepoint name does not have to be “SECTION\_1”. That was the name chosen for the purpose of the example. You can select your own name, preferably something you can easily recollect when you want to.

### ***Release savepoint***

After you have completed certain operations within a transaction, you might not require the savepoints you had previously established. In order to

remove them, you can use the RELEASE SAVEPOINT command.

However, do note that once you release a savepoint, you will not be able to roll back to it again. Which is why the ideal way to release a savepoint is to check if you are satisfied with your work so far. Due diligence might be an added task, but it will provide you with the capability to complete your work in confidence.

To release a savepoint, simply enter the below command:

```
RELEASE SAVEPOINT <name of the savepoint>
```

Let us take an example where you have the below savepoints:

```
SAVEPOINT section_1;
```

```
SAVEPOINT section_2;
```

```
SAVEPOINT section_3;
```

```
SAVEPOINT section_4;
```

In order to release all of them, you will have to specify each one.

Hence, the process is as shown below:

```
RELEASE SAVEPOINT section_1;
```

```
RELEASE SAVEPOINT section_2;
```

```
RELEASE SAVEPOINT section_3;
```

```
RELEASE SAVEPOINT section_4;
```

You might release all savepoints at one time or you might choose to release one of the savepoints while keeping the rest. This is acceptable. All you have to ensure is that you use the right savepoint name. Another note to remember is that you do not have to use the release command in the order of the savepoints you have created. You can choose to release any savepoint at any time, and in any order.

## ***Set Transaction***

This command allows you to work with the different properties of a transaction. These could be one of the below:

- Read Only

- Read Write
- Specify an isolation level
- Attach it to a rollback property

Any effect of the SET TRANSACTION operation affects only your transaction. Other users will not notice any changes on their end.

You can establish SET TRANSACTION using the below command:

`SET TRANSACTION <mode>`

In the above command, the `<mode>` refers to the type of option you would like to specify. You can use three modes in SET TRANSACTION. These are:

- Access Level
- Isolation Level
- Diagnostics Size

You can add multiple transaction modes into the `<mode>` placeholder. If you wish to do that, separate the different modes by a comma. However, you cannot repeat the same mode again.

Example: you can include an isolation level and work on the diagnostics size. However, you cannot include two diagnostics size mode.

Now let us try to look at each mode separately.

## **Access Level**

In a SET TRANSACTION, there are two types of access levels; READ ONLY and READ WRITE. If you select the READ ONLY option, the system will not allow you to make any changes to the database. However, if you choose the READ WRITE access level, you will be able to add statements in your transactions to modify the data or even the database structure.

## **Isolation Level**

Do you remember how we talked about isolating transactions so that nothing can influence it while you are working on it? Well, here we will talk about setting up isolation levels.

For a SET TRANSACTION, you can use four isolation levels, as seen in the syntax below:

SET TRANSACTION ISOLATED LEVEL

READ UNCOMMITTED  
READ COMMITTED  
REPEATABLE READ  
SERIALIZABLE

The isolation levels are arranged in the order of their effectiveness, with the READ UNCOMMITTED being the least level of isolation you can use and the SERIALIZABLE being the highest level of isolation. If you do not specify the level of isolation, the system will assume the SERIALIZATION level, giving your transaction maximum isolation.

### **Diagnostic Size**

The SET TRANSACTION allows you to specify a diagnostic size. The size you include lets the system know how much area to allow you for conditions. In an SQL statement, a condition is a message, warning, or other notifications raised by the statement. If you do not specify a diagnostic size, then your database will automatically assign you one. This number is not fixed. It varies from database to database.

### ***Start Transaction***

In a database, you can start a transaction explicitly or automatically when you begin executing a command.

For example, you can start a transaction when you use commands such as DELETE, CREATE TABLE, and so on.

Alternatively, you can commence a transaction by using the START TRANSACTION statement.

As with the SET TRANSACTION, you can specific one of more modes here as well. These are the access level, isolation level, and diagnostic size modes that we had mentioned before.

Here is an example of a START TRANSACTION command:

START TRANSACTION

```
READ ONLY  
ISOLATION LEVEL SERIALIZED  
DIAGNOSTICS SIZE 10;
```

Once you enter the syntax above, the START TRANSACTION will execute the statement and its operations.

### ***Set Constraint***

When you are working with transactions, you might encounter scenarios where your work will go against the constraints established in the database. Let us take an example.

Assume that you have a table that includes the NO NULL constraint. Now, while you are working on the table, you realize that you might not have a value to place instead of NO NULL constraint at that point in time. But you cannot leave without entering a value. This forces you to insert random values into the section.

To avoid such situations, you can define a constraint as deferrable.

The syntax for this action is shown below:

```
SET CONSTRAINTS (constraint names) DEFERRED/IMMEDIATE
```

To mention multiple constraint names, you need to separate them by a bar. For example:

```
(constraint_1 | constraint_2 | constraint_3)
```

However, if you wish to choose all constraints – and this could be useful when you have lots of them in your database – then you simply have to use the word ALL

Here is an example:

```
SET CONSTRAINTS ALL DEFERRED/IMMEDIATE
```

Another thing to note is that you do not have to mention both DEFERRED and IMMEDIATE. You have to choose the one that fits your activity at that point.

For example, if you are working on the database, you will choose to defer the constraints, in which case your statement should look like this:

```
SET CONSTRAINTS ALL DEFERRED
```

Once you have completed operations on the database, you can then choose to apply the changes. In that case, the statement will look like this:

**SET CONSTRAINT ALL IMMEDIATE**

So through this chapter, you have understood the idea behind SQL transactions, and why they are essential when working with SQL programming.

# CHAPTER 8:

## Logins, Users and Roles

*“See first that the design is wise and just; that ascertained, pursue it resolutely.” – William Shakespeare*

### **Server Logins**

Server logins are user accounts that are created in SQL servers and use SQL Server authentication, which entails entering a password and username to log into the SQL Server instance. SQL Server authentication is different from Windows Authentication, as Windows doesn't prompt you to enter your password.

These logins can be granted access to be able to log in to the SQL Server, but don't necessarily have database access depending on permissions that are assigned. They can also be assigned certain server level roles that provide certain server permissions within SQL Server.

### **Server Level Roles**

Permissions are wrapped into logins by Server Level Roles. The Server Level Roles determine the types of permissions a user has.

There are nine types of predefined server level roles:

Sysadmin - can perform any action on the server, essentially the highest level of access.

Serveradmin - can change server configurations and shutdown the SQL server.

Securityadmin - can GRANT, DENY and REVOKE server level and database level permissions, if the user has access to the database. The securityadmin can also reset SQL Server passwords.

Processadmin - provides the ability to end processes that are running in a SQL Server instance.

Setupadmin - can add or remove linked servers by using SQL commands.

Diskadmin - this role is used to manage disk files in the SQL Server.

Dbcreator - provides the ability to use DDL statements, like CREATE, ALTER, DROP and even RESTORE databases.

Public - all logins are part of the ‘public’ role and inherit the permissions of the ‘public’ role if it does not have specific server or database object permissions.

If you really want to delve deeper into this topic (by all means, I encourage you to do so), then go ahead and [click this link](#) to get more information on server roles via the Microsoft website.

Below is an example of syntax that can be used to create server logins. You are able to swap out ‘dbcreator’ with any of the roles above when you assign a server level role to your login.

You'll also notice the brackets, [], around ‘master’ and ‘User Name’ for instance. These are delimited identifiers, which we have previously discussed.

--First, use the master database

```
USE [master]
```

```
GO
```

--Creating the login - replace 'User A' with the login name that you'd like to use

--Enter a password in the 'Password' field - it's recommended to use a strong password

--Also providing it a default database of 'master'

```
CREATE LOGIN [User A] WITH PASSWORD= N'123456',
```

```
DEFAULT_DATABASE=[master]
```

```
GO41
```

Create a user with a login name called Server User. Give this user a simple password of 123456, and give it default database access to ‘master’.

## ***Assigning Server Roles***

There are a few different ways to assign these roles to user logins in the SQL Server. One of the options is to use the interface in Management Studio, and the other is to use SQL syntax.

Below is the syntax to give a user a particular server role. If you recall the previous syntax we used to create the login for ‘User A’, this will give ‘User A’ a server role of dbcreator.

--Giving the User a login a role of 'dbcreator', (it has the public role by default)

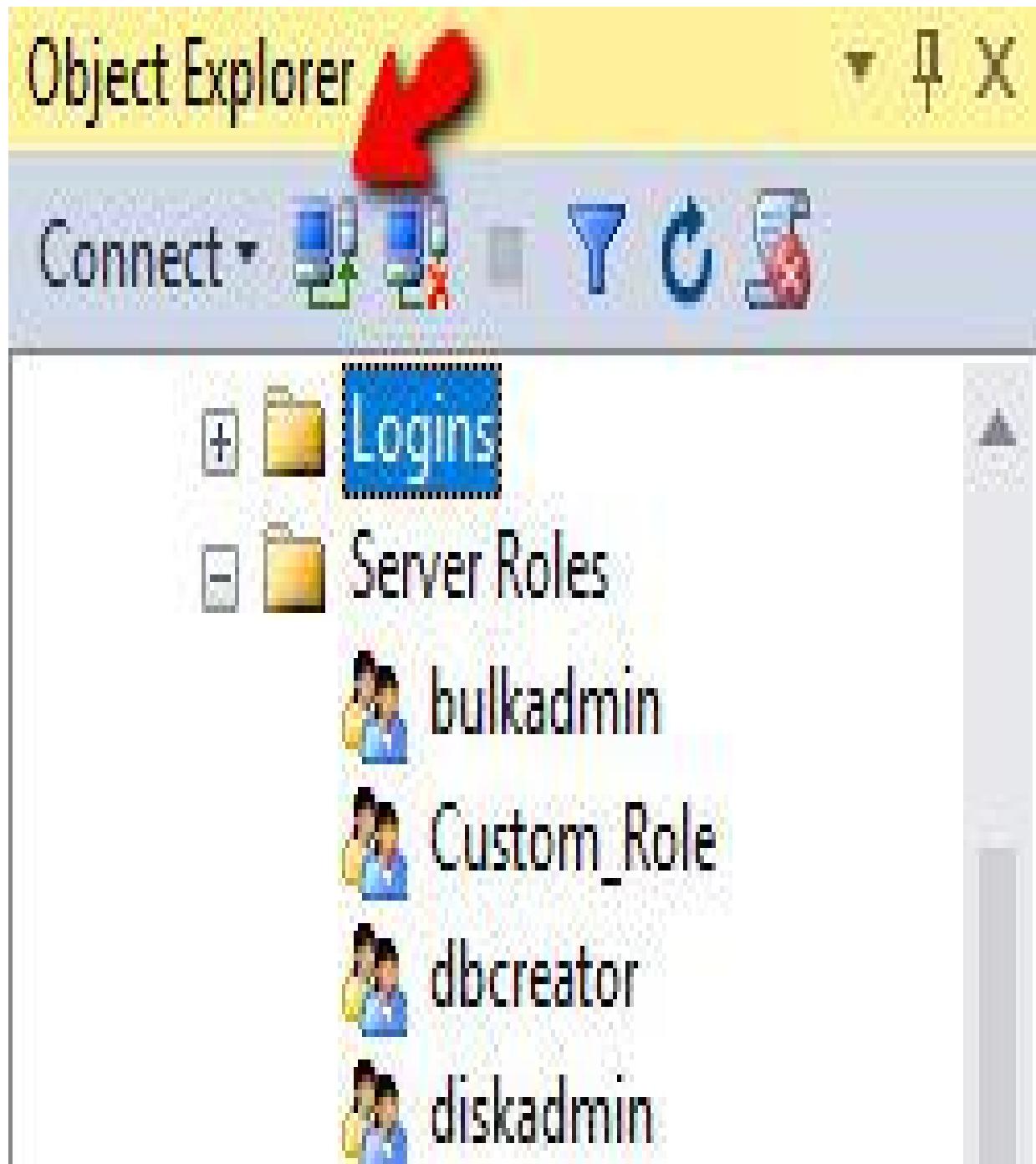
```
ALTER SERVER ROLE dbcreator ADD MEMBER [User A]
```

```
GO
```

For the above user that you created, called Server User, give it a server role of serveradmin.

After you've successfully completed the exercise, you'll receive a message telling you that your command has been completed successfully. You can now login to SQL Server using that account!

To log in, just navigate to the 'Connect Object Explorer' button in the Object Explorer window and click it to bring up the connection window.



When the window comes up to login, select the drop-down where it says 'Windows Authentication' and select 'SQL Server Authentication' instead. Just enter the user name and password and log in!



Once you are able to log in with this user, try to open the databases (other than the system databases) in the Object Explorer. It should look something like this; when you try to expand a database, it displays as blank instead.

## Object Explorer

Connect ▾



+ HOME (SQL Server 13.0.4206 - HOME\jacob)

- HOME (SQL Server 13.0.4206 - User A) 

- Databases 

+ System Databases 

+ Database Snapshots 

AdventureWorks2012  

+ Company 

+ ReportServer 

+ ReportServerTempDB 

+ Sample2 

+ testdb 

+ Security 

+ Server Objects 

+ Replication 

+ Management 

You shouldn't be able to since this user doesn't have any database access. We'll cover this more thoroughly in the next section.

## ***Database Users***

A database user is a user that's typically associated with a login and mapped to a database or set of databases. The purpose of this database user is to provide access to the databases within the server itself. You can also restrict access to other databases.

In many cases, there are multiple databases on one server instance and you wouldn't want to give all database access to all of the users.

## ***Database Level Roles***

Much like the server level roles, there are database level roles that can be assigned to a user to control their permissions within certain databases.

There are nine predefined database roles:

`Db_owner` - provides the ability to perform all configuration and maintenance jobs on the database. This role also allows a user to `DROP` databases - a powerful command!

`Db_securityadmin` - provides the ability to modify role membership and manage permissions.

`Db_accessadmin` – provides the ability to add or remove access to the database for Windows logins, Windows groups and SQL logins.

`Db_backupoperator` - allows the user to back up the database.

`Db_ddladmin` - provides the ability to run any DDL statement in SQL Server.

`Db_datawriter` - provides the ability to add, delete or modify data within any user table.

`Db_datareader` - provides the ability to view any data from user tables.

`Db_denydatawriter` - cannot add, delete or modify data within any user table.

`db_denydatareader` - cannot view any data from user tables.

## ***Assigning Database Roles and Creating Users***

Like assigning server roles and creating logins, you can use either Management Studio or SQL syntax to create a database user and assign

database level roles to that particular user. You can also map that user to a login so that the two are correlated.

Below is some sample syntax to associate a database user to a login, as well as assigning it a database role. It can become a little extensive, but I've broken it down into four parts to make it easier.

--Using the database that we'd like to add the user to

```
USE [AdventureWorks2012]
```

```
GO
```

--Creating the user called 'User A' for the login 'User A'

```
CREATE USER [User A] FOR LOGIN [User A]
```

```
GO
```

--Using AdventureWorks2012 again, since the database level role needs to be properly

--mapped to the user

```
USE [AdventureWorks2012]
```

```
GO
```

--Altering the role for db\_datawriter and adding the database user 'User A' so

--the user can add, delete or modify existing data within AdventureWorks2012

```
ALTER ROLE [db_datawriter] ADD MEMBER [User A]
```

```
GO
```

## **LIKE Clause**

SQL provides us with the LIKE clause that helps us compare a value to similar values using the wildcard operators.

The wildcard operators that are used together with the LIKE clause include the percentage sign (%) and the underscore (\_).

The symbols have the syntax below:

```
SELECT FROM tableName
```

```
WHERE column LIKE 'XXX%'
```

Or the following syntax:

```
SELECT FROM tableName  
WHERE column LIKE '%XXX%'
```

Or the following syntax:

```
SELECT FROM tableName  
WHERE column LIKE 'XXX_'
```

Or the following syntax:

```
SELECT FROM tableName  
WHERE column LIKE '_XXX'
```

Or the following syntax:

```
SELECT FROM tableName  
WHERE column LIKE '_XXX_'
```

If you have multiple conditions, combine them using the conjunctive operators (AND, OR). The XXX in the above case represents any string or numerical value.

Let's describe the meaning of some of the statements that you can create with the LIKE clause:

### 1. WHERE SALARY LIKE '300%'

To return any values that begin with 300.

### 2. WHERE SALARY LIKE '%300%'

This will return any values with 300 anywhere.

### 3. WHERE SALARY LIKE '\_00%'

This will find the values with 00 in second and third positions.

#### 4. WHERE SALARY LIKE '4\_%\_%'

This will find the values that begin with 4 and that are at least 3 characters long.

#### 5. WHERE SALARY LIKE '%3'

This will find the values that end with a 3.

#### 6. WHERE SALARY LIKE '\_3%4'

This will look for the values with a 3 in the second position and end with a 4.

#### 7. WHERE SALARY LIKE '3\_\_\_4'

This will find the values in a 5 digit number that begin with a 3 and end with a 4.

Now, we need to demonstrate how to use this clause. We will use the EMPLOYEES table with the data shown below:<sup>42</sup>

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nico1442 | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

Let's run a command that shows us all records in which the value of salary begins with 400:

```
SELECT * FROM EMPLOYEES
```

```
WHERE SALARY LIKE '400%';
```

The command returns the following:

```
mysql> SELECT * FROM EMPLOYEES
-> WHERE SALARY LIKE '400%';
+----+-----+-----+---+-----+
| ID | NAME | ADDRESS | AGE | SALARY |
+----+-----+-----+---+-----+
| 3  | Joel  | Joel42  | 30 | 4000.00 |
| 7  | Grace | gra361  | 35 | 4000.00 |
+----+-----+-----+---+-----+
2 rows in set (0.13 sec)

mysql>
```

## SQL Functions<sup>43</sup>

Here's the syntax:

```
SELECT COUNT (<expression>)
```

```
FROM table_name;
```

| ID   | EMP_NAME      | SALES    | BRANCH      |
|------|---------------|----------|-------------|
| 1001 | ALAN MARSCH   | 3000.00  | NEW YORK    |
| 1098 | NEIL BANKS    | 5400.00  | LOS ANGELES |
| 2005 | RAIN ALONZO   | 4000.00  | NEW YORK    |
| 3008 | MARK FIELDING | 3555.00  | CHICAGO     |
| 4356 | JENNER BANKS  | 14600.00 | NEW YORK    |
| 4810 | MAINE ROD     | 7000.00  | NEW YORK    |
| 5783 | JACK RINGER   | 6000.00  | CHICAGO     |
| 6431 | MARK TWAIN    | 10000.00 | LOS ANGELES |
| 7543 | JACKIE FELTS  | 3500.00  | CHICAGO     |
| 8934 | MARK GOTH     | 5400.00  | AUSTIN      |

10 rows in set (0.00 sec)

In this statement, the expression can refer to an arithmetic operation or column name. You can also specify (\*) if you want to calculate the total records stored in the table.<sup>44</sup>

To perform a simple count operation, like calculating how many rows are in the SALES\_REP table, enter:

```
SELECT COUNT(EMP_NAME)
```

```
FROM SALES_REP;
```

Here's the result:

| COUNT(EMP_NAME) |
|-----------------|
| 101             |

1 row in set (0.24 sec)

You can also use (\*) instead of specifying a column name:<sup>45</sup>

```
SELECT COUNT(*)  
FROM SALES_REP;
```

This statement will produce the same result because the EMP\_NAME field has no NULL value. Assuming, however, that one of the fields in the EMP\_NAME contains a NULL value, this would not be included in the statement that specifies EMP\_NAME but will be included in the COUNT() result if you use the \* symbol as a parameter.

| BRANCH      | COUNT(*) |
|-------------|----------|
| AUSTIN      | 1        |
| CHICAGO     | 3        |
| LOS ANGELES | 2        |
| NEW YORK    | 4        |

4 rows in set (0.04 sec)

```
SELECT BRANCH, COUNT(*) FROM SALES_REP  
GROUP BY BRANCH;
```

This would be the output:

The COUNT() function can be used with DISTINCT to find the number of distinct entries. For instance, if you want to know how many distinct branches are saved in the SALES\_REP table, enter the following statement:

```
SELECT COUNT (DISTINCT BRANCH)  
FROM SALES_REP;
```

| COUNT(DISTINCT BRANCH) |
|------------------------|
| 4                      |

1 row in set (0.15 sec)

## SQL AVG Function<sup>46</sup>

Here is the syntax:

```
SELECT AVG (<expression>)
FROM "table_name";
```

| ID   | EMP_NAME      | SALES    | BRANCH      |
|------|---------------|----------|-------------|
| 1001 | ALAN MARSCH   | 3000.00  | NEW YORK    |
| 1098 | NEIL BANKS    | 5400.00  | LOS ANGELES |
| 2005 | RAIN ALONZO   | 4000.00  | NEW YORK    |
| 3008 | MARK FIELDING | 3555.00  | CHICAGO     |
| 4356 | JENNER BANKS  | 14600.00 | NEW YORK    |
| 4810 | MAINE ROD     | 7000.00  | NEW YORK    |
| 5783 | JACK RINGER   | 6000.00  | CHICAGO     |
| 6431 | MARK TWAIN    | 10000.00 | LOS ANGELES |
| 7543 | JACKIE FELTS  | 3500.00  | CHICAGO     |
| 8934 | MARK GOTH     | 5400.00  | AUSTIN      |

10 rows in set (0.00 sec)

In the above statement, the expression can refer to an arithmetic operation or to a column name. Arithmetic operations can have single or multiple columns.

In the first example, you will use the AVG() function to calculate the average sales amount. You can enter the following statement:<sup>47</sup>

| AVG(SALES)              |
|-------------------------|
| 6245.500000             |
| 1 row in set (0.00 sec) |

SELECT AVG(Sales) FROM Sales\_Rep;

This is the result:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

The figure 6245.500000 is the average of all sales data in the Sales\_Rep table and it is computed by adding the Sales field and dividing the result by the number of records which, in this example, is 10 rows.

The AVG() function can be used in arithmetic operations. For example, assuming that sales tax is 6.6% of sales, you can use this statement to calculate the average sales tax figure:

|                         |
|-------------------------|
| AVG(Sales * .066)       |
| 412.203000000           |
| 1 row in set (0.00 sec) |

```
SELECT AVG(Sales*.066) FROM Sales_Rep;
```

Here's the result:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

To obtain the result, SQL had to calculate the result of the arithmetic operation 'Sales \*.066' before applying the AVG function.

```
SELECT Branch, AVG(Sales) FROM Sales_Rep
```

| Branch      | Avg(Sales) |
|-------------|------------|
| AUSTIN      | 5400.00000 |
| CHICAGO     | 4351.66667 |
| LOS ANGELES | 7700.00000 |
| NEW YORK    | 7150.00000 |

4 rows in set (0.06 sec)

GROUP BY Branch;

Here's the result:

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

## SQL ROUND Function

The following is the syntax for SQL ROUND() function:

ROUND (expression, [decimal place])

In the above statement, the decimal place specifies the number of decimal points that will be returned. For instance, specifying -1 will round off the number to the nearest tens.

The examples on this section will use the Student\_Grade table with the following data:

| ID | Name         | Grade   |
|----|--------------|---------|
| 1  | Jack Knight  | 87.6498 |
| 2  | Daisy Poult  | 98.4359 |
| 3  | James McDuff | 97.7853 |
| 4  | Alicia Stone | 89.9753 |

To round off the grades to the nearest tenths, enter the following statement:

```
SELECT Name, ROUND (Grade, 1) Rounded_Grade FROM Student_Grade;
```

| Name         | Rounded_Grade |
|--------------|---------------|
| Jack Knight  | 87.6          |
| Daisy Poult  | 98.4          |
| James McDuff | 97.8          |
| Alicia Stone | 90.0          |

4 rows in set (0.46 sec)

This would be the result:<sup>48</sup>

Assuming that you want to round the grades to the nearest tens, you will use a negative parameter for the ROUND() function:

| Name         | Rounded_Grade |
|--------------|---------------|
| Jack Knight  | 90            |
| Daisy Poult  | 100           |
| James McDuff | 100           |
| Alicia Stone | 90            |

4 rows in set (0.04 sec)

```
SELECT Name, ROUND (Grade, -1) Rounded_Grade FROM Student_Grade;
```

Here's the result:

## SQL SUM Function

The SUM() function is used to return the total for an expression.

Here's the syntax for the SUM() function:

```
SELECT SUM (<expression>)  
FROM "table_name";
```

The expression parameter can refer to an arithmetic operation or a column name. Arithmetic operations may include one or more columns.

Likewise, there can be more than one column in the SELECT statement in addition to the column specified in the SUM() function. These columns should also form part of the GROUP BY clause. Here's the syntax:<sup>49</sup>

| ID   | EMP_NAME      | SALES    | BRANCH      |
|------|---------------|----------|-------------|
| 1001 | ALAN MARSCH   | 3000.00  | NEW YORK    |
| 1098 | NEIL BANKS    | 5400.00  | LOS ANGELES |
| 2005 | RAIN ALONZO   | 4000.00  | NEW YORK    |
| 3008 | MARK FIELDING | 3555.00  | CHICAGO     |
| 4356 | JENNER BANKS  | 14600.00 | NEW YORK    |
| 4810 | MAINE ROD     | 7000.00  | NEW YORK    |
| 5783 | JACK RINGER   | 6000.00  | CHICAGO     |
| 6431 | MARK TWAIN    | 10000.00 | LOS ANGELES |
| 7543 | JACKIE FELTS  | 3500.00  | CHICAGO     |
| 8934 | MARK GOTHE    | 5400.00  | AUSTIN      |

10 rows in set (0.00 sec)

SELECT column n1, column n2, ... column N, SUM ("column nN+1")

FROM table\_name;

GROUP BY column n1, column n2, ... column n\_nameN;

For the examples in this section, you will use the SALES\_REP table with the following data:<sup>50</sup>

To calculate the total of all sales from the Sales\_Rep table, enter the following statement:

SELECT SUM(Sales) FROM Sales\_Rep;

| SUM(Sales)              |
|-------------------------|
| 62455.00                |
| 1 row in set (0.00 sec) |

This would be the result:<sup>51</sup>

The figure 62455.00 represents the total of all entries in the Sales column.

To illustrate how you can use an arithmetic operation as an argument in the SUM() function, assume that you have to apply a sales tax of 6.6% on the sales figure. Here's the statement to obtain the total sales tax:

```
SELECT SUM(Sales*.066) FROM Sales_Rep;
```

|  |                         |  |
|--|-------------------------|--|
|  | SUM(Sales*.066)         |  |
|  | 4122.03000              |  |
|  | 1 row in set (0.00 sec) |  |

You will get the following result:<sup>52</sup>

In this example, you will combine the SUM() function and the GROUP BY clause to calculate the total sales for each branch. You can use the following statement:

```
SELECT Branch, SUM(Sales) FROM Sales_Rep  
GROUP BY Branch;
```

| Branch      | SUM (Sales) |
|-------------|-------------|
| AUSTIN      | 5400.00     |
| CHICAGO     | 13055.00    |
| LOS ANGELES | 15400.00    |
| NEW YORK    | 28600.00    |

4 rows in set (0.00 sec)

Here's the result:<sup>53</sup>

## SQL MAX() Function

The MAX() function is used to obtain the largest value in a given expression.

Here's the syntax:

```
SELECT MAX (<expression>)
```

```
FROM table_name;
```

The expression parameter can be an arithmetic operation or a column name. Arithmetic operations can have multiple columns.

The SELECT statement can have one or more columns, aside from the column specified in the MAX() function. If this is the case, these columns will have to form part of the GROUP BY clause.

The syntax would be:

```
SELECT column1, column2, ... "columnN", MAX (<expression>)
```

```
FROM table_name;
```

```
GROUP BY column1, column2, ... "columnN";
```

| ID   | EMP_NAME      | SALES    | BRANCH      |
|------|---------------|----------|-------------|
| 1001 | ALAN MARSH    | 3000.00  | NEW YORK    |
| 1098 | NEIL BANKS    | 5400.00  | LOS ANGELES |
| 2005 | RAIN ALONZO   | 4000.00  | NEW YORK    |
| 3008 | MARK FIELDING | 3555.00  | CHICAGO     |
| 4356 | JENNER BANKS  | 14600.00 | NEW YORK    |
| 4810 | MAINE ROD     | 7000.00  | NEW YORK    |
| 5783 | JACK RINGER   | 6000.00  | CHICAGO     |
| 6431 | MARK TWAIN    | 10000.00 | LOS ANGELES |
| 7543 | JACKIE FELTS  | 3500.00  | CHICAGO     |
| 8934 | MARK GOTH     | 5400.00  | AUSTIN      |

10 rows in set (0.00 sec)

To demonstrate this, you will use the Sales\_Rep table with this data:<sup>54</sup>

To get the highest sales amount, enter the following statement:

```
SELECT MAX(Sales) FROM Sales_Rep;
```

Here's the result:

| MAX(Sales) |
|------------|
| 14600.00   |

1 row in set (0.08 sec)

To illustrate how the MAX() function is applied to an arithmetic operation, assume that you have to compute a sales tax of 6.6% on the sales figure. To get the highest sales tax figure, use the following statement;

```
SELECT MAX(Sales*0.066) FROM Sales_Rep;55
```

| MAX(Sales*.066)         |
|-------------------------|
| 963.60000               |
| 1 row in set (0.00 sec) |

Here's the output:

You can combine the MAX() function with the GROUP BY clause to obtain the maximum sales value per branch. To do that, enter the following statement:

| Branch      | MAX(Sales) |
|-------------|------------|
| AUSTIN      | 5400.00    |
| CHICAGO     | 6000.00    |
| LOS ANGELES | 10000.00   |
| NEW YORK    | 14600.00   |

4 rows in set (0.00 sec)

SELECT Branch, MAX(Sales) FROM Sales\_Rep GROUP BY Branch;<sup>56</sup>

### ***SQL MIN() Function***

The MIN() function is used to obtain the lowest value in a given expression.

Here's the syntax:

```
SELECT MIN(<expression>)
```

```
FROM table_name;
```

The expression parameter can be an arithmetic operation or a column name. Arithmetic operations can also have several columns.

The SELECT statement can have one or several columns aside from the column specified in the MIN() function. If this is the case, these columns

will have to form part of the GROUP BY clause.

The syntax would be:

```
SELECT column1, column2, ... "columnN", MIN (<expression>)
```

```
FROM table_name;
```

```
GROUP BY column1, column2, ... "columnN";
```

To demonstrate how the MIN() function is used in SQL, use the Sales\_Rep table with the following data:<sup>57</sup>

| ID   | EMP_NAME      | SALES    | BRANCH      |
|------|---------------|----------|-------------|
| 1001 | ALAN MARSCH   | 3000.00  | NEW YORK    |
| 1098 | NEIL BANKS    | 5400.00  | LOS ANGELES |
| 2005 | RAIN ALONZO   | 4000.00  | NEW YORK    |
| 3008 | MARK FIELDING | 3555.00  | CHICAGO     |
| 4356 | JENNER BANKS  | 14600.00 | NEW YORK    |
| 4810 | MAINE ROD     | 7000.00  | NEW YORK    |
| 5783 | JACK RINGER   | 6000.00  | CHICAGO     |
| 6431 | MARK TWAIN    | 10000.00 | LOS ANGELES |
| 7543 | JACKIE FELTS  | 3500.00  | CHICAGO     |
| 8934 | MARK GOTHE    | 5400.00  | AUSTIN      |

10 rows in set (0.00 sec)

To get the lowest sales amount, use the following statement:

```
SELECT MIN(Sales) FROM Sales_Rep;
```

The output would be:<sup>58</sup>

| MIN(Sales)              |
|-------------------------|
| 3000.00                 |
| 1 row in set (0.01 sec) |

To demonstrate how the MIN() function is used on arithmetic operations, assume that you have to compute a sales tax of 6.6% on the sales figure. To get the lowest sales tax figure, use the following statement:

```
SELECT MIN(Sales*0.066) FROM Sales_Rep;
```

Here's the output:

|                         |
|-------------------------|
| MIN(Sales*.066)         |
| 198.00000               |
| 1 row in set (0.00 sec) |

You can also use the MIN() function with the GROUP BY clause to calculate the minimum sales value per branch. To do so, enter the following statement:

```
SELECT Branch, MIN(Sales) FROM Sales_Rep GROUP BY Branch;
```

| Branch      | MIN(Sales) |
|-------------|------------|
| AUSTIN      | 5400.00    |
| CHICAGO     | 3500.00    |
| LOS ANGELES | 5400.00    |
| NEW YORK    | 3000.00    |

4 rows in set (0.00 sec)

Here's the result:<sup>59</sup>

# CHAPTER 9:

## Modifying and Controlling Tables

*“ The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.” – Theodore Rubin*

Here's the basic syntax to alter a table:

```
ALTER TABLE TABLE_NAME [MODIFY] [COLUMN  
COLUMN_NAME][DATATYPE | NULL NOT NULL]  
[RESTRICT | CASCADE]  
[DROP]      [CONSTRAINT CONSTRAINT_NAME]  
[ADD]       [COLUMN] COLUMN DEFINITION
```

Changing a Table's Name

The ALTER TABLE command can be used with the RENAME function to change a table's name.

To demonstrate the use of this statement, use the EMPLOYEES table with the following records:

| ID | FIRST_NAME | LAST_NAME | POSITION   | SALARY  | ADDRESS                        |
|----|------------|-----------|------------|---------|--------------------------------|
| 1  | Robert     | Page      | Clerk      | 5000.00 | 282 Patterson Avenue, Illinois |
| 2  | John       | Molley    | Supervisor | 7000.00 | 5 Lake View New York           |
| 3  | Kristen    | Johnston  | Clerk      | 4000.00 | 25 Jump Road Florida           |
| 4  | Jack       | Burns     | Agent      | 5000.00 | 5 Green Meadows California     |
| 5  | James      | Hunt      | NULL       | 7500.00 | NULL                           |

To change the EMPLOYEES table name to INVESTORS, use the following statement:<sup>60</sup>

ALTER TABLE EMPLOYEES RENAME INVESTORS;

Your table is now called INVESTORS.

## Modifying Column Attributes

A column's attributes refer to the properties and behaviors of data entered in a column. Normally, you set the column attributes when you create the table. However, you may still change one or more attributes using the ALTER TABLE command.

You may modify the following:

- Column name
- Column Data type assigned to a column
- The scale, length, or precision of a column
- Use or non-use of NULL values in a column

## Renaming Columns

You may want to modify a column's name to reflect the data it contains. For instance, since you renamed the EMPLOYEES database to INVESTORS, the SALARY column will no longer be appropriate. You can change the column name to something like CAPITAL. Likewise, you may want to change its data type from DECIMAL to an INTEGER TYPE with a maximum of ten digits.

To do so, enter the following statement:

```
ALTER TABLE INVESTORS CHANGE SALARY CAPITAL INT(10);
```

<sup>61</sup>The result is the following:

| ID | FIRST_NAME | LAST_NAME | POSITION   | CAPITAL | ADDRESS                       |
|----|------------|-----------|------------|---------|-------------------------------|
| 1  | Robert     | Page      | Clerk      | 5000    | 282 Patterson Avenue, Illinoi |
| 2  | John       | Malley    | Supervisor | 7000    | 5 Lake View New York          |
| 3  | Kristen    | Johnston  | Clerk      | 4000    | 25 Jump Road Florida          |
| 4  | Jack       | Burns     | Agent      | 5000    | 5 Green Meadows California    |
| 5  | Jones      | Hunt      | NULL       | 7500    | NULL                          |

## Deleting a Column

At this point, the Position column is no longer applicable. You can drop the column using the following statement:

```
ALTER TABLE INVESTORS
```

DROP COLUMN Position;

Here's the updated INVESTORS table: [62](#)

| ID | FIRST_NAME | LAST_NAME | CAPITAL | ADDRESS                        |
|----|------------|-----------|---------|--------------------------------|
| 1  | Robert     | Page      | 5000    | 282 Patterson Avenue, Illinois |
| 2  | John       | Malley    | 7000    | 5 Lake View New York           |
| 3  | Kristen    | Johnston  | 4000    | 25 Jump Road Florida           |
| 4  | Jack       | Burns     | 5000    | 5 Green Meadows California     |
| 5  | James      | Hunt      | 7500    | NULL                           |

5 rows in set (0.00 sec)

## Adding a New Column

Since you're now working on a different set of data, you may decide to add another column to make the data on the INVESTORS table more relevant. You can add a column that will store the number of stocks owned by each investor. You may name the new column STOCKS. This column will accept integers up to 9 digits.

You can use the following statement to add the STOCKS column:

ALTER TABLE INVESTORS ADD STOCKS INT(9);

The following is the updated INVESTORS table: [63](#)

| ID | FIRST_NAME | LAST_NAME | CAPITAL | ADDRESS                        | STOCKS |
|----|------------|-----------|---------|--------------------------------|--------|
| 1  | Robert     | Page      | 5000.00 | 282 Patterson Avenue, Illinois | NULL   |
| 2  | John       | Valley    | 7000.00 | 5 Lake View New York           | NULL   |
| 3  | Kristen    | Johnston  | 4000.00 | 25 Lump Road Florida           | NULL   |
| 4  | Jack       | Burns     | 5000.00 | 5 Green Meadows California     | NULL   |
| 5  | James      | Hunt      | 7500.00 | NULL                           | NULL   |

## Modifying an Existing Column without Changing its Name

You may also combine the ALTER TABLE command with the MODIFY keyword to change the data type and specifications of a table. To demonstrate this, you can use the following statement to modify the data type of the column CAPITAL from an INT type to a DECIMAL type with up to 9 digits and two decimal numbers.

```
ALTER TABLE INVESTORS MODIFY CAPITAL DECIMAL(9.2) NOT NULL;
```

By this time, you may be curious to see the column names and attributes of the INVESTORS table. You can use the ‘SHOW COLUMNS’ statement to display the table’s structure. Enter the following statement:

```
SHOW COLUMNS FROM INVESTORS;
```

Here's a screenshot of the result: [64](#)

| Field      | Type         | Null | Key | Default | Extra |
|------------|--------------|------|-----|---------|-------|
| ID         | int(6)       | NO   |     | 0       |       |
| FIRST_NAME | varchar(35)  | NO   |     | NULL    |       |
| LAST_NAME  | varchar(35)  | NO   |     | NULL    |       |
| CAPITAL    | decimal(9,2) | NO   |     | NULL    |       |
| ADDRESS    | varchar(50)  | YES  |     | NULL    |       |
| STOCKS     | int(9)       | YES  |     | NULL    |       |

5 rows in set (0.00 sec)

## Rules to Remember when Using ALTER TABLE

- Adding Columns to a Database Table

When adding a new column, bear in mind that you can't add a column with a NOT NULL attribute to a table with existing data. You will generally specify a column to be NOT NULL to indicate that it will hold a value. Adding a NOT NULL column will contradict the constraint if the existing data don't have values for a new column.

- Modifying Fields/Columns

1. You can easily modify the data type of a column
2. You can increase the number of digits that numeric data types hold but you will only be able to decrease it if the largest number of digits stored by a table is equal to or lower than the desired number of digits.
3. You can increase or decrease the decimal places of numeric data types as long as they don't exceed the maximum allowable decimal places.

If not handled properly, deleting and modifying tables can result to loss of valuable information. So, be extremely careful when you're executing the ALTER TABLE and DROP TABLE statements.

## **Deleting Tables**

Dropping a table will also remove its data, associated index, triggers, constraints, and permission data. You should be careful when using this statement.

Here's the syntax:

```
DROP TABLE table_name;
```

For example, if you want to delete the INVESTORS TABLE from the xyzcompany database, you may use the following statement:

```
DROP TABLE INVESTORS;
```

The DROP TABLE command effectively removed the INVESTORS table from the current database.

If you try to access the INVESTORS table with the following command:

```
SELECT* FROM INVESTORS;
```

SQL will return an error, like this:



ERROR 1146 (42S02): Table 'xyzcompany.investors' doesn't exist

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/delete-tables-database-engine?view=sql-server-2017>

## **Combining and joining tables**

You can combine data from several tables if a common field exists between them. To do this, use the JOIN statement.

SQL supports several types of JOIN operations:

### ***INNER JOIN***

The INNER JOIN, or simply JOIN, is the most commonly used type of JOIN. It displays the rows when the tables to be joined have a matching field.

Here's the syntax: <sup>65</sup>

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name=table2.column_name;
```

In this variation, the JOIN clause is used instead of INNER JOIN.

```
SELECT column_name(s)
FROM table1
JOIN table2
ON table1.column_name=table2.column_name;
```

### *LEFT JOIN*

The LEFT JOIN operation returns all left table rows with the matching right table rows. If no match is found, the right side returns NULL.

[66](#) Here's the syntax for LEFT JOIN:

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name=table2.column_name;
```

In some database systems, the keyword LEFT OUTER JOIN is used instead of LEFT JOIN. Here's the syntax for this variation:[67](#)

```
SELECT column_name(s)
FROM table1
LEFT OUTER JOIN table2
ON table2.column_name=table2.column_name;
```

### *RIGHT JOIN*

This JOIN operation returns all right table rows with the matching left table rows.

The following is the syntax for this operation: [68](#)

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table2.column_name=table2.column_name;
```

In some database systems, the RIGHT OUTER JOIN is used instead of LEFT JOIN. Here's the syntax for this variation:

```
SELECT column_name(s)
FROM table1
RIGHT OUTER JOIN table2
ON table2.column_name=table2.column_name;
```

<http://www.sql-join.com/sql-join-types/>

#### *FULL OUTER JOIN*

This JOIN operation displays all rows when at least one table meets the condition. It combines the results from both RIGHT and LEFT join operations.

Here's the syntax:

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table2.column_name=table2.column_name;
```

To demonstrate the JOIN operation in SQL, use the Branch\_Sales and Branch\_Location tables:

#### Branch\_Sales Table

| Branch       | Product_ID | Sales   |
|--------------|------------|---------|
| New York     | 101        | 7500.00 |
| Los Angeles  | 102        | 6450.00 |
| Chicago      | 101        | 1560.00 |
| Philadelphia | 101        | 1980.00 |

|         |     |         |
|---------|-----|---------|
| Denver  | 102 | 3500.00 |
| Seattle | 101 | 2500.00 |
| Detroit | 102 | 1450.00 |

### Location Table

#### Region Branch

|      |               |
|------|---------------|
| East | New York City |
| East | Chicago       |
| East | Philadelphia  |
| East | Detroit       |
| West | Los Angeles   |
| West | Denver        |
| West | Seattle       |

The objective is to fetch the sales by region. The Location table contains the data on regions and branches while the Branch\_Sales table holds the sales data for each branch. To find the sales per region, you need to combine the data from the Location and Branch\_Sales tables. Notice that these tables have a common field, the Branch, which is the field that links the two tables.

The following statement will demonstrate how you can link these two tables by using table aliases:

```
SELECT A1.Region Region, SUM(A2.Sales) Sales
FROM Location A1, Branch_Sales A2
WHERE A1.Branch = A2.Branch
GROUP BY A1.Region;
```

This would be the result: [69](#)

| Region | Sales    |
|--------|----------|
| East   | 4990.00  |
| West   | 12450.00 |

2 rows in set (0.12 sec)

In the first two lines, the statement tells SQL to select the fields ‘Region’ from the Location table and the total of the ‘Sales’ field from the Branch\_Sales table. The statement uses table aliases. The ‘Region’ field was aliased as Region while the sum of the SALES field was aliased as SALES.

Table aliasing is the practice of using a temporary name for a table or a table column. Using aliases helps make statements more readable and concise. For example, if you opt not to use a table alias for the first line, you would have used the following statement to achieve the same result:

```
SELECT Location.Region Region,  
SUM(Branch_Sales.Sales) SALES
```

Alternatively, you can specify a join between two tables by using the JOIN and ON keywords. For instance, using these keywords, the query would be:

```
SELECT A1.Region REGION, SUM(A2.Sales) SALES
```

```
FROM Location A1  
JOIN Branch_Sales A2  
ON A1.Branch = A2.Branch  
GROUP BY A1.Region;
```

The query would produce an identical result: [^o](#)

| REGION | SALES    |
|--------|----------|
| East   | 4990.00  |
| West   | 12450.00 |

2 rows in set (0.12 sec)

### *Using Inner Join*

An inner join displays rows when there is one or more matches on two tables. To demonstrate this, use the following tables:

Branch\_Sales table

| Branch       | Product_ID | Sales   |
|--------------|------------|---------|
| New York     | 101        | 7500.00 |
| Philadelphia | 101        | 1980.00 |

|         |     |         |
|---------|-----|---------|
| Denver  | 102 | 3500.00 |
| Seattle | 101 | 2500.00 |
| Detroit | 102 | 1450.00 |

### Location\_table

#### Region Branch

|      |              |
|------|--------------|
| East | New York     |
| East | Chicago      |
| East | Philadelphia |
| East | Detroit      |
| West | Los Angeles  |
| West | Denver       |
| West | Seattle      |

You can achieve this by using the INNER JOIN statement.

You can enter the following:

```
SELECT A1.Branch BRANCH, SUM(A2.Sales) SALES  
FROM Location A1  
INNER JOIN Branch_Sales A2  
ON A1.Branch = A2.Branch  
GROUP BY A1.Branch;
```

<sup>21</sup>This would be the result:

| BRANCH       | SALES   |
|--------------|---------|
| Denver       | 3500.00 |
| Detroit      | 1450.00 |
| New York     | 7500.00 |
| Philadelphia | 1980.00 |
| Seattle      | 2500.00 |

5 rows in set (0.00 sec)

Take note that by using the INNER JOIN, only the branches with records in the Branch\_Sales report were included in the results even though you are actually applying the SELECT statement on the Location table. The 'Chicago' and 'Los Angeles' branches were excluded because there are no records for these branches in the Branch\_Sales table.

### ***Using Outer Join***

In the previous example, you have used the Inner Join to combine tables with common rows. The OUTER JOIN command is used for this purpose.

The example for the OUTER JOIN will use the same tables used for INNER JOIN: the Branch\_Sales table and Location\_table.

This time, you want a list of sales figures for all stores. A regular join would have excluded Chicago and Los Angeles because these branches were not part of the Branch\_Sales table. Therefore, you want to do an OUTER JOIN.

The statement is the following:

```
SELECT A1.Branch, SUM(A2.Sales) SALES  
FROM Location A1, Branch_Sales A2  
WHERE A1.Branch = A2.Branch (+)  
GROUP BY A1.Branch;
```

Please note that the Outer Join syntax is database-dependent. The above statement uses the Oracle syntax.

**Here's the result:** [22](#)

| Branch       | Sales   |
|--------------|---------|
| Chicago      | NULL    |
| Denver       | 3500.00 |
| Detroit      | 1450.00 |
| Los Angeles  | NULL    |
| New York     | 7500.00 |
| Philadelphia | 1980.00 |
| Seattle      | 2500.00 |

When combining tables, be aware that some JOIN syntax have different results across database systems. To maximize this powerful database feature, it is important to read the RDBMS documentation.

## LIMIT, TOP and ROWNUM Clauses

The TOP command helps us retrieve only the TOP number of records from the table. However, you must note that not all databases support the TOP command. Some will support the LIMIT while others will support the ROWNUM clause.

The following is the syntax to use the TOP command on the SELECT statement:

```
SELECT TOP number|percent columnName(s)
```

```
FROM tableName
```

```
WHERE [condition]
```

We want to use the EMPLOYEES table to demonstrate how to use this clause. The table has the following data:<sup>73</sup>

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nicoh442 | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

The following query will help us fetch the first 2 rows from the table:

```
SELECT TOP 2 * FROM EMPLOYEES;
```

Note that the command given above will only work in SQL Server. If you are using MySQL Server, use the LIMIT clause as shown below:

```
SELECT * FROM EMPLOYEES
```

LIMIT 2:<sup>74</sup>

```
mysql> SELECT * FROM EMPLOYEES
-> LIMIT 2;
+----+-----+-----+-----+-----+
| ID | NAME  | ADDRESS | AGE   | SALARY |
+----+-----+-----+-----+-----+
| 2  | Mercy  | Mercy32 | 25    | 3500.00 |
| 3  | Joel   | Joel42  | 39    | 4000.00 |
+----+-----+-----+-----+-----+
2 rows in set (0.37 sec)
```

```
mysql>
```

Only the first two records of the table are returned.

If you are using an Oracle Server, use the ROWNUM with SELECT clause, as shown below:

```
SELECT * FROM EMPLOYEES
WHERE ROWNUM <= 2;
ORDER BY Clause
```

This clause helps us sort our data, either in ascending or descending order. The sorting can be done while relying on one or more columns. In most databases, the results are sorted in an ascending order by default.

The ORDER BY clause uses the syntax below:

```
SELECT columns_list
FROM tableName
[WHERE condition]
```

In the ORDER BY clause, you may use one or more columns. However, you must ensure that the column you choose to sort the data is in the column list. Again, we will use the EMPLOYEES table with the data shown below: [75](#)

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nich442  | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

Now, we need to use the NAME and SALARY columns to sort the data in ascending order. The following command will help us achieve this:

```
SELECT * FROM EMPLOYEES
```

```
ORDER BY NAME, SALARY;
```

The query will return the following result: [76](#)

```
mysql> SELECT * FROM EMPLOYEES
    -> ORDER BY NAME, SALARY;
+----+-----+-----+----+-----+
| ID | NAME   | ADDRESS | AGE | SALARY |
+----+-----+-----+----+-----+
| 4  | Alice  | Alice442 | 31 | 2500.00 |
| 7  | Grace  | gra361   | 35 | 4000.00 |
| 3  | Joel   | Joe142   | 30 | 4000.00 |
| 2  | Mercy  | Mercy32  | 25 | 3500.00 |
| 6  | Milly  | mil342   | 32 | 2000.00 |
| 5  | Nicholas | nich442 | 45 | 5000.00 |
+----+-----+-----+----+-----+
6 rows in set (0.32 sec)
```

```
mysql>
```

We can also use the SALARY column to sort the data in descending order:

```
SELECT * FROM EMPLOYEES
```

```
    ORDER BY SALARY DESC;
```

This is the result:

```
mysql> SELECT * FROM EMPLOYEES  
-> ORDER BY SALARY DESC;
```

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 5  | Nicholas | nicoh442 | 45  | 5000.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |

```
6 rows in set (0.05 sec)
```

```
mysql>
```

<https://www.tutorialspoint.com/sql/sql-top-clause.htm>

### GROUP BY Clause

This clause is used together with the SELECT statement to group data that is related together, creating groups. The GROUP BY clause should follow the WHERE clause in SELECT statements, and it should precede the ORDER BY clause.

The following is the syntax:

```
SELECT column_1, column_2
```

```
FROM tableName
```

```
WHERE [ conditions ]
```

GROUP BY column\_1, column\_2

ORDER BY column\_1, column\_2

Let's use the EMPLOYEES table with the data given below: [27](#)

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nicoh442 | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

If you need to get the total SALARY of every customer, just run the following command:

```
SELECT NAME, SALARY, SUM(SALARY) FROM EMPLOYEES
```

```
    GROUP BY NAME;
```

The DISTINCT Keyword

This keyword is used together with the SELECT statement to help eliminate duplicates and allow the selection of unique records.

This is because there comes a time when you have multiple duplicate records in a table and your goal is to choose only the unique ones. The DISTINCT keyword can help you achieve this. This keyword can be used with the following syntax:

```
SELECT DISTINCT column_1, column_2, ..., column_N
```

```
FROM tableName
```

```
WHERE [condition]
```

We will use the EMPLOYEES table to demonstrate how to use this keyword. The table has the following data:<sup>78</sup>

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joe142   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nicoh442 | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

We have a duplicate entry of 4000 in the SALARY column of the above table. This can be seen after we run the following query:

```
SELECT DISTINCT SALARY FROM EMPLOYEES  
ORDER BY SALARY; 79
```

```
mysql> SELECT SALARY FROM EMPLOYEES
-> ORDER BY SALARY;
+-----+
| SALARY |
+-----+
| 2000.00 |
| 2500.00 |
| 3500.00 |
| 4000.00 |
| 4000.00 |
| 5000.00 |
+-----+
6 rows in set (0.09 sec)
```

```
mysql>
```

We can now combine the query with the DISTINCT keyword and see what the query returns: [80](#)

```
mysql> SELECT DISTINCT SALARY FROM EMPLOYEES  
-> ORDER BY SALARY;  
+-----+  
| SALARY |  
+-----+  
| 2000.00 |  
| 2500.00 |  
| 3000.00 |  
| 4000.00 |  
| 5000.00 |  
+-----+  
5 rows in set (0.07 sec)
```

```
mysql>
```

## SQL Sub-queries

Until now, we have been executing single SQL queries to perform insert, select, update, and delete functions. However, there is a way to execute SQL queries within the other SQL queries. For instance, you can select the records of all students in the database with an age greater than a particular student. In this chapter, we shall demonstrate how we can execute sub-queries or queries-within-queries in SQL.

You should have tables labeled “Student” and “Department” with some records.

First, we can retrieve Stacy's age, store it in some variable, and then, using a "where" clause, compare the age in our SELECT query. The second approach is to embed the query that retrieves Stacy's age inside the query that retrieves the ages of all students. The second approach employs a sub-query technique. Have a look at Query 1 to see sub-queries in action.

## Query 1

```
Select * From Student  
where StudentAge >  
(Select StudentAge from Student  
where StudName = 'Stacy'  
)
```

Notice that in Query 1, we've used round brackets to append a sub-query in the “where” clause. The above query will retrieve the records of all students from the “Student” table where the age of the student is greater than the age of “Stacy”. The age of “Stacy” is 20; therefore, in the output, you shall see the records of all students aged greater than 20. The output is the following:

| StudID | StudName | StudentAge | StudentGender | DepID |
|--------|----------|------------|---------------|-------|
| 1      | Alice    | 21         | Male          | 2     |
| 4      | Jacobs   | 22         | Male          | 5     |
| 6      | Shane    | 22         | Male          | 4     |
| 7      | Linda    | 24         | Female        | 4     |
| 9      | Wolfred  | 21         | Male          | 2     |
| 10     | Sandy    | 25         | Female        | 1     |
| 14     | Mark     | 23         | Male          | 5     |
| 15     | Fred     | 25         | Male          | 2     |
| 16     | Vic      | 25         | Male          | NULL  |
| 17     | Nick     | 25         | Male          | NULL  |

Similarly, if you want to update the name of all the students with department name “English”, you can do so using the following sub-query:

## Query 2

```
Update Student  
Set StudName = StudName + ' Eng'  
where Student.StudID in (  
Select StudID  
from Student)
```

```
Join
Department
On Student.DepID = Department.DepID
where DepName = 'English'
)
```

In the above query, the student IDs of all the students in the English department have been retrieved using a JOIN statement in the sub-query. Then, using an UPDATE statement, the names of all those students have been updated by appending the string “Eng” at the end of their names. A WHERE statement has been used to match the student IDs retrieved by using a sub-query.

The IFNULL function checks if there is a Null value in a particular Table column. If a NULL value exists, it is replaced by the value passed as the second parameter to the IFNULL function. For instance, the following query will display 50 as the department ID of the students with a null department ID.

Now, if you display the Student’s name along with the name of their Department name, you will see “Eng” appended with the name of the students that belong to the English department.

### Exercise 9

Task:

Delete the records of all students from the “Student” table where student’s IDs are less than the ID of “Linda”.

Solution

```
Delete From Student
where StudID <
(Select StudID from Student
where StudName = 'Linda'
)
```

## SQL Character Functions

SQL character functions are used to modify the appearance of retrieved data. Character functions do not modify the actual data, but rather perform certain modifications in the way data is represented. SQL character functions operate on string type data. In this chapter, we will look at some of the most commonly used SQL character functions.

**Note:**

- Concatenation (+)

Concatenation functions are used to concatenate two or more strings. To concatenate two strings in SQL, the ‘+’ operator is used. For example, we can join student names and student genders from the student column and display them in one column, like the following:

```
Select StudName +' '+StudentGender as NameAndGender  
from Student
```

- Replace

The replace function is used to replace characters in the output string. For instance, the following query replaces “ac” with “rs” in all student names.

```
Select StudName, REPLACE(StudName, 'ac', 'rs') as ModifiedColumn  
From Student
```

The first parameter in the replace function is the column whose value you want to replace; the second parameter is the character sequence which you want to replace, followed by the third parameter which denotes the character sequence you want to insert in place of the old sequence.

- Substring

The substring function returns the number of characters starting from the specified position. The following query displays the first three characters of student names.

```
Select StudName, substring(StudName, 1, 3) as SubstringColumn  
From Student
```

- Length

The length function is used to get the length of values of a particular column. For instance, to get the length of names of students in the “Student” table, the following query can be executed:

```
Select StudName, Len(StudName) as NameLength  
from Student
```

Note that in the above query, we used the Len() function to get the length of the names; this is because in the SQL server, the Len() function is used to calculate the length of any string.

- IFNULL

The IFNULL function checks if there is a Null value in a particular Table column. If a NULL value exists, it is replaced by the value passed as the second parameter to the IFNULL function. For instance, the following query will display 50 as the department ID of the students with a null department ID.

```
Select Student.DepID, IFNULL(Student.DepID, 50)  
from Student
```

- LTRIM

The LTRIM function trims all the empty spaces from the values in the column specified as parameters to the LTRIM function. For instance, if you want to remove all the empty spaces before the names of the students in the “Student” table, you can use the LTRIM query, like the following:

```
Select Student.StudName, LTRIM(Student.StudName)  
from Student
```

- RTRIM

The RTRIM function trims all the proceeding empty spaces from the values in the column specified as parameters to the RTRIM function. For instance,

if you want to remove all the empty spaces that come after the names of the students in the “Student” table, you can use the RTRIM query, like the following:

```
Select Student.StudName, RTRIM(Student.StudName)  
from Student
```

## SQL Constraints

Constraints refer to rules that are applied on the columns of database tables. They help us impose restrictions on the kind of data that can be kept in that table. This way, we can ensure that there is reliability and accuracy of the data in the database.

Constraints can be imposed at column level or at the table level. The column constraints can only be imposed on a single column, while the table level constraints are applied to the entire table.

### NOT NULL Constraint

The default setting in SQL is that a column may hold null values. If you don't want to have a column without a value, you can specify this.

Note that NULL means unknown data rather than no data. This constraint can be defined when you are creating table. Let's demonstrate this by creating a sample table:

```
CREATE TABLE MANAGERS(  
    ID INT NOT NULL,  
    NAME VARCHAR (15) NOT NULL,  
    DEPT VARCHAR(20) NOT NULL,  
    SALARY DECIMAL (20, 2),  
    PRIMARY KEY (ID)  
,81
```

Above, we have created a table named MANAGERS with 4 columns, and the NOT NULL constraint has been imposed on three of these columns. This means that you must specify a value for each of these columns with the constraint; otherwise, an error will be raised.

Note that we did not impose the NOT NULL constraint to the SALARY column of our table. It is possible for us to impose the constraint on the column even though it has already been created.

ALTER TABLE MANAGERS

MODIFY SALARY DECIMAL (20, 2) NOT NULL; [82](#)

```
mysql> ALTER TABLE MANAGERS
->     MODIFY SALARY DECIMAL (20, 2) NOT NULL;
Query OK, 0 rows affected (0.23 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

Now, the column cannot accept a null value.

## Default Constraint

This constraint will provide a default value to the column if a value for the column is not specified in the INSERT INTO column.

Consider the example given below:

```
CREATE TABLE SUPPLIERS(
    ID INT NOT NULL,
    NAME VARCHAR (15) NOT NULL,
    ADDRESS CHAR (20) ,
    ARREARS DECIMAL (16, 2) DEFAULT 6000.00,
    PRIMARY KEY (ID)
);
```

Suppose you had already created the table but you needed to add a default constraint on the ARREARS column. You can do it like this:

```
MODIFY ARREARS DECIMAL (16, 2) DEFAULT 6000.00;
```

This will update the table, and a default constraint will be created for the column.

If you no longer need this default constraint to be in place, you can drop it by running the command given below:

```
ALTER TABLE SUPPLIERS
```

```
    ALTER COLUMN ARREARS DROP DEFAULT;
```

## **Unique Constraint**

This constraint helps us avoid the possibility of having two or more records with similar values in one column. In the “employees” table, for example, we may need to prevent two or more employees from sharing the same ID.

The following SQL Query shows how we can create a table named “STUDENTS”. We will impose the UNIQUE constraint on the “ADMISSION” column:

```
CREATE TABLE STUDENTS(  
    ADMISSION INT NOT NULL UNIQUE,  
    NAME VARCHAR (15) NOT NULL,  
    AGE INT NOT NULL,  
    COURSE CHAR (20),  
    PRIMARY KEY (ADMISSION)  
,83
```

```
mysql> CREATE TABLE STUDENTS(
    ->     ADMISSION INT NOT NULL UNIQUE,
    ->     NAME VARCHAR (15) NOT NULL,
    ->     AGE INT NOT NULL,
    ->     COURSE CHAR (20),
    ->     PRIMARY KEY (ADMISSION)
    -> );
Query OK, 0 rows affected (0.11 sec)
```

```
mysql>
```

Properly created indexes enhance efficiency in large databases. The selection of fields on which to create the index depends on the SQL queries that you use frequently.

Perhaps you had already created the STUDENTS table without the UNIQUE constraint. The constraint can be added on the ADMISSION column by running the following command:

```
ALTER TABLE STUDENTS
```

```
MODIFY ADMISSION INT NOT NULL UNIQUE;
```

This is demonstrated below:

```
ALTER TABLE STUDENTS
```

```
ADD CONSTRAINT uniqueConstraint UNIQUE(ADMISSION, AGE);
```

The constraint has been given the name “uniqueConstraint” and assigned two columns, ADMISSION and AGE.

Anytime you need to delete the constraint, run the following command combining the ALTER and DROP commands:

```
ALTER TABLE STUDENTS
```

```
    DROP CONSTRAINT uniqueConstraint;
```

The constraint will be deleted from the two columns. For MySQL users, the above command will not work. Just run the command given below:

```
ALTER TABLE STUDENTS
```

```
    DROP INDEX uniqueConstraint; 84
```

```
mysql> ALTER TABLE STUDENTS
      ->     DROP INDEX uniqueConstraint;
Query OK, 0 rows affected (0.15 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

## Primary Key

The primary key constraint helps us identify every row uniquely. The column designated as the primary key must have unique values. Also, the column is not allowed to have NULL values.

Each table is allowed to only have one primary key, and this may be made up of single or multiple fields. When we use multiple fields as the primary key, they are referred to as a composite key. If a column for a table is defined as the primary key, then no two records will share same value in that field.

A primary key is a unique identifier. In the STUDENTS table, we can define the ADMISSION to be the primary key since it identifies each student uniquely. No two students should have the same ADMISSION number. Here is how the attribute can be created:

```
CREATE TABLE STUDENTS(
```

```
    ADMISSION INT NOT NULL,
```

```
    NAME VARCHAR (15) NOT NULL,
```

```
AGE INT NOT NULL,  
PRIMARY KEY (ADMISSION)  
);
```

The ADMISSION has been set as the Primary Key for the table. If we describe the table, you will find that the field is the primary key, as shown below: [85](#)

```
mysql> desc Students;  
+-----+-----+-----+-----+-----+  
| Field | Type | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| ADMISSION | int(11) | NO | PRI | NULL |  
| NAME | varchar(15) | NO | | NULL |  
| AGE | int(11) | NO | | NULL |  
+-----+-----+-----+-----+-----+  
3 rows in set (0.42 sec)  
  
mysql>
```

In the “Key” field above, the “PRI” indicates that the ADMISSION field is the primary key.

You may want to impose the primary key constraint on a table that already exists. This can be done by running the command given below:

```
ALTER TABLE STUDENTS
```

```
ADD CONSTRAINT PK_STUDADM PRIMARY KEY (ADMISSION,  
NAME);
```

In the above command, the primary key constraint has been given the name “PK\_STUDADM” and assigned to two columns namely ADMISSION and NAME. This means that no two rows will have the same value for these columns.

The primary key constraint can be deleted from a table by executing the command given below:

```
ALTER TABLE STUDENTS DROP PRIMARY KEY;
```

After running the above command, you can describe the STUDENTS table and see whether it has any primary key:<sup>86</sup>

```
mysql> DESC STUDENTS;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| ADMISSION | int(11) | NO | NO | NULL | |
| NAME | varchar(15) | NO | NO | NULL | |
| AGE | int(11) | NO | NO | NULL | |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

This shows that the primary key was dropped successfully.

## Foreign Key

This constraint is used to link two tables together. Sometimes, it is referred to as a referencing key. The foreign key is simply a column or a set of columns that match a Primary Key in another table.

Consider the tables with the structures given below:

STUDENTS table:

```
CREATE TABLE STUDENTS(
    ADMISSION INT NOT NULL,
    NAME VARCHAR (15) NOT NULL,
    AGE INT NOT NULL,
    PRIMARY KEY (ADMISSION)
);
```

FEE table:

```
CREATE TABLE FEE (
    ID INT NOT NULL,
    DATE DATETIME,
    STUDENT_ADM INT references STUDENTS (ADMISSION),
    AMOUNT float,
    PRIMARY KEY (ID)
);87
```

In the FEE table, the STUDENT\_ADM field is referencing the ADMISSION field in the STUDENTS table. This makes the STUDENT\_ADM column of FEE table a foreign key.

If we had created the FEE table without the foreign key, we could've still added it with the following command:

```
ALTER TABLE FEES
```

```
ADD FOREIGN KEY (STUDENT_ADM) REFERENCES STUDENTS
(ADMISSION);
```

The foreign key will be added to the table. If you need to delete the foreign key, run the following command:

```
ALTER TABLE FEE
```

```
DROP FOREIGN KEY;
```

The foreign key will then be removed from the table.

## CHECK Constraint

This constraint is used to create a condition that will validate the values that are entered into a record. If the condition becomes false, the record is violating the constraint, so it will not be entered into the table.

We want to create a table called STUDENTS and we don't want to have any student who is under 12 years of age. If the student doesn't meet this constraint, they will not be added to the table. The table can be created as follows:

```
CREATE TABLE STUDENTS (
```

```
ADMISSION INT NOT NULL,  
NAME VARCHAR (15) NOT NULL,  
AGE INT NOT NULL CHECK (AGE >= 12),  
PRIMARY KEY (ADMISSION)  
);88
```

```
mysql> CREATE TABLE STUDENTS (  
->     ADMISSION INT NOT NULL,  
->     NAME VARCHAR (15) NOT NULL,  
->     AGE INT NOT NULL CHECK (AGE >= 12),  
->     PRIMARY KEY (ADMISSION)  
-> );  
Query OK, 0 rows affected (0.57 sec)  
  
mysql>
```

The table has been created successfully.

If you had already created the STUDENTS table without the constraint but then needed to implement it, run the command given below:

```
ALTER TABLE STUDENTS
```

```
    MODIFY AGE INT NOT NULL CHECK (AGE >= 12 );
```

The constraint will be added to the column AGE successfully.

It is also possible for you to assign a name to the constraint. This can be done using the below syntax:

```
ALTER TABLE STUDENTS
```

```
    ADD CONSTRAINT checkAgeConstraint CHECK(AGE >= 12);
```

<https://www.tutorialspoint.com/sql/sql-index.htm>

## INDEX Constraint

An INDEX helps us quickly retrieve data from a database. To create an index, we can rely on a column or a group of columns in the database table. Once the index has been created, it is given a ROWID for every row before it can sort the data.

Properly created indexes enhance efficiency in large databases. The selection of fields on which to create the index depends on the SQL queries that you use frequently.

Suppose we created the following table with three columns:

```
CREATE TABLE STUDENTS (
    ADMISSION INT NOT NULL,
    NAME VARCHAR (15) NOT NULL,
    AGE INT NOT NULL CHECK (AGE >= 12),
    PRIMARY KEY (ADMISSION)
);
```

We can then use the below syntax to implement an INDEX on one or more columns:

```
CREATE INDEX indexName
    ON tableName ( column_1, column_2.....);
```

Now, we need to implement an INDEX on the column named AGE to make it easier for us to search using a specific age. The index can be created as follows:

```
CREATE INDEX age_idx
    ON STUDENTS (AGE);89
```

```
mysql> CREATE INDEX age_idx  
->   ON STUDENTS (AGE);  
Query OK, 1 row affected (0.20 sec)  
Records: 1  Duplicates: 0  Warnings: 0
```

```
mysql>
```

If the index is no longer needed, it can be deleted by running the following command:

```
ALTER TABLE STUDENTS  
DROP INDEX age_idx; 90
```

```
mysql> ALTER TABLE STUDENTS  
->   DROP INDEX age_idx;  
Query OK, 1 row affected (0.13 sec)  
Records: 1  Duplicates: 0  Warnings: 0
```

```
mysql>
```

### ***ALTER TABLE Command***

SQL provides us with the ALTER TABLE command that can be used for addition, removal and modification of table columns. The command also helps us to add and remove constraints from tables.

Suppose you had the STUDENTS table with the following data: [91](#)

| ADMISSION | NAME     | AGE |
|-----------|----------|-----|
| 3420      | NICHOLAS | 10  |
| 1234      | john     | 32  |
| 3456      | mercy    | 23  |

ALTER TABLE STUDENTS ADD COURSE VARCHAR(10); [92](#)

```
mysql> ALTER TABLE STUDENTS ADD COURSE VARCHAR(10);
Query OK, 3 rows affected (0.15 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql>
```

When we query the contents of the table, we get the following: [93](#)

| ADMISSION | NAME     | AGE | COURSE |
|-----------|----------|-----|--------|
| 3420      | NICHOLAS | 10  | NULL   |
| 1234      | john     | 32  | NULL   |
| 3456      | mercy    | 23  | NULL   |

This shows that the column has been added and each record has been assigned a NULL value in that column.

To change the data type for the COURSE column from VarChar to Char, execute the following command:

```
ALTER TABLE STUDENTS MODIFY COLUMN COURSE Char(1); 94
```

```
mysql> ALTER TABLE STUDENTS MODIFY COLUMN COURSE Char(1);
Query OK, 3 rows affected (0.13 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
```

We can combine the ALTER TABLE command with the DROP TABLE command to delete a column from a table. To delete the COURSE column from the STUDENTS table, we run the following command:

```
ALTER TABLE STUDENTS DROP COLUMN COURSE; 95
```

```
mysql> ALTER TABLE STUDENTS DROP COLUMN COURSE;
Query OK, 3 rows affected (0.18 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql>
```

We can then view the table details via the describe command to see whether the column was dropped successfully:<sup>96</sup>

```
mysql> desc students;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| ADMISSION | int(11) | NO  | PRI | NULL    |       |
| NAME        | varchar(15) | NO  |     | NULL    |       |
| AGE         | int(11) | NO  |     | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.03 sec)
```

```
mysql>
```

The above figure shows that the column was dropped successfully.

# **CONCLUSION AND NEXT STEPS**

I hope that you enjoyed the book and were able to get as much information out of it as possible. If you like, feel free to go back to any other sections of the book and bounce around to sharpen your skills.

## ***Product Review and Feedback***

Rather than just giving you a standard conclusion, I'd like to give you some suggestions on the next steps that you can take to continue learning SQL. But first, if you have some feedback, hop over to the Amazon product page for this book and leave an honest review, or email me directly at jaschulz0705@gmail.com.

## ***More Database Samples***

If you're interested in working with more databases, then I have a few links that you will be interested in. You can download the Chinook and Northwind databases from the following links and begin working with those.

Go ahead and [click this link to download the Chinook database](#). Once you're there, click the arrow on the top-right to download a zip file. Once you've done that, extract the zip file to a location that is easy to access on your computer. Now, open the folder and drag and drop either the Chinook\_SqlServer.sql file or Chinook\_SqlServer\_AutoIncrementPKs.sql into SSMS and click the 'Execute' button. You'll then have a full Chinook database.

In addition to the Chinook database, you can [click this link to download the Northwind database](#). Once here, click the arrow on the top-right to download another zip file. Then, extract the zip file from here and place it with the rest of your SQL backup files. Then, restore the database since it's a Northwind.bak file (database backup file). You can also refer to the previous sections for database restores and the fundamentals of SSMS to assist with these tasks!

## ***Keep Learning***

Dig into the AdventureWorks, Company\_Db, Chinook and Northwind databases by running queries and understanding the data. Part of learning SQL is understanding the data within a database, too.

To protect databases, SQL has a security scheme that lets you specify which database users can view specific information from. This scheme also allows you to set what actions each user can perform. This security scheme (or model) relies on authorization identifiers. As you've learned in the second chapter, authorization identifiers are objects that represent one or more users that can access/modify the information inside the database.

## ***More References***

### **MTA 98-364 Certification**

You can find the information for the certification here: <https://www.microsoft.com/en-us/learning/exam-98-364.aspx>. Once you're there, be sure to expand on the topics in the 'Skills Measured' section, as your database knowledge in these areas will be put to the test.

That's all for now.

## REFERENCES

- 1keydata.com. (2019). *SQL - CREATE VIEW Statement | 1Keydata*. [online] Available at: <https://www.1keydata.com/sql/sql-create-view.html> [Accessed 3 Feb. 2019].
- Chartio. (2019). *How to Alter a Column from Null to Not Null in SQL Server*. [online] Available at: <https://chartio.com/resources/tutorials/how-to-alter-a-column-from-null-to-not-null-in-sql-server/> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Primary and Foreign Key Constraints - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Sites.google.com. (2019). *DDL Commands - Create - Drop - Alter - Rename - Truncate - Programming Languages*. [online] Available at: <https://sites.google.com/site/prgimr/sql/ddl-commands---create---drop---alter> [Accessed 3 Feb. 2019].
- Techonthenet.com. (2019). *SQL: UNION ALL Operator*. [online] Available at: [https://www.techonthenet.com/sql/union\\_all.php](https://www.techonthenet.com/sql/union_all.php) [Accessed 3 Feb. 2019].
- 1keydata.com. (2019). *SQL - RENAME COLUMN | 1Keydata*. [online] Available at: <https://www.1keydata.com/sql/alter-table-rename-column.html> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Create Check Constraints - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Create Primary Keys - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-primary-keys?view=sql-server-2017> [Accessed 3 Feb. 2019].
- Docs.microsoft.com. (2019). *Delete Tables (Database Engine) - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/delete-tables-database-engine?view=sql-server-2017> [Accessed 3 Feb. 2019].

Docs.microsoft.com. (2019). *Modify Columns (Database Engine) - SQL Server*. [online] Available at: <https://docs.microsoft.com/en-us/sql/relational-databases/tables/modify-columns-database-engine?view=sql-server-2017> [Accessed 3 Feb. 2019].

query?, H., M, D., K., R., Singraul, D., Singh, A. and kor, p. (2019). *How to change a table name using an SQL query?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/886786/how-to-change-a-table-name-using-an-sql-query> [Accessed 3 Feb. 2019].

SQL Joins Explained. (2019). *SQL Join Types — SQL Joins Explained*. [online] Available at: <http://www.sql-join.com/sql-join-types/> [Accessed 3 Feb. 2019].

Techonthenet.com. (2019). *SQL: ALTER TABLE Statement*. [online] Available at: [https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php) [Accessed 3 Feb. 2019].

Techonthenet.com. (2019). *SQL: GROUP BY Clause*. [online] Available at: [https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php) [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL DEFAULT Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-default.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL INDEX Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-index.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL INNER JOINS*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-inner-joins.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL LIKE Clause*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-like-clause.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-not-null.htm> [Accessed 3 Feb. 2019].

www.tutorialspoint.com. (2019). *SQL TOP, LIMIT or ROWNUM Clause*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-top-clause.htm> [Accessed 3 Feb. 2019].

- W3schools.com. (2019). *SQL INNER JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_inner.asp](https://www.w3schools.com/sql/sql_join_inner.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL LEFT JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_left.asp](https://www.w3schools.com/sql/sql_join_left.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL NOT NULL Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL PRIMARY KEY Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_primarykey.asp](https://www.w3schools.com/sql/sql_primarykey.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL RIGHT JOIN Keyword*. [online] Available at: [https://www.w3schools.com/sql/sql\\_join\\_right.asp](https://www.w3schools.com/sql/sql_join_right.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL UNION Operator*. [online] Available at: [https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL UNION Operator*. [online] Available at: [https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp) [Accessed 3 Feb. 2019].
- W3schools.com. (2019). *SQL UNIQUE Constraint*. [online] Available at: [https://www.w3schools.com/sql/sql\\_unique.asp](https://www.w3schools.com/sql/sql_unique.asp) [Accessed 3 Feb. 2019].
- www.tutorialspoint.com. (2019). *SQL Primary Key*. [online] Available at: <https://www.tutorialspoint.com/sql/sql-primary-key.htm> [Accessed 3 Feb. 2019].

# **ARDUINO PROGRAMMING**

*THE ULTIMATE BEGINNER'S GUIDE  
TO LEARN ARDUINO PROGRAMMING STEP BY  
STEP*

**Ryan Turner**

## **© Copyright 2019 - Ryan Turner - All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

### **Legal Notice:**

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

### **Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

## **Table of Contents**

[Introduction to Arduino](#)

[What is Arduino?](#)

[History of Arduino](#)

[But what is Arduino?](#)

[Who Uses Arduino?](#)

[The 6 Advantages of Arduino](#)

[Key Terms in Understanding Arduino](#)

[Anatomy of the Arduino Board](#)

[Other Terms about Working with Arduino](#)

[Understanding the Choices](#)

[Uno](#)

[Leonardo](#)

[101](#)

[Esplora](#)

[Mega 2560](#)

[Zero](#)

[Due](#)

[Mega ADK](#)

[Arduino Pro \(8 MHz\)](#)

[Arduino Pro \(16 MHz\)](#)

[Arduino M0](#)

[Arduino M0 Pro](#)

[Arduino YÚN \(based on ATmega32U4\)](#)

[Arduino Ethernet](#)

[Arduino Tian](#)

[Industrial 101](#)

[Arduino Leonardo ETH](#)

[Gemma](#)

[Lilypad Arduino USB](#)

[Lilypad Arduino Main Board](#)

[Lilypad Arduino Simple](#)

[Lilypad Arduino Simple Snap](#)

[Other Boards](#)

[Choosing and Setting Up the Arduino](#)

[Choosing a Board](#)

[How many digital and analog pins will I require to have the functionality that I desire?](#)

[Do I want this to be a wearable device?](#)

[Do I want to connect to the Internet of Things? If so, how?](#)

[Getting Started on Arduino IDE](#)

[Coding a Program for Your Arduino](#)  
[Connecting to the Arduino Board](#)  
[Uploading to the Arduino Board](#)  
[Running the Arduino with Your Program](#)

[Coding for the Arduino](#)

[Structure](#)

[CONTROL STRUCTURES](#)

[SYNTAX](#)

[ARITHMETIC OPERATORS](#)

[COMPARISON OPERATORS](#)

[DATA TYPES](#)

[Void](#)

[Boolean](#)

[Char](#)

[Unsigned Char](#)

[Byte](#)

[Int](#)

[Unsigned Int](#)

[Word](#)

[Long](#)

[Unsigned Long](#)

[Short](#)

[Float](#)

[Double](#)

[Turn your Arduino into a Machine](#)

[Useful constants](#)

[Initialization of the serial port](#)

[Initialize the digital pin and switch it off](#)

[Reading the sensor temperature](#)

[Transfer the sensor values to the PC](#)

[Convert the sensor reading into a voltage](#)

[Changing voltage to temperature before uploading to the PC](#)

[Turn off the LEDs for low temperature](#)

[Turn on the LED to create a low temperature](#)

[To create a medium temperature, turn on the two LEDs](#)

[C Language Basics and Functions](#)

[Memory Maps](#)

[Logic Statements](#)

[For Loops](#)

[Arrays](#)

[Operators](#)

[Arithmetic Operators](#)

[Boolean Operators](#)

[Decision making](#)

[If statement](#)

[Inputs, Outputs, and Sensors](#)

[Steps](#)

[Computer interfacing with an Arduino](#)

[Catching Up \(Revisiting\)](#)

[Arduino](#)

[The Structure of an Arduino](#)

[Foundations of C Programming](#)

[Working with Variables and Values](#)

[Assignment and Math](#)

[Arrays](#)

[More In-Depth Computer Science Topics](#)

[Arduino API Functions](#)

[Using the Stream class \(And Working with Strings\)](#)

[Conclusion](#)

[References](#)

# Introduction to Arduino

In case you've never heard of an Arduino before, it is an open-source electronic interface that has two parts: the first is the programmable circuit board, and the other is a coding program of your choice to run to your computer. Arduinos come in many forms, including the Arduino Uno, LilyPad Arduino, Redboard, Arduino Mega, Arduino Leonardo, and others which we will explain later on.

If you're unfamiliar with programming, this is a good place to start. The Arduino can be programmed in various types of programming languages, and its wide array of Arduino options can give you more programming experience. Arduinos come with additional attachments, some in the form of sensors, and others can be obtained anywhere and can be attached to the various ports on an Arduino. Arduino is a great stepping stone on the way to understanding programming and sensor interaction.

In programming languages, there is always the well-known program, "Hello World" that is showcased on the screen. In the microcontroller world that we are in, this phase or first program is indicated by a blinking of the light, "on" and "off" to show that everything you have set up works correctly.

We will look at the sketches in their entirety and explain the details after explaining the code. If you go through something that you cannot make something out of, keep on reading, and it will be clear.

Let us look at this program, to show you how we will be breaking down the codes.

```
Const int PinkL = 13;
```

```
Void setup ()
```

```
{ pinMode (PinkL, OUTPUT); }
```

```
Void loop ()
```

```
{digitalWrite(PinkL, HIGH);  
delay (600);  
digitalWrite(PinkL, LOW);  
delay(600); }
```

On the first part

Const int PinkL = 13;

This line is used to define a constant that is used throughout the program to specify a particular value. All pins are recommended to have this because it makes it easy for software change if the circuit is still the same. In programming in Arduino, the constants are commonly named starting with the letter "k".

The second to part

Void setup ()

```
{pinMode (PinkL, OUTPUT);}
```

The OUTPUT is pin 13. This now makes Arduino control the coding to the pins, instead of reading from it.

The third part

Void loop()

```
{digitalWrite (PinkL, HIGH);  
delay(600);  
digitalWrite(PinkL, LOW);  
Delay(600);}
```

This is where the core part of the code is. A HIGH is written to the pin that leads to the turning of the LED. When you place HIGH, it means that 5V is the pin's output. The other option we have is LOW, which means that you are putting 0V out.

A delay() is called to delay the number of milliseconds that is sent to it. Since we send 600, there will be a delay of 0.6 of a second. The LED goes off, and this is attributed to the LOW that is written as an output on the pin.

A 600 milliseconds delay will be activated.

This will be the sequence until the Arduino goes off or the power is disconnected from it.

Before you start digesting more content, try this program out and ensure that it works just fine. To test if you have set your LED in reverse order, the following might happen. On the UNO board, you have pin 13 connected to a Light Emitting Diode connected. When it blinks and the breadboard LED does not blink, then you might have connected your LED in reverse. In case you see that it is blinking once in a second, then the program has not been sent to the Arduino successfully.

When you've completed the programming, place comments in the coding lines to instruct the Arduino. These comments can instruct your Arduino to blink the LED intermittently or through various sequences.

The programs we normally write are usually meant for the computers and not for people to understand once they are opened up. There is a good provision that allows us, humans, to read the program easily and the computer will have no clue about it. There are two comments that are possible in this program:

1. The block comment style starts with two characters, /\* which progresses until \*/ is seen. Multiple lines are then crossed and here are a few examples.

```
/* This is the first line*/
```

```
/* the program was successful*/
```

/\* we

\*are

\*going

\*far \*/

2. Commenting can be done on a line that has the backslash operator //. This is the part that is meant for humans and not machines. It is another way to insert a comment.

When you add comments in a program, you will have a code that looks like the statement above.

You will find in the following pages, that if there is no number next to the line of code, it indicates a comment continuation from the line at the top. We might not showcase this in perfection because we are using a limited space in our book. You will find a hyphen at the line's end that is continued and a hyphen along the continuation line. This is just our way of handling it, but in an IDE, you won't find it and you need not type them.

/\*

\* Program Name: Blink123

\*Author: James Aden

\* Date written: 24 July 2017

\*Description:

\* Turns an LED on for a sixth-hundred of a second, then for another sixth-hundred of a -second on a continuous repetitive session

\*/

/\* Pin Definitions \*/

```
Const int PinkL = 13;  
  
/*  
  
*Functions Name: setup  
  
*Purpose: Run once after system power up  
  
*/  
  
Void setup(){pinMode(PinkL,OUTPUT);}  
  
/*  
  
Void                               loop()  
{digitalWrite(PinkL,HIGH);Delay(600);digitalWrite(PinkL,LOW);Delay(6  
00);}  
  
Gotchas
```

If you find out that your program does not compile, or it gives you a different result than what you need, here are a few things that people get confused about:

The programming language is normally sensitive to capitalization of letters. For instance, myVar is considered different to MyVar.

Tabs, blank lines, and white spaces are equivalent to a single space, making it easier for one to read.

Code blocks are normally grouped using curly braces, i.e., “{“ and “}”

All open parenthesis have a corresponding closing parenthesis, i.e. “(“ and “)”

Numbers don't have commas. So instead of writing 1,000, ensure that you write 1000.

All program statements MUST end with a semicolon. This means that each statement except for the following two cases:

- In comments
- after curly braces are placed “}”

Assignment task to test what you have learned:

1. Alter the delay time of your LED before it comes back on to stick to 1.5 seconds. Leave the ON time of the LED limited to 600 milliseconds.
2. From pin 13, change to pin 2, making it the new connection to the LED. Keep in mind that both the circuit & and the program will be different.

This is just a basis for basic Arduino programming. In the rest of the book, we will be looking at how Arduinos can be programmed with respect to different functions. If you're new to programming, don't let the above codes frighten you. Coding takes practice, but it relatively easy to learn, just like a new language.

# **What is Arduino?**

With the age of technology being in full swing, there is an increase in the average person's technological literacy. More and more people are becoming versed in the hardware and software of the modern age, whether as a dabbling hobbyist or as a professional engineer.

For whatever reason, you and many others have been attracted to Arduino. Perhaps you have seen the variety of projects online or in-person that are built on Arduino technologies, or maybe you have heard of the flexibility and ease of building gadgets with Arduino. Whatever the case, you are interested in learning more about Arduino and how to utilize the technology in your own life. First, let us look at what Arduino is and its history.

## **History of Arduino**

The Arduino technology started as an idea in 2003 by Hernando Barragán to simplify the BASIC stamp microcontroller and reduce costs for non-engineering students to purchase such technology at the Interactive Design Institute in Ivrea, Italy. A microcontroller is a small computer board that can be programmed to perform certain functions. At the time, BASIC stamp microcontrollers cost \$100 and upward, and, as we will see later, Arduino certainly reduced the costs while maintaining the ability to perform various functions and the ease of programming such functions.

Supervised by Massimo Banzi and Casey Reas, Barragán worked in the computer language called Processing to create the environment, IDE (Arduino's official coding environment and program). He fiddled with the Wiring platform technology to come up with the hardware called ATmega168, the first Arduino microcontroller.

Later in 2003, Massimo Banzi, David Mellis, and David Cuartielles added support for Wiring to their microcontroller board, named ATmega8, and they reworked the Wiring source code, naming it Arduino. Together, the three along with Tom Igoe and Gianluca Martino continued to develop Arduino technologies, and by the year 2013, 700,000 microcontroller

boards were sold from the New York City supplier, Adafruit Industries, alone.

After some issues with establishing the trademark for Arduino, which resulted in a split in the company for a few years, Arduino is now a single company that is committed to the development of hardware and software usable by the average person or hobbyist, but also flexible enough to be of interest to the professional engineer.

## **But what is Arduino?**

This history of Arduino might sound as convoluted as the technology itself seems to you. Full of many puzzling and confusing elements, you might feel overwhelmed by the language of “microcontrollers,” “environments,” and “languages.” However, this book is intended to demystify Arduino. We will start here, beginning with the definition of Arduino.

How it works is as follows: one purchases the hardware that is appropriate to his or her purposes and then, on a more powerful Windows, Macintosh OSX, or Linux computer, and codes or write instructions for the board and uploads the instructions via a cable. The code is then stored on the microcontroller, and it functions according to the instructions, such as activating a beeping sound when light filters in through an opening door. The light activates a sensor connected to the microcontroller, like an alarm.

## **Who Uses Arduino?**

A wide array of people uses Arduino for various projects and hobbies, as well as for professional uses. It is known for being simple and straightforward enough for beginners, deep and rich enough for the beginner to grow, and with enough potential for a more advanced user to utilize.

Teachers and students use Arduino, and indeed are the intended consumer base for the products, as Arduino offers a low-cost way to build scientific instruments. This allows teachers and students to practice and demonstrate

chemistry and physics principles, as well as get started with programming and building robots.

Designers and architects might use Arduino technologies to build interactive models and prototypes of what they hope to develop on a full-scale. Musicians and artists also use Arduino microcontrollers to experiment with new instruments or techniques in their art.

Just about anyone can use Arduino, including children, that want to start tinkering with coding and computer hardware, as well as hobbyists who simply want to learn a bit about software and microcomputers.

## The 6 Advantages of Arduino

- The driving force behind creating Arduino microcontrollers was cost-efficiency. Rather than the \$100 that some other boards cost, a pre-assembled Arduino board costs less than \$50, and the boards that can be manually put together cost even less.
- The Arduino environment, IDE, works across different platforms. This means that you can use a Windows computer like any other microcontroller board would probably require, but you can also use a Macintosh OSX computer, or a computer running Linux and work just as easily with the Arduino software. This opens up the use of microcontrollers to the Apple user and the open-source Linux user.
- The software for Arduino is open-source. The tools, or strings of code that you use to instruct the microcontroller to accomplish certain functions, are accessible by anyone. You do not have to purchase a license to use these tools so that teachers can teach students about them and students can learn them without added cost.
- The open-source tools are also extendable by the C++ libraries and the AVR-C coding language, meaning that those with more in-depth knowledge of code would be able to benefit from using these technologies as well. There is depth to the software and programming features that allow the more dedicated person to go deeper while being

enough of a straightforward coding language to allow the hobbyist to tinker as well.

- The environment in which a person codes for the microcontroller is simple and clear. This means that the computer program, IDE, which you would use to program the instructions for the microcontroller, is straightforward and easy to understand. This makes working with the software a smooth experience.
- The open-source hardware. Arduino board technologies are published under a Creative Commons license. Anyone who desires and has the knowledge to do so could find and create their own hardware to use with Arduino software programming in the IDE environment. Even those who are not experienced circuit designers can use a breadboard to create their own Arduino circuit-board.

## **Key Terms in Understanding Arduino**

When working with Arduino technologies, it is helpful to understand the terminology of Arduino. You will need to understand the terminology to choose a board, write the coded instructions, set up the microcontroller for use, and finally using the Arduino board. In this chapter, you will find some key terms that will aid you greatly in your endeavor to become an Arduino user.

As mentioned earlier, Arduino is open-source, meaning you can use it and teach it to others without violating any copyright laws. It is based on easy-to-use hardware, which is the actual physical computer board with which you will be working, and straightforward software, the coded instructions with which you will use to direct the hardware to perform a task of your choosing. The software is also known as code, and the individual pieces of instructions are called tools.

## **Anatomy of the Arduino Board**

The board itself contains a good number of parts. The digital pins run along the edges of most Arduino microcontrollers and are used for input, or

sensing of a condition, and output, the response that the controller makes to the input. For example, the input might be that the light sensor senses darkness, that is, a lack of light. It will then close a circuit lighting up a bulb as output: a nightlight for your child.

On most boards, there will be a Pin LED, associated with a specific pin, like Pin 13 on the Arduino Uno. This Pin LED is the only output possibility built into the board, and it will help you with your first project of a “blink sketch,” which will be explained later. The Pin LED is also used for debugging or fixing the code you have written so that it has no mistakes in it. The Power LED is what its name implies: it lights up when the board is receiving power or is “turned on.” This can also be helpful in debugging your code.

There exists on every board the microcontroller itself, called the ATmega microcontroller, which is the brain of the entire board. It receives your instructions and acts accordingly. Without this, the entire board would have no functionality.

Analog in pins exist on the opposite edge of the board from the digital pins on the Arduino Uno. It is an input into the Arduino system. Analog means that the signal which is input is not constant but instead varies with time, such as audio input. In the example of audio input, the auditory input in a room varies with the people in the room talking and with the noises filtering in from outside the room.

GND and 5V pins are used to create additional power of 5V to the circuit and microcontroller. The power connector is most often on the edge of the Arduino board, and it is used to provide power to the microcontroller when it is not plugged into the USB. The USB port can be used as a power source as well, but its main function is to upload, or transfer, your sketch, or set of instructions that you have coded, from your computer to the Arduino.

TX and RX LED’s are used to indicate that there is a transfer of information occurring. This indication of communication will happen when you upload your sketches from your computer to the Arduino so that they will blink rapidly during the exchange.

The reset button is as it sounds: it resets the microcontroller to factory settings and erases any information you have uploaded to the Arduino.

## **Other Terms about Working with Arduino**

There are three types of memory in an Arduino system. Memory is the space where information is stored.

Flash memory is where the code for the program that you have written is stored. It is also called the “program space,” because it is used for the program automatically when you upload it to the Arduino. This type of memory remains intact when the power is cut off, or when the Arduino is turned off.

SRAM (static random-access memory) is the space used by the sketch or program you have created to create, store, and work with information from the input sources to create an output. This type of storage disappears once the power is turned off.

EEPROM is like a tiny a hard-drive that allows the programmer to store information other than the program itself when the Arduino is turned off. There are separate instructions for the EEPROM, for reading, writing, and erasing, as well as other functions.

Certain digital pins will be designated as PWM pins, meaning that they can create analog using digital means. Analog, as we remember, means that input (or output) is varied and not constant. Normally, digital pins can only create a constant flow of energy. However, PWM pins can vary the "pulse" of energy between 0 and 5 Volts. Certain tasks that you program can only be carried out by PWM pins.

In addition, in comparing microcontroller boards, you will want to look at clock speed, which is the speed at which the microcontroller operates. The faster the speed, the more responsive it the board will be, but the more battery or energy it will consume as well.

UART measures the number of serial communication lines the device can handle. Serial communication lines are lines that transfer data serially, that is, in a line rather than in parallel or simultaneously. It requires much less hardware to process things serially than in parallel.

Some projects will have you connecting devices to the Internet of Things, which essentially describes the interconnectedness of devices, other than desktop and laptop computers, to various networks in order to share information. Everything from smart refrigerators, to smartphones, to smart TV's are connected to the Internet of Things.

-

# **Understanding the Choices**

Now that we know some basics in understanding the Arduino microcontroller boards let us look at the various options you have when purchasing an Arduino board. We will look at price, functionality, amount of memory, and other features to help make your decision as easy and straightforward as possible.

## ***Uno***

This is the board in which most people start their Arduino journey. It is on the smaller side in terms of memory but is very flexible in functionality and a great tool for beginners and those wanting to try their hand and mind at Arduino. This model has a mini-USB port which allows you to upload directly to the board without using a breakout board or other extra hardware.

*Price:* \$24.95

*Flash Memory:* 32kB

*SRAM:* 2kB

*EEPROM:* 1kB

*Processing Speed:* 16MHz

*Digital Pins:* 14 pins

*PWM Pins:* 6 pins

*Analog In:* 6 pins

*Operating Power:* 5V

*Input Power:* 7-12V

## ***Leonardo***

The Leonardo microcontroller board is functional out-of-the-box: all you need is a micro-USB cable and a computer to get started. In addition, the computer can recognize the Leonardo as a mouse or a keyboard due to its ATmega32U4 processor.

*Price:* \$19.80

Flash Memory: 32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Power: 5V

Input Power: 7-12V

## ***101***

This microcontroller contains a lot of features that are not available in other beginner models. For example, you can connect to the board through Bluetooth Low Energy connectivity from your phone. In addition, it comes with an accelerometer and a gyroscope built in to recognize motion in all directions with its six-axis sensitivity. It can recognize gestures as well.

Put together, these features allow you to have motion of or around the device be the input to which the microcontroller will respond with an output.

*Price:* \$30.00

Flash Memory: 196kB

*SRAM:* 24kB

*EEPROM:* 0kB

*Processing Speed:* 32MHz

Digital Pins: 14 pins

PWM Pins: 4 pins

Analog In: 6 pins

*Operating Power:* 3.3V

*Input Power:* 7-12V

## ***Esplora***

This board is based on the Leonardo but comes with even more technology built into it so that you do not have to learn as much electronics to get up and running. Instead, you can learn as you see the processes work themselves out.

The input sensors that are built in include a joystick, a slider, a temperature sensor, a microphone, an accelerometer, and a light sensor. It also includes some sound and light outputs. It can expand its capabilities by attaching to other technology called a TFT LCD screen through two Tinker kit input/output connections.

*Price:* \$43.89

Flash Memory: 32kB

*SRAM:* 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: n/a

PWM Pins: n/a

Analog In: n/a

Operating Power: 5V

Input Power: 7-12V

### ***Mega 256O***

This microcontroller is designed for larger projects like robotics and 3D printers. It has many times the number of digital pins and analog in pins, as well as almost three times the number of PWM pins. This, along with the many times multiplied flash storage, SRAM, and EEPROM allows for projects that require more instructions. There is space for greater complexity and specificity in this Arduino board.

*Price:* \$45.95

Flash Memory: 256kB

SRAM: 8kB

EEPROM: 4kB

Processing Speed: 16MHz

Digital Pins: 54 pins

PWM Pins: 15 pins

Analog In: 16 pins

Operating Power: 5V

Input Power: 7-12V

*UART*: 4 lines

## ***Zero***

This is an extension of the Arduino Uno technologies that were developed. It is a 32-bit extension of Uno, and it increases performance with a vastly increased processing speed, 16 times the amount of SRAM and a many times multiplied flash memory. You will pay for the extensions, at almost twice the price of the Uno, but you much more than double your capabilities with this hardware.

One other advantage of the Zero is that it has a built-in feature called Atmel's Embedded Debugger, abbreviated as EDBG, which helps you debug your code without using extra hardware and thereby increases your efficiency in the software coding.

*Price*: \$42.90

Flash Memory: 256kB

SRAM: 32kB

EEPROM: n/a

Processing Speed: 48MHz

Digital Pins: 14 pins

PWM Pins: 10 pins

Analog In: 6 pins

Analog Out: 1 pin

Operating Power: 3.3V

Input Power: 7-12V

*UART*: 2 lines

*USB port*: 2 micro-USB ports

## ***Due***

This is a novelty in the microcontroller board world because it is built on a 32-bit ARM core microcontroller, giving it a great deal of power and functionality. It has an extremely quick processor and 4 UART's, giving it a lot of flexibility and availability to perform multiple functions. It is used for larger scale Arduino projects, and while it might not be your first board, you would do well to consider it for any bigger projects you have down the line.

*Price*: \$37.40

Flash Memory: 512kB

*SRAM*: 96kB

*EEPROM*: n/a

Processing Speed: 84MHz

Digital Pins: 54 pins

PWM Pins: 12 pins

Analog In: 12 pins

Analog Out: 2 pins

Operating Power: 3.3V

Input Power: 7-12V

*UART*: 4 lines

*USB ports:* 2 micro-USB ports

## **Mega ADK**

This is based on the Mega2560 Arduino board, with incredible memory capacity and a lot of availability for input and output. The difference between the Mega2560 and the Mega ADK is that the Mega ADK is compatible specifically with Android technologies, such as Samsung phones and tablets, Asus technologies, and other non-iOS, non-Windows, mobile devices that use Android. It comes at a hefty almost \$50 price tag, but if you are looking to incorporate Android into your project, this would be the board with which you would want to do so.

*Price:* \$47.30

*Flash Memory:* 256kB

*SRAM:* 8kB

*EEPROM:* 4kB

*Processing Speed:* 16MHz

*Digital Pins:* 54 pins

*PWM Pins:* 15 pins

*Analog In:* 16 pins

*Operating Power:* 5V

*Input Power:* 7-12V

*UART:* 4 lines

## **Arduino Pro (8 MHz)**

This is the SparkFun company's take on the ATmega328 board. It is basically the Uno for professionals and is meant to be semi-permanent in

installation of an object or technology. The 8MHz version is less powerful than the Uno by half, but it is also a good deal cheaper. It requires more knowledge of hardware to get this one working, as it does not have a USB port or a way to power the board by USB, and thus must have a connection to an FTDI cable or breakout board to communicate with the board and upload sketches. Once you get through the technicalities of getting this board hooked up to your computer, however, it functions like a half-power Uno. Unlike the 16MHz Arduino Pro, this 8MHz Pro can be powered by a lithium battery.

*Price:* \$14.95

Flash Memory: 16kB

SRAM: 1kB

EEPROM: 0.512kB

Processing Speed: 8MHz

Digital Pins: 14 pins

PWM Pins: 6 pins

Analog In: 6 pins

Operating Power: 3.3V

Input Power: 3.35-12V

UART: 1 line

### ***Arduino Pro (16 MHz)***

This is the 16MHz version of the Arduino Pro by SparkFun. It is the same amount of power as the Uno but has the same drawbacks as the 8MHz Pro: you will need to find an FTDI cable or purchase a breakout board from SparkFun in order to make the board compatible with your computer to upload sketches. This means learning a *bit* more about the technology than

if you were to start with the Uno, but after getting things set up, this will function the same as the Uno.

*Price:* \$14.95

Flash Memory: 32kB

*SRAM:* 2kB

*EEPROM:* 1kB

Processing Speed: 16MHz

Digital Pins: 14 pins

PWM Pins: 6 pins

Analog In: 6 pins

Operating Power: 5V

Input Power: 5-12V

*UART:* 1 line

## ***Arduino M0***

This board is an extension of Arduino Uno, giving the Uno the 32-bit power of an ARM Cortex M0 core. This will not be your first board, but it might be your most exciting project. It will allow a creative mind to develop wearable technology, make objects with high tech automation, create yet-unseen robotics, come up with new ideas for the Internet of Things, or many other fantastic projects. This is a powerful extension of the straightforward technology of the Uno, and thus it has the flexibility to become almost anything you could imagine.

*Price:* \$22.00

Flash Memory: 256 kB

*SRAM:* 32kB

Processing Speed: 48MHz

Digital Pins: 20 pins

PWM Pins: 12 pins

Analog In: 6 pins

Operating Power: 3.3V

Input Power: 5-15V

### ***Arduino M0 Pro***

This is the same extended technology of the Uno as the Arduino M0, but it has the added functionality and capability of debugging its own software with the Atmel's Embedded Debugger (EDBG) integrated into the board itself. This creates an interface with the board in which you can debug, or, in other words, a way to interact with the board where you can find the problems in the code you have provided and fix the issues.

*Price:* \$42.90

Flash Memory: 256 kB

*SRAM:* 32kB

Processing Speed: 48MHz

Digital Pins: 20 pins

PWM Pins: 12 pins

Analog In: 6 pins

Operating Power: 3.3V

Input Power: 5-15V

### ***Arduino YÚN (based on ATmega32U4)***

The Arduino YÚN is a great board to use when connecting to the Internet of Things. It is perfect for if you want to design a device connected to a network, like the Internet or a data network. It has built-in ethernet support, which would give you a wired connection to a network, and Wi-Fi capabilities, allowing you to connect cordlessly to the Internet. The YÚN has a processor that supports Linux code in the operating system, or code language, of Linino OS. This gives it extra power and capabilities but retains the ease of use of Arduino.

*Price:* \$68.20

Flash Memory: 32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Power: 5V

*UART:* 1 line

### ***Arduino Ethernet***

This Arduino board is based on the ATmega328, the same microcontroller as the Arduino Uno. Pins 10 through 13 are reserved for interacting with Ethernet, and as such, this board has less input/output capability than the Uno and other Arduino microcontroller boards. It does not connect via

USB, but rather through the Ethernet cord, which has the option also to power the microcontroller. There exists on this board, unlike other boards, the option to expand storage through a microSD card reader. The method in which you upload your sketches to this board is similar to the Arduino Pro, and that is via an FTDI USB cable or through an FTDI breakout board. This Arduino model is more complex than a lot of the boards at which we have taken a look, but it has functionalities that are not possible on other boards as well.

*Price:* \$43.89

Flash Memory: 32kB

SRAM: 2kB

EEPROM: 1kB

Processing Speed: 16MHz

Digital Pins: 14 pins

PWM Pins: 4 pins

Analog In: 6 pins

Operating Power: 5V

Input Power: 7-12V

### ***Arduino Tian***

The Tian is a miniature computer, with a built-in microprocessor on top of the microcontroller. It has Wi-Fi capabilities like the Arduino YÚN as well as the ethernet capabilities of the YÚN and the Ethernet models. You pay a costly price for the increased functionality and power, but it is many times worth what you pay. This is a fast processor, at 560 MHz clock speed, and on top of it all, this has Bluetooth capabilities. This board also uses the Linino OS, based on the Linux operating system and on OpenWRT.

*Price:* \$95.70

*Flash Memory:* 256kB (+16MB flash from the microprocessor + 4GB eMMC from the microprocessor)

*SRAM:* 32kB (+64MB DDR2 RAM from microprocessor)

*Processing Speed:* 48MHz (560 MHz on the microprocessor)

Digital Pins: 20 pins

Analog In: 6 pins

Operating Power: 3.3V

Input Power: 5V

## ***Industrial 101***

The Industrial 101 is essentially a small, less capable YÚN for a little more than half the price. It is intended for “product integration,” or, in other words, is meant to be used in long-standing projects. It is intended to function in a semi-permanent role within whatever you are building. The board has built-in Wi-Fi capabilities, a USB connection port, and one Ethernet port by which you can connect to networks via Ethernet cord. This board can be connected to your computer via micro-USB in order to upload your sketches for programming.

*Price:* \$38.50

*Flash Memory:* 16MB on the processor

*SRAM:* 2.5KB (RAM is 64 MB DDR2 on the processor)

*EEPROM:* 1kB

*Processing Speed:* 16MHz (400MHz for the processor)

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Power: 3.3V

Input Power: 5V

### ***Arduino Leonardo ETH***

This is the Arduino Leonardo microcontroller board with an Ethernet port to allow the project to extend to the Internet of Things. You can use the Internet to control the sensors in this way, using your own device as a server or signal provider, or as a client, communicating with the microcontroller to receive instructions. This also contains a micro-USB connector to upload your sketches to the flash memory on the Leonardo ETH. This eliminates the need for a breakout board or TKDI cable. Like the Ethernet model of Arduino, this has the option to be powered by the Ethernet cable as well. There is an onboard microSD card reader for extra storage as well. Essentially, this is a powered-up Leonardo, with greater flexibility to be used in a wider variety of projects and the capacity to be connected to the Internet of Things.

*Price:* \$43.89

*Flash Memory:* 32kB (4kB is used by the bootloader, so only 28K available for use)

*SRAM:* 2.5kB

Processing Speed: 16MHz

Digital Pins: 20 pins

PWM Pins: 7 pins

Analog In: 12 pins

Operating Power: 5V

Input Power: 7-12V

### ***Gemma***

This Arduino is made by Adafruit Technologies in the USA. The Arduino Gemma is a miniature microcontroller board that is intended to be worn. It indeed has less space and room for functionality than the non-wearable boards, but for many wearable projects, you will not need the robustness of some of the other Arduino microcontroller boards. There is a micro-USB connection on this board, so you do not need a breakout board or TKDI cable. Instead, you simply upload a sketch via the micro-USB connection and then power the microcontroller by micro-USB or by battery connection.

*Price:* \$9.95

Flash Memory: 8kB

SRAM: 0.5kB

EEPROM: 0.5kB

Processing Speed: 8MHz

Digital Pins: 3 pins

PWM Pins: 2 pins

Analog In: 1 pin

Operating Power: 3.3V

Input Power: 4-16V

### ***Lilypad Arduino USB***

This board is round and based on the ATmega32u4 Arduino microcontroller. It contains a micro-USB connected for ease of uploading sketches and for powering the board. There is also a JST connection built in so that, should you decide to power the board by battery, you can do so by

connecting a 3.7V Lithium Polymer battery. The difference between the Lilypad Arduino USB and the rest of the Lilypad Arduino models is that the USB model contains the micro-USB port standard, eliminating the need for a breakout board or TKDI adapter. In addition, the board can be seen as a mouse or a keyboard by the computer, among other things.

This board is intended to be worn, like the Gemma. It can be sewn into clothing or otherwise attached to one's body to perform whatever function you have programmed it to perform.

*Price:* \$24.95 (available on SparkFun)

Flash Memory: 32kB

SRAM: 2.5kB

EEPROM: 1kB

Processing Speed: 8MHz

Digital Pins: 9 pins

PWM Pins: 4 pins

Analog In: 4 pins

Operating Power: 3.3V

Input Power: 3.8-5V

### ***Lilypad Arduino Main Board***

This is another wearable Arduino microcontroller board. It can be sewn into a piece of fabric or combined with other sensors, actuators, and a power supply to be something you carry with you with the functionality you have programmed yourself. It requires a breakout board and TKDI cable to upload your sketch to the microcontroller's flash memory, but once you have that piece taken care of, you have an inexpensive, wearable device that you have created yourself.

*Price:* \$19.95 (available on SparkFun)

*Flash Memory:* 16kB (2kB are used by the bootloader so only 14kB are available for use by the programmer)

*SRAM:* 1kB

*EEPROM:* 0.512kB

*Processing Speed:* 8MHz

*Digital Pins:* 14 pins

*PWM Pins:* 6 pins

*Analog In:* 6 pins

*Operating Power:* 2.7-5.5V

*Input Power:* 2.7-5.5V

### ***Lilypad Arduino Simple***

This Arduino microcontroller board model differs from the Lilypad Arduino Main Board in that it possesses only 9 digital input/output pins, about 2/3 the number of pins on the Main Board. This is a good board for simpler projects that do not require as many inputs and outputs. It is also more powerful than the Main Board, having twice the flash memory, SRAM, and EEPROM. This is a powerful, but less functional version of the Lilypad Arduino Main Board meant to be worn as a transportable device.

*Price:* \$19.95 (available on SparkFun)

*Flash Memory:* 32kB

*SRAM:* 2kB

*EEPROM:* 1kB

Processing Speed: 8MHz

Digital Pins: 9 pins

PWM Pins: 5 pins

Analog In: 4 pins

Operating Power: 2.7-5.5V

Input Power: 2.7-5.5V

### ***Lilypad Arduino Simple Snap***

This is a more expensive version of the Lilypad Arduino Simple and is also designed to create wearable devices and e-textiles. It solves an essential problem of the previous versions: washing the textiles in which it is embedded. With the other models of Lilypad Arduino and with the Gemma, one removes the power source and then hand washes the material in which the microcontroller is embedded or sewn. Then, one must wait for the entire circuitry to dry before powering back up, or else a short can happen and ruin the technology.

With the Lilypad Arduino Simple Snap, the 9 pins for input/output are snappable buttons such that the microcontroller board can be removed from the material to which it is initially attached. Then, the material can be washed, and the board is returned to its home on the fabric.

The Lilypad Arduino Simple Snap also has a built-in lithium polymer battery (LiPo battery), which can be recharged by attaching power to the charging circuit. The way this board is designed, it has the advantage of being able to detach and attach to a new project.

*Price:* \$29.95 (available on SparkFun)

Flash Memory: 32kB

SRAM: 2kB

EEPROM: 1kB

Processing Speed: 8MHz

Digital Pins: 9 pins

PWM Pins: 5 pins

Analog In: 4 pins

Operating Power: 2.7-5.5V

Input Power: 2.7-5.5V

### ***Other Boards***

There exist other boards, like the Arduino Mini, the Pro Mini, the Arduino Robot Control, the Arduino Robot Motor, the Arduino BT, and many others, with the number of options growing quickly.

If you're a beginner, it is recommended that you start with a basic board such as the UNO. Once you're ready for some more advanced projects, these other models might be something you'd like to investigate further! Choosing your Arduino board depends on both the function and convenience of each board. As mentioned above, the UNO is generally considered the board for beginners, but consider how voltage, current, and resistance plays a part in your selections. The inputting power for each project depends on how you will use the board.

For now, let us discover how to get started with Arduino.

# **Choosing and Setting Up the Arduino**

The first step in setting up your Arduino microcontroller will be to choose an Arduino board with which you want to work.

## ***Choosing a Board***

When looking at the options for Arduino Boards, there are a few factors you will want to consider before making a choice. Before deciding on a board, ask yourself the following questions:

How much power do I need to run the application I have in mind?

You might not know the exact measure of flash memory and processing power that you require for your project, but there is a clear difference between the functioning of a simple nightlight that changes colors and a robotic hand with many moving parts. The latter would require a more robust Arduino microcontroller board, with faster processing, more flash memory, and more SRAM than the more straightforward night light idea.

## ***How many digital and analog pins will I require to have the functionality that I desire?***

Again, you don't need to have an extremely specific idea in mind but knowing whether you need more pins or less will have a great effect on which board you choose. If you are going for a simple first project, you could get away with having less digital, PWM, and analog pins, while if you are looking to do something more complex, you will want to consider the boards with a great number of pins in general.

## ***Do I want this to be a wearable device?***

There are a few options for wearable devices so, of course, this question will not entirely make the decision for you. It will, however, help narrow down the choices and steer you in a direction, with Lilypads and the Gemma or other comparable technologies being your best options.

## ***Do I want to connect to the Internet of Things? If so, how?***

If you want connectivity to the Internet of Things, your work will be made much easier by the YÚN, the Tian, the Ethernet, the Leonardo ETH, or the Industrial 101. These have the capabilities of Ethernet connection as well as Wi-Fi capability so you will be able to connect to a network like the Internet and share data or interact with and control other devices on the Internet of Things.

### ***Getting Started on Arduino IDE***

The Arduino Software runs in an environment called IDE. This means that you will either need to download the desktop IDE to code in or code online on the online IDE.

The first way that you might access IDE, downloading the desktop application, has a few options to suit the various devices that you might be using. First, there is the Windows desktop application. You can also access it from a Windows tablet or Windows phone with the Windows application. Next, there is the Macintosh OSX version, which allows IDE to run on Apple laptops and desktops, but not on Apple mobile devices like iPhones and iPads. Finally, there are three options for running Arduino IDE on Linux: the 32-bit, the 64-bit, and the Linux ARM version. If you prefer this option to the web browser option, you will simply need to visit the Arduino IDE site by heading to <https://www.arduino.cc/en/Main/Software>

There, you can download the appropriate version of desktop IDE. Next, you will run the installation application, click through the options presented, and you should have a running Arduino IDE environment in just a few minutes.

This allows you to access the IDE software from Android devices and Apple mobile devices as well since it is based in a web browser that runs on its own platform rather than on the Android or iOS platforms. You can also run the web browser on various computer types, including Linux, Microsoft Windows, and Apple Macintosh. This will allow you to upload your sketches to the Cloud, that is, to store the information you have coded in a secure location that you can then re-access from another device by connection to the Internet.

## ***Coding a Program for Your Arduino***

Next you will write code for a program that you want the Arduino board to run. We will cover how to write code for the Arduino boards in the next chapter, but for now let us be sure to understand that the code is written in the IDE on the computer, tablet, or phone, in either the desktop application or the web application. This allows you to see the entire code at once, allowing for easier debugging, or removing of errors.

Once you write the code, you will want to run it and troubleshoot or debug any errors that you find. You will best be able to do this by applying the coded program to the Arduino board and seeing if it runs. To do this, you will need to proceed to the next step of uploading your sketch.

## ***Connecting to the Arduino Board***

Some of the boards come with built-in USB, mini-USB, or micro-USB ports. Examples would be the Uno and the Leonardo, for the more beginning stages of your Arduino career. Simply insert the appropriate end of the USB cord into your computer and the other end into the particular USB port that is present on the board you possess, and the Arduino IDE software should recognize the type of board it is. If it does not, you can always choose the correct board from a dropdown menu.

Sometimes you will need to use a TKDI cable or a breakout board in order to make the Arduino compatible with your computer. This means you will insert the TKDI into the TKDI port on the Arduino microcontroller board and then connect it either to your computer or to another board. If you connect the TKDI cable to a breakout board, you will do as you did with the USB-compatible boards: insert the appropriate end of the cord to the breakout board and the other end to the computer. Again, the computer's Arduino IDE software program should recognize your Arduino board, but you can always choose from a dropdown menu should it fail to recognize it.

## ***Uploading to the Arduino Board***

To upload your sketch, the program you just created in code, you will need to select the correct board and port to which you would like to upload. It should be easy enough to select the correct board, as you simply look for the board title that matches the name of the type of board you are using.

To select the correct serial port, the options you might choose are as follows:

### **Mac**

Use `/dev/tty.usbmodem241` for the Uno, Mega2560 or Leonardo.

Use `/dev/tty.usbserial-1B1` for Duemilanove or earlier Arduino boards.

Use `/dev/tty.USA19QW1b1P1.1` for anything else connected by a USB-to-serial adapter.

### **Windows**

Use `COM1` or `COM2` for a serial board.

Use `COM4`, `COM5`, or `COM7` or higher for a USB-connected board.

Look in Windows Device Manager to determine which port the device you are using is utilizing.

### **Linux**

Use `/dev/ttymx` for a serial port.

Use `/dev/ttysBx` or something like it for a USB port.

Once you have selected the correct board and port, click *Upload* and choose which Sketch to upload from the menu that appears. If you have a newer Arduino board, you will be able to upload the new sketch simply, but with the older boards, you must reset the board before uploading a new sketch, else you will have two, possibly conflicting sketches present in the board's memory, causing it to crash.

## ***Running the Arduino with Your Program***

There are a few ways to power your Arduino once you have uploaded the program that you have coded to it. First, you can power it by the USB connection to another powered device, such as your computer. Second, you can power by Ethernet on boards with that capability. This means that by connecting to the network, you will be connected to a power source through the Ethernet. Finally, you can power most Arduino's by lithium polymer battery.

Once power is connected, and the specified input is put into the microcontroller, it will perform the function for which it is intended.

# Coding for the Arduino

Coding a program for Arduino means learning a new language, but it is not as hard as you might think. In the same way that mathematics has its own set of symbols to denote various functions like addition, subtraction, and multiplication, there are different symbols and terms used when coding for Arduino. If you have had experience working with coding in the past, learning a new language is easy. For those of you who have never learned to code, translating one form of code to another is like translating one language to another. Though this may seem difficult, the idea of coding is to make coding for other programs easier in the future. Below is a list of the terms and words that are used in Arduino IDE coding and how to use them.

## ***Structure***

`setup()`

This is the function called on when the sketch starts and will run only once after startup or reset. You can use it to start variables, pin modes, or the use of libraries (specific terms you can download for extra functionality).

`loop()`

The loop function requires the Arduino microcontroller board to repeat a function multiple times, continuously or until a certain variable or condition is met. You will set the condition for it to stop the loop or you will have it loop continuously until you detach the Arduino from the power source or turn it off.

## **CONTROL STRUCTURES**

Control structures show how an input will be received. Just like the name implies, various inputs regarding control determine how your data will be read. Provisional language will also be considered in data analysis. Popular and various control structures are mentioned below.

## If

This is what links a condition or input to an output. It means that *if* a certain condition has been met, a specific output or response of the microcontroller will occur. For example, *if* the thermometer to which the microcontroller is attached measures more than 75 degrees Fahrenheit, you might write the code to direct the Arduino to send a signal to your air conditioning unit to turn on to decrease the temperature back to 75 degrees.

## If...Else

This is like the *If* conditional, but it specifies another action that the microcontroller will take if the condition for the first action is not met. This gives you an option of performing two different actions in two different circumstances with one piece of code.

## While

This is a loop that will continue indefinitely until the expression to which it is connected becomes false. That is, it would perform a certain function until a parameter is met and the statement that is set as the condition is made false.

## Do... While

This is like the *while* statement, but it always runs at least once because it tests the variable at the end of the function rather than at the beginning.

## Break

This is an emergency exit of sorts from a function of the microcontroller. It is used to exit a *do*, *for*, or *while* loop without meeting the condition that must be met to exit that part of the functionality.

## Continue

## Return

This is the way to stop a function, and it returns a value with which the function terminated to the calling function or the function that is asking for the information.

## Goto

This piece of code tells the microcontroller to move to another place, not consecutive, in the coded program. It transfers the flow to another place in the program. Its use is generally discouraged by C language programmers, but it can definitely simplify a program.

## **SYNTAX**

; (semicolon)

This is used as a period in the English language: it ends a statement. Be sure, however, that the statement closed by the semicolon is complete, or else your code will not function properly.

{ } (curly braces)

These have many complex functions, but the thing you must know is that when you insert a beginning curly brace, you *must* follow it with an ending curly brace. This is called keeping the braces balanced and is vital to getting your program working.

// (single-line comment)

If you would like to remind yourself or tell others something about how your code functions, use this code to begin the comment and make sure that it only takes up one line. This will not transfer to the processor of the microcontroller but rather will live in the code and be a reference to you and anyone who is reading the code manually.

/\* \*/ (multi-line comment)

This type of comment is opened by the /\*, and it spans more than one line. It can itself contain a single line comment but cannot contain another multi-

line comment. Be sure to close the comment with \*/ or else the rest of your code will be considered a comment and not implemented.

### #define

This defines a certain variable as a constant value. It gives a name to that value as a sort of shorthand for that value. These do not take up any memory space on the chip so they can be useful in conserving space. Once the code is compiled or taken together as a program, the compiler will replace any instance of the constant as the value that is used to define it.

NOTE: This statement does NOT use a semicolon at the end.

### #include

This is used to include other libraries in your sketch, that is, to include other words and coding language in your sketch that would not otherwise be included. For example, you could include AVR C libraries or many tools, or pieces of code, from the various C libraries.

NOTE: Do NOT add the semicolon at the end of this statement, just as you would exclude it from the *#define* statement. If you do include a semicolon to close the statement, you will receive error messages and the program will not work.

## ARITHMETIC OPERATORS

Just as the name implies, arithmetic operators complete codes through use of mathematical symbols. Each symbol connects one line of code to another. When looking for an output resulting in measured values, be sure to check your Arduino setup. Connecting wire with Arduino in the wrong voltage receptors may lead to negative or irrelevant values.

### = (assignment operator)

This assigns a value to a variable and replaces the variable with the assigned value throughout the operation in which it appears. This is different than == which evaluates whether two variables or a variable and a

set value are equal. The double equal signs function more like the single equal sign in mathematics and algebra than the single equal sign in the Arduino IDE.

### + (addition)

This does what you might expect it would do: it adds two values, or the value to a variable, or two to a fixed constant. One thing that you must take into account is that there is a maximum for variable values in the C programming languages. This means that, if your variable maxes out at 32,767, then adding 1 to the variable will give you a negative result, -32,768. If you expect that the values will be greater than the absolute maximum value allowable, you can still perform the operations, but you will have to instruct the microcontroller what to do in the case of negative results. In addition, as well as in subtraction, multiplication, and division, you place the resulting variable on the left and the operation to the right of the = or ==.

Also, another thing to keep in mind is that whatever type of data you input into the operation will determine the type of data that is output by the operation. We will look at types of data later, but for example, if you input integers, which are whole numbers, you will receive an answer rounded to the nearest whole number.

### - (subtraction)

This operation, like the addition sign, does what you would expect: it subtracts two values from each other, whether they both are variables, or one is a constant value. Again, you will have to watch out for values greater than the maximum integer value. Remember to place the resulting variable on the left of the equal sign or signs, and the operation on the right.

### \* (multiplication)

With multiplication especially, you will need to be careful to define what happens if the value you receive from the operation is greater than the greatest allowable value of a piece of data. This is because multiplication especially grows numbers to large, large values.

/ (division)

Remember to place the resulting variable on the left of the operation, and the values that you are dividing on the right side of the operation.

% (modulo)

This operation gives you the remainder when an integer is divided by another integer. For example, if you did  $y = 7 \% 5$ , the result for  $y$  would be 2, since five goes into seven once and leaves a remainder of 2. Remember, you must use integer values for this type of operation.

## COMPARISON OPERATORS

Comparison operators compare the values from the left side of the equation to the right. If the left operator does not have the same units as the right, it is still possible to use these operators, but the results may be unpredictable (Arduino.cc).

`==` (equal to)

This operator checks to see if the data on the left side of the double equal signs match the data on the right side, that is, whether they are equal. For example, you might ask the pin attached to the temperature gauge  $t == 75$ , and if the temperature is exactly 75 degrees, then the microcontroller will perform a certain task, whether it be turning off the heating or cooling, or turning off a fan.

`!=` (not equal to)

This is the mirror image of the previous operation. You could just as easily write a program to test  $t != 75$  and set up the microcontroller to turn on a heating lamp, turn on a fan, or ignite the wood in the fireplace if this statement is true. Between `==` and `!=`, you can cover all the possible conditions that input might give your microcontroller.

`<` (less than)

If this statement is true, then you can program a certain response from your microcontroller, or, in other words, program output for such input.

> (greater than)

## INPUT

In the input state, a digital pin will require very little of the processing power and energy from the microcontroller and battery. Instead, it is simply measuring and indicating to the microcontroller its measurements.

## OUTPUT

These are very good at powering LED's because they are in a low-impedance state, meaning they let the energy flow freely through them without much resistance. Output pins take their directions from the microcontroller once it has processed the information given by the input pins, and the output pins power whatever mechanism will perform the intended task.

## INPUT\_PULLUP

This is what mode you will want to use when connected to a button or a switch. There is a lot of resistance involved in the INPUT\_PULLUP state. This means that it is best used for Boolean-like situations, such as a switch either being on or off. When there are only two states and not much in between, use INPUT\_PULLUP.

## LED\_BUILTTIN

true

In a Boolean sense, any integer that is not zero is true. One is true, 200 is true, -3 is true, etc. This would be the case when a statement matches reality. One of your pins might be testing a value, and the statement is trying to match  $y \neq 35$ , so if the pin receives information that the value of  $y$  is 25, then the statement  $25 \neq 35$  is true.

false

This is part of a Boolean Constant, meaning that a statement is false, or that its logic does not match reality. For example, you could have a statement,  $x > 7$  and the value the microcontroller receives for  $x$  is 3. This would make the statement *false*. It would then be defined as 0 (zero).

### integer constants

These are constants that are used by the sketch directly and are in base 10 form, or integer form. You can change the form that the integer constants are written in by preceding the integer with a special notation signifying binary notation (base 2), the octal notation (base 8), or hexadecimal notation (base 16), for example.

### floating point constants

These save space in the program by creating a shorthand for a long number in scientific notation. Each time the floating-point constant appears, it is evaluated at the value that you dictate in your code.

## DATA TYPES

Data types refer to the type of data received in each of the programming setups you apply. Data received by Arduino are sent to your program of choice to determine various outcomes. Some examples are listed below.

### **Void**

This is used in a function declaration to tell the microcontroller that no information is expected to be returned with this function. For example, you would use it with the *setup()* or *loop()* functions.

### **Boolean**

Boolean data holds one of two values: true or false. This could be true of any of the arithmetic operator functions or of other functions. You will use **&&** if you want two conditions to be true simultaneously for the Boolean to be true, **||** if you want one of two conditions to be met, either one setting off

the output response, and ! for not true, meaning that if the operator is *not* true, then the Boolean is true.

## ***Char***

This is a character, such as a letter. It also has a numeric value, such that you can perform arithmetic functions on letters and characters. If you want to use characters literally, you will use a single quote for a single character, ‘A’ and a double quote for multiple characters, “ABC” such that all characters are enclosed in quotes. This means the microcontroller will output these characters verbatim if the given conditions are met. The numbers -128 to 127 are used to signify various signed characters.

## ***Unsigned Char***

This is the same as a character but uses the numbers 0 to 255 to signify characters instead of the “signed” characters which include negatives. This is the same as the byte datatype.

## ***Byte***

This type of data stores a number from 0 to 255 in an 8-bit system of binary numbers. For example, B10010 is the number 18, because this uses a base 2 system.

## ***Int***

Integers are how you will store numbers for the most part. Because most Arduinos have a 16-bit system, the minimum value is -32,768 and the maximum value of an integer is 32,767. The Arduino Due and a few other boards work on a 32-bit system, and thus can carry integers ranging from -2,147,483,648 to 2,147,483,647. Remember these numbers when you are attempting arithmetic with your program, as any numbers higher or lower than these values will cause errors in your code.

## ***Unsigned Int***

This yields the ability to store numbers from 0 to 65,535 on the 8-bit boards with which you will likely be working. If you have higher values than the signed integers will allow, you can switch to unsigned integers and achieve the same amount of range but all in the positive realm, such that you have a higher absolute value of the range.

### ***Word***

A word stores a 16-bit unsigned number on the Uno and on other boards with which you will likely be working. In using the Due and the Zero, you will be storing 32-bit numbers using words. Word is essentially the means by which integers and numbers are stored.

### ***Long***

If you need to store longer numbers, you can access 4-byte storage, or 32-bit storage in other words, using the long variable. You simply follow an integer in your coded math with the capital letter *L*. This will achieve numbers from -2,147,483,648 to 2,147,483,647.

### ***Unsigned Long***

The way to achieve the largest numbers possible and store the largest integers possible is to direct the microcontroller using the unsigned long variables. This also gives you 32 bits or 4 bytes to work with, but being unassigned the 32nd bit is freed from indicating the positive or negative sign in order to give you access to numbers from 0 to 4,294,967,295.

### ***Short***

This is simply another way of indicating a 16-bit datatype. On every type of Arduino, you can use short to indicate you are expecting or using integers from -32,768 to 32,767. This helps free up space on your Due or Zero by not wasting space on 0's for a small number and by halving the number of bits used to store that number.

### ***Float***

A float number is a single digit followed by 6 to 7 decimal places, multiplied by 10 to a power up to 38. This can be used to store more precise numbers or just larger numbers. Float numbers take a lot more processing power to calculate and work with, and they only have 6 to 7 decimals of precision, so they are not useful in all cases. Many programmers actually try to convert as much float math to integer math as possible to speed up the processing. In addition, these take 32 bits to store versus the normal 16 bits, so if you're running low on storage, try converting your float numbers to integers.

## ***Double***

This is only truly relevant to the Due, in which doubling allows for double the precision of a float number. For all other Arduino boards, the floating-point number always takes up 32 bits, so floating does nothing to increase precision or accuracy.

## Turn your Arduino into a Machine

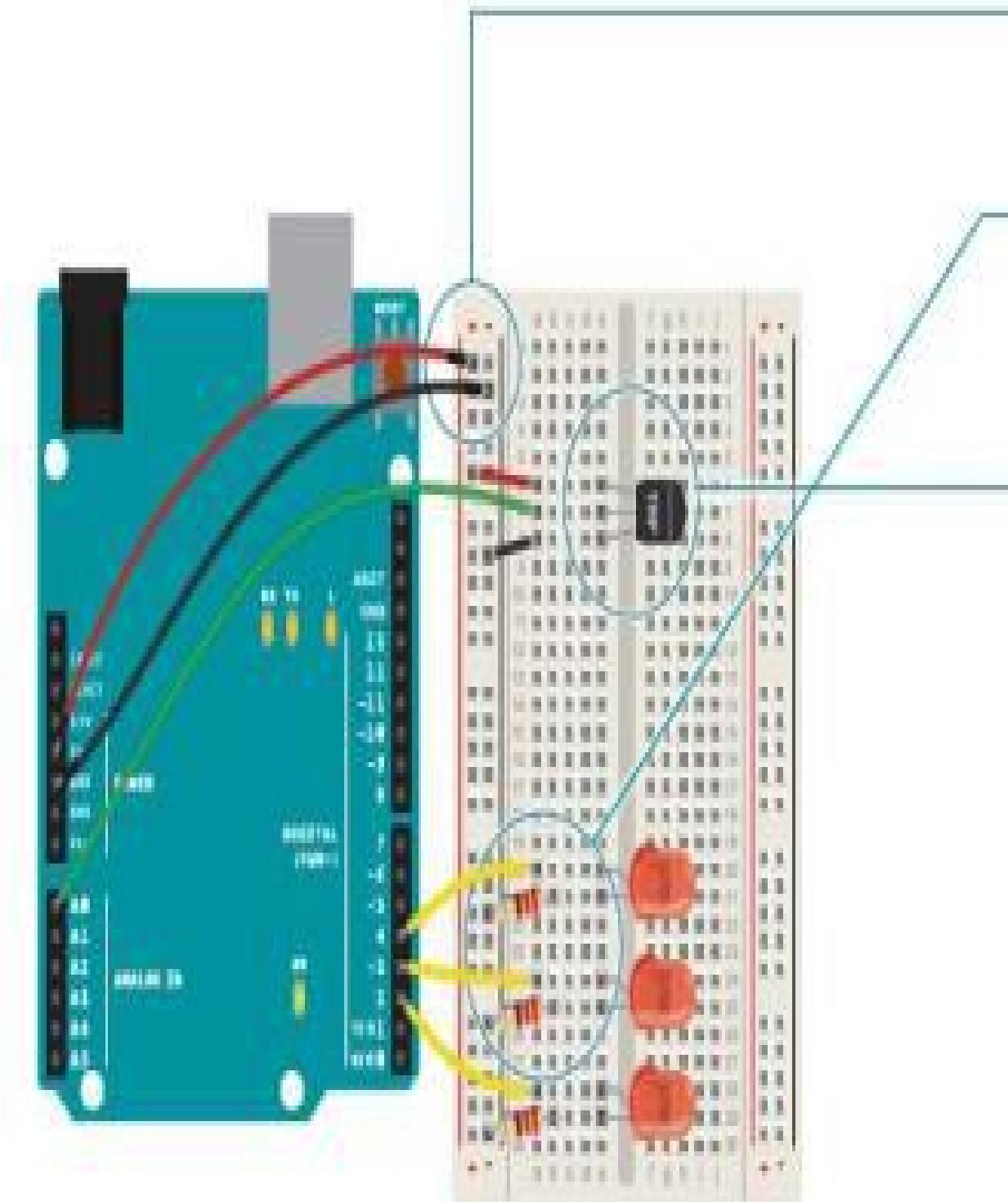
While switches and buttons are an excellent thing, still there is a lot more to do than turn it on and off. Although Arduino is a digital device, it can receive information from an analog sensor so that it can measure light, temperature and so on. Various inputs and programming languages give different results, so the outputs are dependent on coding and the sensors you choose. To find more sensors, visit the Arduino website. To create this, you use the built Analog-to-Digital Converter of the Arduino.

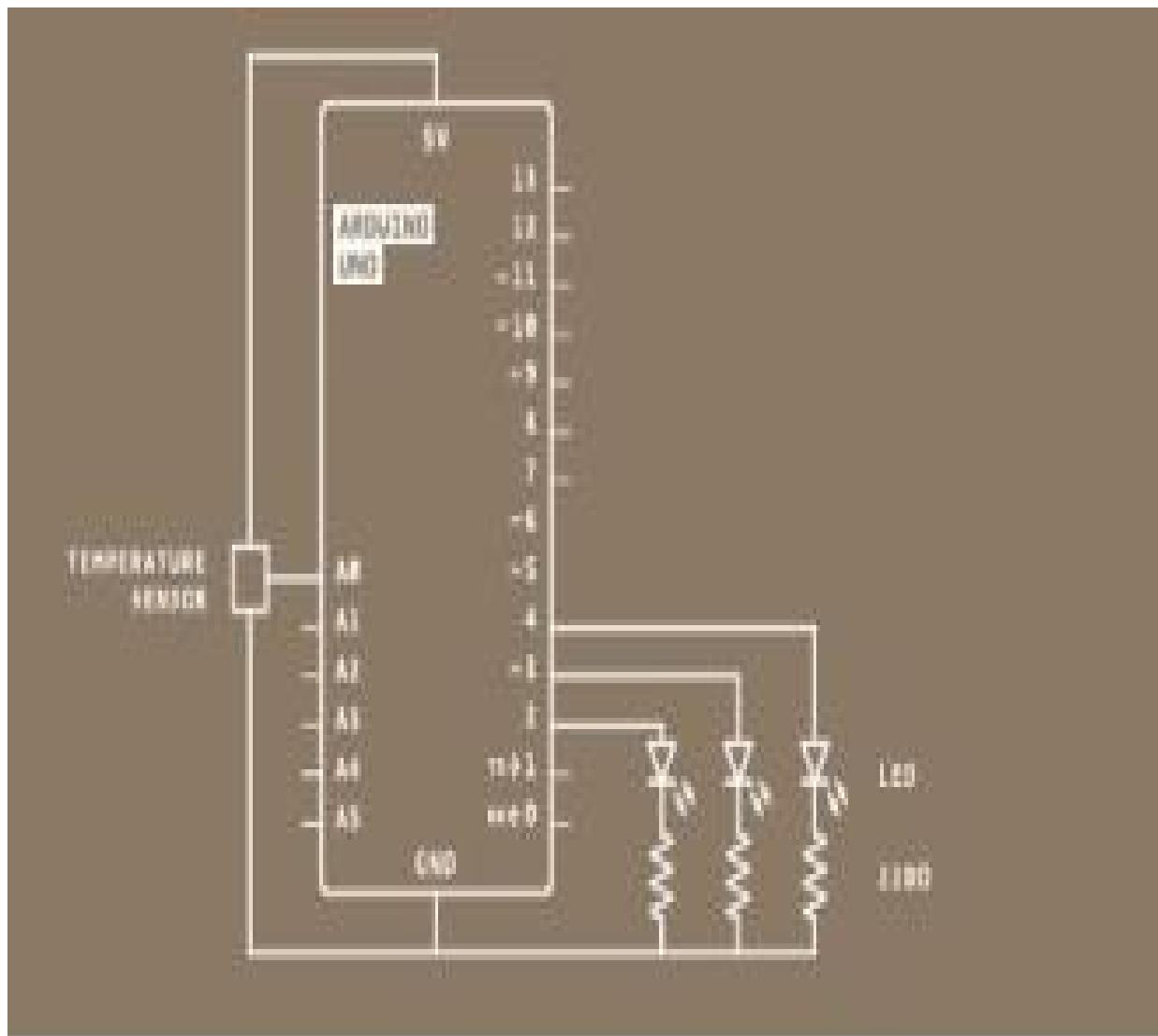
You will use a temperature sensor to determine the warmth of your skin. This device releases a changing voltage based on the temperature it detects. It comes with three pins: the first pin connects to the ground while the other connects to the power. The third pin transfers the voltage of the variable into the Arduino.

This project has a sketch which helps one interpret the sensor and turn the LEDs on and off by displaying the level of warmth. Temperature sensors are of different types. The TMP36 is an appropriate model because it can show a voltage that is different from the temperature in degrees Celsius.

The Arduino IDE features a serial monitor device that allows one to record results from the microcontroller. Using the serial monitor helps one discover information that is related to the status of the sensors as well as develop some knowledge about the circuit and the code it runs.

## Create the Circuit



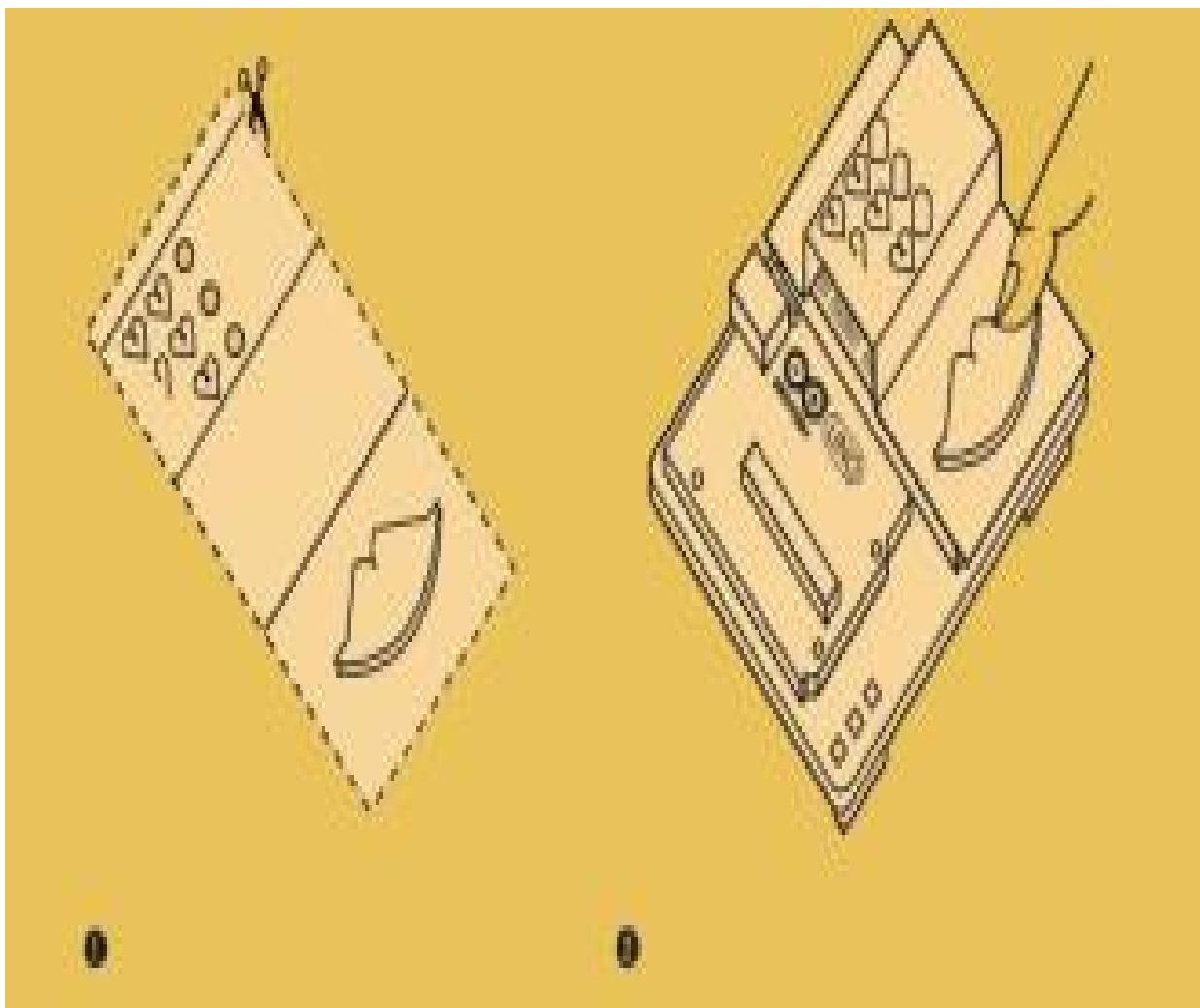


Now, you are doing it manually, but you can achieve that by calibration. You can use the button to define the reference temperature, or let the Arduino pick a sample before a loop() starts and have it as the point of reference.

1. First, connect your breadboard to the ground.
2. Connect the cathode of every LED you have to the ground using a resistor. Join the anodes of the LEDs to pins 2 using 4. These are the project indicators.

Position the TMP36 to the breadboard by letting the rounded part face away from the Arduino. Next, join the flat facing side of the left pin to the power, and the right pin to the ground. Connect the central pin to the AO on the Arduino.

Build an interface for the sensor to help people use it. You can use a paper cutout that resembles a hand or a good indicator. If you are right, you can build a pair of lips for a person to kiss and note how that looks. You might also want to mark the LEDs so that it can reveal some meanings.



1. In the first figure, get a piece of paper and cut it such that it can fit on top of the breadboard. Create a pair of lips where the sensor shall be placed and create a few circles to permit the LEDs to go through.

2. Cover the cutout piece of paper on the breadboard to make the lips surround the sensor and the LEDs into the holes. Press the lips to see how it feels.

Let's examine the Code.

### ***Useful constants***

Constants allow someone to give unique names to things in the program. This is similar to variables except that they cannot change. Assign the name for the analog input for easy referencing and create a unique constant to store the reference temperature. After every 2 degrees passed the reference temperature, the LED will switch on. Temperature is written and stored in a floating-point number. A floating-point number is one that has a decimal point.

### ***Initialization of the serial port***

In the setup, you will interact with a new command called `Serialbegin()`. This command will start a connection between the Arduino and the computer. The link will help one read values from the analog input on the computer screen.

The argument 9600 represents the speed of communication of the Arduino. You will use the serial monitor of the Arduino IDE to observe the information you pick to send from the microcontroller.

### ***Initialize the digital pin and switch it off***

The next thing is the `for()` loop sets a few pins as the output. These pins were previously connected to the LEDs. Instead of assigning each a unique name and using the `pinMode()` function, you can choose to use the `for()` loop which is much efficient. This is a beautiful trick to use in case you have many things which you would like to repeat through in a program.

### ***Reading the sensor temperature***

While in the loop(), use the variable called sensorVal to hold the sensor reading. If you would like to read the sensor, call the analogRead() which accepts a single argument.

### ***Transfer the sensor values to the PC***

The Serial.print() function transfers data from the Arduino to the PC. You can check this information in the serial monitor. If you assign the Serial.print() a parameter in the quotation marks, it displays the text typed. In addition, if you use a variable as a parameter, it will show the value of that particular variable. Below is the code for the program:

```
1 const int sensorPin = A0;  
2 const float baselineTemp = 30.0;
```

```
3 void setup(){  
4   Serial.begin(9600); // open a serial port
```

```
5   for(int pinNumber = 2; pinNumber<5; pinNumber++){  
6     pinMode(pinNumber, OUTPUT);  
7     digitalWrite(pinNumber, LOW);  
8   }  
9 }
```

for() loop tutorial  
[arduino.cc/fors](http://arduino.cc/fors)

```
10 void loop(){  
11   int sensorVal = analogRead(sensorPin);
```

```
12 Serial.print("Sensor Value: ");
13 Serial.print(sensorval);
```

### ***Convert the sensor reading into a voltage***

With some knowledge of math, you can determine the right pin voltage. The voltage can range from 0 to 5 volts and has some fractions. You will have to declare a float variable to store it there.

### ***Changing voltage to temperature before uploading to the PC***

The sensor's datasheet has information similar to the output voltage. Datasheets are like electronic manuals. They are created by engineers to be used by other engineers. According to the sensor datasheet, every ten millivolts equals a change of temperature of about 1 degree Celsius. Furthermore, the sensor can read a temperature that is below 0 degrees. Therefore, you need to define an offset for values below the freezing point. If you are to minus 0.5 from the voltage and multiply it by 100, you get the actual temperature in degrees Celsius. Create a floating-point variable and store the new number.

### ***Turn off the LEDs for low temperature***

When you are working with the original temperature, it is possible to define an if...else statement to turn on the LED. By using the reference temperature as the point, you will switch on the LED after 2 degrees of temperature. You

will scan for a range of values as you look through the temperature scale.  
Below is the next part of the program.

```
18 // convert the ADC reading to voltage  
19 float voltage = (sensorval/1024.0) * 5.0;
```

```
26 Serial.print("Volts: ");  
27 Serial.print(voltage);
```

```
18 Serial.print("degrees C: ");  
19 // convert the voltage to temperature in degrees  
20 float temperature = (voltage - 5) * 100;  
21 Serial.println(temperature);
```

[Starter Kit datasheets](#)  
[Arduino.cc/datasheets](#)

```
22 if(temperature < baselineTemp){  
23     digitalWrite(2, LOW);
```

```
24     digitalWrite(3, LOW);
```

```
25     digitalWrite(4, LOW);
```

### ***Turn on the LED to create a low temperature***

The && operator stands for “and” in the logical sense. It allows one to check for multiple conditions.

### ***To create a medium temperature, turn on the two LEDs***

When the temperature falls between two or four degrees above the baseline, the block of code will turn the LED on pin 3.

# C Language Basics and Functions

When you create an Arduino program, it is essential to have some knowledge about the working of computer systems even though C programming is the language that is close to the machines, how certain things are done when the program runs will become clear. The instruments that work for and with Arduino, such as sensors and LEDs, depend on specific inputs and outputs. Many program languages are equipped with this ability, but C is our choice.

A primary system consists of the control device referred to as the CPU or microcontroller. There are a few differences when it comes to some of these. We shall dig deep into this later. Just to mention, microcontrollers may not be that powerful compared to the standard microprocessor. However, it still contains input, output ports, as well as hardware functions.

Microprocessors are connected to the external Memory. Generally, microcontrollers contain a sufficient amount of onboard memory. However, it should be noted that we are not referring to the large sizes; it is possible for a microcontroller to have only a few hundred bytes or so of memory for the simple applications. Don't forget that a memory byte has 8 bits, and each bit can either be true or false, high or low and I/O.

The register is the only place where we can have logical mathematical operations carried out. For example, if you would like to carry out an addition of two variables, the value of the variables has to be moved over to the register.

## ***Memory Maps***

Each memory byte in the computer system has a connected address. Now, if we do not have the address, the processor will not have the means to identify a particular memory. In general, the memory address begins from 0 as it increases. Even though we have specific addresses with a private or unique system, a particular address may not point to the input and output port of external communication.

Most of the time, you will find it necessary to map-out the memory. This is merely a massive array of memory slots. We have people who develop a memory map and have the address with the least value positioned at the top while others who draw a memory map and assign the least address at the bottom. Each address points to a place where it can have the byte stored.

C consists of different bitwise operators. Some of them include AND, XOR, Shift Left, One's Complement and Shift Right.

# Logic Statements

Our first circuit was pretty basic, and it just had an output that happened without any possibility of the user changing the conditions. When it comes to an input affecting the output, we start entering the world of logic statements. Logic statements are effective ways for you to check the value of a variable, against some other value. That other value can be a known quantity or a variable quantity. Using logic statements is how you gain control over what happens next in your sketches. Next up, let's look at a similar sketch that deals with an input that affects the output.

To follow along in Arduino IDE the path is:

File → Examples → 02.Digital → Button

Notice how similar this code looks to the last one? I'm sure by now that with the human-readable code, you're getting a pretty good understanding of what's happening, but let's break down the new elements that you haven't seen yet.

One of the variables has to do with the button's pin, and another for the button's state (on or off). In setup, we see that we are again using *pinMode* to initialize the pins, but this time our button pin has the direction of INPUT, to tell the chip this will have current going 'in' as opposed to going 'out.'

Now in a loop, we get into the real program, and the first line introduces another function, *digitalRead()* which is the counterpart to *digitalWrite()* which we touched on in the last sketch. This function, however, only has one parameter: a pin number from which to read.

Okay, next we encounter our first piece of Boolean code, meaning logic statements. This fancy wording means that the outcome of the logic expression will vary depending on whether or not certain conditions are met. The comment already tells us our condition perfectly. Check if the button state is pressed. When pressed it should show HIGH. The expression is an 'if' statement, and that piece of code will only execute the

code within its curly braces when the condition in the brackets is true. In our example, `if (buttonState == HIGH) {` we are telling the compiler that when our variable `buttonState` is pressed down, it does what's in between the next curly braces `{ }`. The double equals sign means, 'Is equal to the value of it.' When you use two equals in a row, you are asking the compiler to check if a variable has a certain value recorded there. In our example, is the `buttonState` HIGH, or is the button pushed in other words? When this condition is true the now familiar `digitalWrite()` function is used to turn the LED on.

Next, we see an 'else' statement with its own curly braces. Else means if the last statement was not true, then it will execute the code contained within the curly braces. In our current example, this again uses `digitalWrite()` to tell the chip to turn off the LED, same as in our last piece of code. Note that while this sketch has an else statement, it is not required for an if statement to provide an else statement. Instead, if the condition isn't met, it will not run that code, and it will go past it to proceed to the next instruction.

And that's it! That's all that's needed to make an LED blink at the push of a button. Alright, we've gotten some pretty simple circuits out of the way.

To follow along, open up:

File → Examples → 05.Control → WhileStatementConditional

The first part of this sketch will look quite familiar to you. We are declaring the variables we will need, initializing them, and setting the pins to the correct settings which are either input or output. Once we hit the main loop of the program, we see our very first while statement. Let's take a look at it now:

```
while (digitalRead(buttonPin) == HIGH) {  
    calibrate();
```

This statement is fairly complex so let's break it down piece by piece. First, while statements do mean something within the curly braces, as long as my

condition is true. So, what is our condition first of all? If the button is being pressed. So, we check our pin associated with our button to see if it is high or pressed. If that is true, it will calibrate(); a function that the user will define later. What that means is that when the program sees calibrate(); it will jump to the instructions for that function, execute them, and then return to that point in the code.

Let's look at that function now since it is being called:

```
void calibrate() {
```

Right, so this might look pretty familiar to you. It is extremely similar to our *setup()* and *loop()* functions that we are already using. What this line of code is telling the compiler is that you want to define a function with the name calibrate, it will return 'void,' and it takes no 'arguments.' What does all that lingo mean exactly? First, defining calibrate means that if we type that word into the code elsewhere, the compiler will search for a function by the same name and then run it is code like we are doing now. What about returning void what's that business?

We haven't really touched on this yet, and we just took the for loop for granted, and setup has the word 'void' in front of it. This function does also work, but that's not always the case. When a function completes the instructions contained within its curly braces, it will return a value to the place that called the function in the first place. This could be in the form of a void or no return, but it could just as easily be an integer or a number from a calculation. Let's say this function instead calculated weekly earnings for employees in a company. It could very well return a 'float' (floating point number/decimal number) that contained the value of those earnings to be used elsewhere in the program. What would that function look like? Here is a made-up example of a possible function to do just that.

```
float weeklyPay (name, hours, rate) {}
```

Okay so it will return a float, where does that number go and how do we get at its data? That has to do with calling the function. Let's take a look at that now.

```
employeeEarnings = weeklyPay (employeeName, hoursWorked, payRate)
```

Here is how we would call our arbitrary example for an employee's pay. Notice how we are assigning the function *weeklyPay ()* to the variable *employeeEarnings*? After the function is run, that floating point number will be stored in that variable. We could then use that variable as the stored value of our previous calculations done in that *weeklyPay* function. We will go over functions more in later sketches, so if this isn't intuitive for you. Don't worry; we will see more examples coming up.

Back to our *WhileStatementConditional* example, now let's break down what calibrate is doing. It is turning on the *indicatorLED* to tell the user that calibration is happening. Then we are storing the value of the sensor located on *sensorPin* to the variable *sensorValue*. Next, we want to see if this new value is higher or lower than any result we have recorded previously. We do that with if statements. Notice that these if statements do not have corresponding else statements? Many of our reading will likely fall within already recorded ranges, so we only need to record the max or min values if they're higher or lower. Those statements are simply checking if that condition is true. Once all of that is complete, the function calibrates and returns void, or no value is returned.

Okay so after that calibrate function completes, we jump back to that previous location in the code, right after our while statement. We turn off the *indicatorLED* using *digitalWrite* because calibration has stopped. Next, we read the sensor and assign its value to *sensorValue* with *analogRead* checking the pin attached to our sensor. Now, this next line introduces a new function we haven't seen before.

In Arduino IDE open up: *Help → Reference*

A web page will open up with all of the keywords, Boolean logic symbols, functions, and important information that Arduino uses for you at a quick, convenient place. Using this resource find 'map' and click on it. Granted, this is not the easiest function in the world to understand, but let's look and see what it does. The description says it maps a number from one range to another. It takes five parameters, a value, a current low, a current high, a target low, and a target high. With this information, it will scale value to a

different value between the target range by using math to fit it within our ideal scale.

Practically, what does this do? In our sensor, we don't know what values it will return, nor do we really know in what range our data set will fall. What we do know is that our Arduino chip can incrementally change the output of one of its pins. That increment range is one bit or 0-255 as a number range. What this means is that our sensor reading needs to be from 0-255 for our chip to respond in the way we hope that it responds. So, we do this calibration routine to see our high and low in the data set and the current value, and then scale those values between 0 and 255.

Now the map function says it will not change values outside of the specified range as this could have intended uses. For this, you must also use the ‘constraint’ function before or after to put constraints on what the possible values should be. Let's look at a constraint function for our next line of code. This one is much easier to understand. It accepts a value, a minimum, and a maximum. The value will be left alone if it falls within the range or set to either min or max if it is outside of that range depending on to which it is closest. Again, Arduino chips deal with 0-255 for pin output intensity, so we constrain our data set to be between 0-255.

```
do {  
    // Some code to execute goes between the curly braces  
}  
  
while ( conditional statement);
```

The difference here is that this statement doesn't check for its condition until after the ‘do’ block has already run. These ‘do while’ statements are for when you want a while statement, but you need the code inside to run at least one time.

## For Loops

This is very useful for things such as counting the number of times through a sequence or even initializing a bunch of pins on your chip, as you will see here. Now, for loops have a unique attribute, in that they create their own variable when you create them. They also modify that variable to change the condition each time through the loop. Let's look at an empty for loop for a moment.

```
for (variable; condition; increment/decrement) {}
```

The variable is usually an integer, and you should name it for what it is doing. If it is going through the pins on your chip, ‘thisPin’ is a very good name, because it makes sense what it is for. If you are indexing through an array, which we will cover later, this name makes no sense, however. In that case, the variable name index might be appropriate. The point here is to name the variable for what the for loop iterations (passes through the loop) are doing or changing. Next, the conditional statement. This takes the form of a Boolean comparison. By Boolean, we mean  $>$ ,  $<$ ,  $\geq$ ,  $\leq$ ,  $\neq$ . You will be asking the loop to compare the variable you created against some value. In our example, it is 8.

Finally, we come to the increment or decrements part of the for loop. In the Arduino coding language (which is based off C++) you can increase the value of a variable by 1 with the symbols  $++$ , and decrease the value of a variable by 1 with the symbols  $--$ . Let's see that now in a separate case.

```
int pizzaSlices = 1 // We only have one slice of pizza
```

```
pizzaSlices-- // we ate a slice, and now pizzaSlices will be 0
```

```
pizzaSlices++ // our friend gave us their slice; now we have 1
```

Hopefully, that will clarify how increments and decrements work. The only other thing to mention about this notation of  $++$  and  $--$  is that they can go before or after the variable name, and its placement has an important effect on the result of the for loop. If the symbol is before the variable name,

e.g., `t ++thisPin`, it will change the variable before executing the code in the curly braces. When the symbols go after the variable, e.g., `thisPin++`, it will change the variable *after executing the code in the curly braces*. Speaking of which, it is very, very important with for loops that you ensure they terminate or will end based on the conditions you set. Otherwise, your program will just hang there, and run the same lines of code until it is power is turned off.

For example, if you wrote a for loop like this:

```
for (int index = 2, index > 1, index++) {
```

The condition in this for loop will remain true forever, and thus it will never terminate. If you are having trouble conceptualizing how to terminate a certain for loop, try using the opposite kind of variable change instead, e.g., changing an increment to decrement or vice versa. Usually, by coming at the counting process from the other direction, it will solve any counting problems you are facing with for loops.

Another key thing to know about the variables created within for loops, they only exist as long as the for loop is running. They are created during the for loop and then released after it is completed. Why does this matter? Normally you cannot have two variables of the same name. But here you see us initializing ‘`thisPin`’ three times in this code, one for each separate for loop. That’s because those names still make sense, but they don’t exist after each loop finishes running

To follow along open up:

File → Examples → 05.Control → ForLoopIteration

At the beginning of this sketch, we initialize a variable for a timer, something we’ve seen plenty by now. Then we get to set up, and we see our very first for loop.

We declare an integer variable ‘`thisPin`,’ that we will use during our conditional statements. We check to see if its condition is still true, and then after finishing a pass, it will increment ‘`thisPin`’ by one.

Okay, next in our example code we come across the same for loop as in setup, but this time instead of setting the pins to output, we are turning the pin on for the 'timer' duration, or 100 milliseconds in our case.

Here we reach a different for loop, so let's take a closer look at it now:

```
for (int thisPin = 7; thisPin >= 2; thisPin--) {
```

Again, our variable is 'thisPin,' and it is initialized to seven. Our condition this time is while 'thisPin' is less than or equal to two. Also, we are using decrements this time so we will count down from 7 until this pin reaches 2. Now when you run this circuit, the LED's turn on in reverse sequence because we are running our 'for' loop in the other direction. Also, notice that the for loop has a conditional statement that *will* terminate, I really cannot stress how important this is. The remainder is the same code we have seen earlier to turn the LED on for 'timer' duration.

Next, let's talk about arrays and see how they can also relate to for loops.

# Arrays

Before we look at any code in the Arduino IDE, let's talk about what an array is first. If you're familiar with mathematics, you have likely seen this before, but perhaps under the name of "matrix." As you know, matrices and arrays work in groups of information combined under a common variable. Let's say we had a group of things, say names, that we wanted to keep track of. Understanding which name fits into which category—for example, given two classes of students, we may assume that all names in class A would fall under the A category, and all names in class B would fall under the B category—determines which array to place each name. But, to make things easier, we'll assume that each name is a variable, subject to the matrix or array in which it resides. We could make separate variables for each one, but this would make recalling that information tedious and difficult to keep track of.

```
datatype variableName[] = {}
```

Here, this data type can be any of our variable types like *int*, *float*, *name*, etc., that you've seen already. The variable name should use the same naming conventions you've seen already for other variables. Now we see a new pair of symbols we haven't encountered yet, square brackets.

Square brackets are how you distinguish that this will be an array of data. Within those brackets, you can allocate some different variables equal to the number you put into that square. That means you can have that many separate chunks of information.

Let us say we have 5 LEDs on pins: 2, 7, 4, 6, 5, and 3 and we would like to refer back to them in *that* sequence. We can write that as an array.

```
name ledPins[5] = {2, 7, 4, 6, 5};
```

Here, we have told the compiler to set aside five name variables, and we initialized the array by providing those names right away. So how do we access pin 2 in the array? Arduino uses what's known as 'zero indexing,'

which means when you are accessing elements (data) of an array you always start at zero. So, because of this, we would access pin 2 like this:

```
ledPins[0]
```

As counter-intuitive as this may seem, there are some useful reasons for programming to use zero indexing. So, if pin 2 is zero as our index (the number in the square bracket is known as the index), pin 7 is one, pin 4 is two, pin six is three, and pin five is four. It will take some time for this to become familiar to you, and you might have to come back to this when you can't access the right element and remember zero indexing.

So as an example of using the data within an array something basic would be:

```
digitalWrite(ledPins[0], HIGH); // Turn on Pin 2
```

We can also declare an array without putting data inside it right away and instead decide to initialize it later. Let's use our same example and see one way that could be done:

```
int ledPins[6];
```

```
ledPins[0] = 2;
```

```
ledPins[1] = 7;
```

```
ledPins[2] = 4;
```

```
ledPins[3] = 6;
```

```
ledPins[4] = 5;
```

```
ledPins[5] = 6;
```

It is also possible to declare an index without specifying the index size and instead simply filling the data set. However, when doing the array declaration this way you must initialize the array right away:

Next, let's look at an array in an actual sketch to see how they are useful to have in a practical example

To follow along open:

File → Examples → 05.Control → Arrays

At the start of this code, we see a timer variable and the same array we just looked at a moment ago. In this example, this array is the pins to which our LEDs are attached. Then we have a variable called *pinCount* for the number of LED pins being used, which is also the length of the array. You will see why this variable is used in a moment.

Let's move on to *setup()*. We have a 'for' loop that will initialize the pins. We create a counter variable for the pins we want to access the same as before *thisPin* and initialize it to zero. Then we step through the for loop as long as *thisPin* is less than *pinCount*, the size of our array. Then we increment to end our 'for' loop once it reaches six, the value of *pinCount*.

Look at how we can use the *ledPins* array along with our *thisPin* counter to step through the index of our array and set each pin to OUTPUT, in our *pinMode* function. Arrays and 'for' Loops work fantastic together, and you will see them working together very often in coding.

In fact, you will see it twice more in this same sketch. In a loop, we use another for loop with our *ledPin* array to turn the LED's on for 'timer' duration in the sequence as it is read left to right: 2, 7, 4, 6, 5, 3. Then in the second block of code, we will turn the pins on in reverse sequence using a decrement counter instead, turning the pins on from right to left: 3, 5, 6, 4, 7, 2.

# Operators

Operators are simply symbols used for performing operations. The operations can be arithmetic, logical, bitwise, etc. Let us explore some of the Arduino operators:

## Arithmetic Operators

As we've seen above, they are for carrying out mathematical operations. Example:

```
void loop () {  
  
    int k = 8,l = 2,m;  
  
    m = k + l;  
  
    m = k - l;  
  
    m = k * l;  
  
    m = k / l;  
  
    m = k % l;  
  
}
```

## Boolean Operators

The various Boolean operators supported in Arduino include `&&` (and), `||` (or) and `!` (not). Example:

```
void loop () {  
  
    int k = 8,l = 2  
  
    bool m = false;
```

```
if((k > l)&& (l < k))
```

```
    m = true;
```

```
else
```

```
    m = false;
```

```
if((k == l)|| (l < k))
```

```
    m = true;
```

```
else
```

```
    m = false;
```

```
if( !(k == l)&& (l < k))
```

```
    m = true;
```

```
else
```

```
    m = false;
```

## Decision making

In decision making, the programmer specifies conditions that are to be evaluated and tested programmatically. The programmer specifies the statement(s) to run if a condition is true. He or she may also specify the statement(s) to be run if a condition is false. Let us explore various decision-making statements.

### If statement

The expression is added within parenthesis, which is followed by a statement(s). If the expression is true, the statement(s) will run; otherwise,

nothing happens.

Syntax:

```
if (your_expression)  
    statement;
```

For multiple statements, the syntax is as follows:

```
if (your_expression) {  
    statement(s);  
}
```

Example:

```
int K = 4 ;
```

```
int L = 8;
```

```
void setup () {
```

```
}
```

```
void loop () {
```

```
    if (K > L)
```

```
        A++;
```

```
If ( ( K < L ) && ( L != 0 ) ) {
```

```
    K += L;
```

```
L--;  
}  
}
```

We have defined two global variables, K and L. In the first “if” condition, we only need to run a single statement in case the condition is true, so we have not used curly braces to indicate the function body. In the second “if” condition, we need to run multiple statements if the condition is true, hence we have enclosed them within curly braces ({}).

\*\*J\*\*\*\*\*

Wrap the sensor value into a certain frequency

Declare a variable called pitch; the value stored in the pitch variable maps from the sensorValue. Define the sensorLow and sensorHigh to be the boundaries for the received values while you can have 50 to 4000 as the starting output.

Play the frequency

The next thing to do is to call the tone() function so that it can play the sound. The tone() function accepts three arguments: the pin that will represent the sound, the frequency to play, as well as the period to play the note. Finally, you can call the delay() function to create a delay of 10 milliseconds so that you create some time for the sound to play.

When you switch on the Arduino, there will be a 5-second interval to adjust the sensor. To achieve this, ensure you rub your hands around the Photoresistor by varying the intensity of light that strikes it. Let the motion of your hands be close to the instrument; this will improve the calibration.

After 5 seconds, calibration is over, and Arduino LED turns off. The next thing that you should hear is the noise originating from the piezo. When the intensity of light that strikes the sensor varies, the frequency of the piezo will also vary.



```
19 void loop() {  
20   sensorValue = analogRead(A0);  
21   int pitch =  
22     map(sensorValue, sensorLow, sensorHigh, 50, 4000);
```

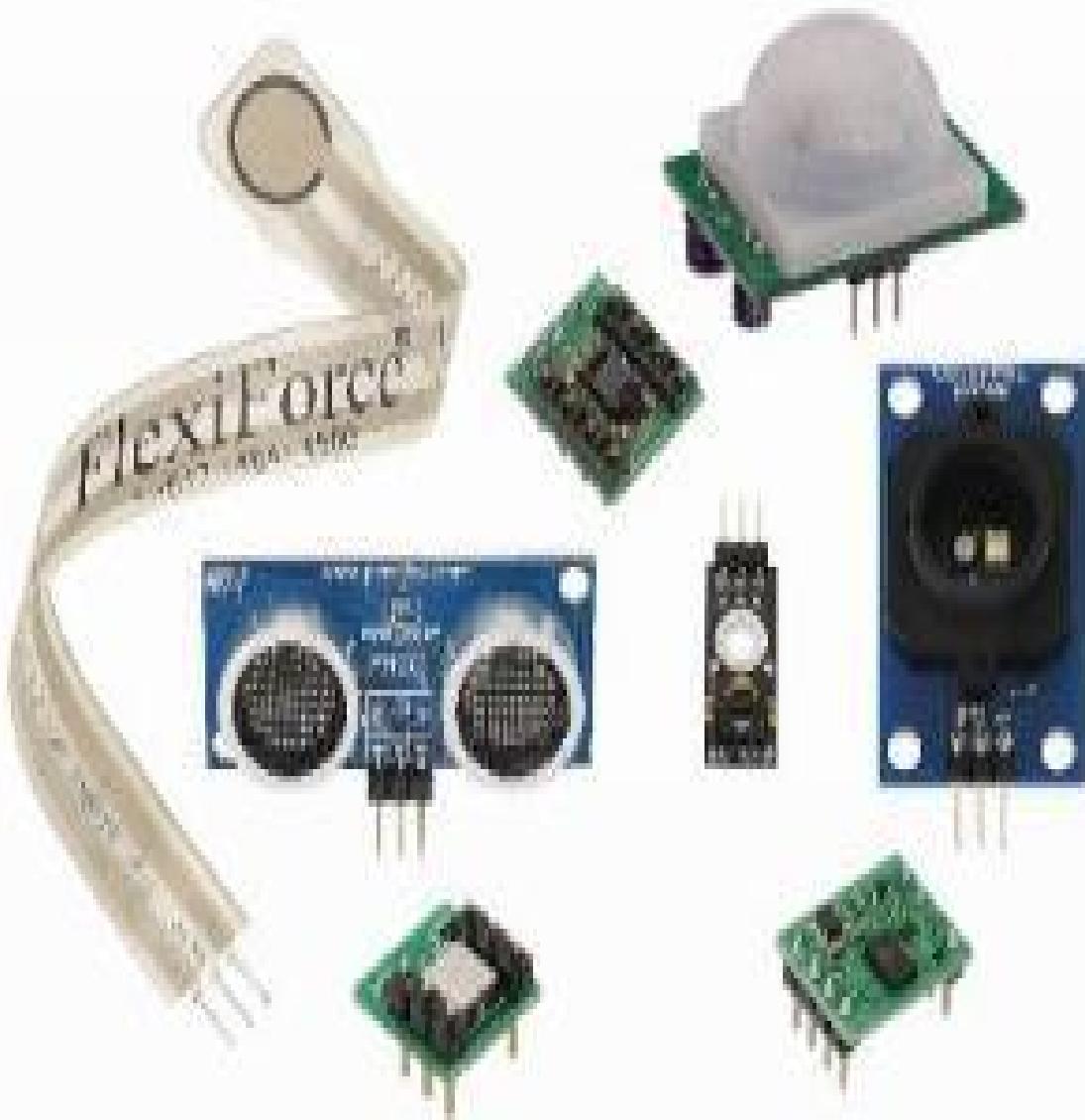
```
22   tone(8,pitch,20);  
23   delay(10);  
24 }
```

The map() function defines the pitch as wide, and you can attempt to change the frequencies to determine the one which is perfect for your musical style.

The tone() function works in the same manner as the PWM in the function analogWrite(), but it has one major difference. The analogWrite() has a fixed frequency. However, with the tone(), you will continue to send pulses while you change the rate.

The tone() function allows one to define frequencies when it pulses a piezo or speaker. If you apply sensors into a voltage divider circuit, you will not receive a complete range of values. However, calibrating the sensor allows you to map inputs into a specific field.

## Inputs, Outputs, and Sensors



As we've mentioned above, sensors for Arduino range in uses and functions. Arduino is designed to be inclusive of multiple types of sensors, which can, in turn, be applied to the programming language C. Arduino is equipped to work with these sensors, and they can be purchased relatively cheaply from Arduino or on other sites. Some examples for Arduino sensors include

- The ultrasonic module
- IR infrared obstacle avoidance sensor module
- Soil hygrometer detection module soil moisture sensor
- Microphone sensor
- Digital barometric pressure sensor board
- Photoresistor sensor module light detection light
- Digital thermal sensor module temperature sensor module
- Rotary encoder module brick sensor development board
- MQ-2gas sensor module smoke methane butane detection
- Motion sensor module vibration switch alarm
- Humidity and rain detection sensor module
- Speed sensor module
- IR infrared flame detection sensor module
- Accelerator module
- Wi-Fi module

While there are many others, these are just a few. Some sensors are easier than others to connect with different Arduino units, so be aware which will best fit your Arduino.

What you will learn in this chapter:

Introduction to signals

Work with sensors

Understand PWM

What you will need for this chapter:

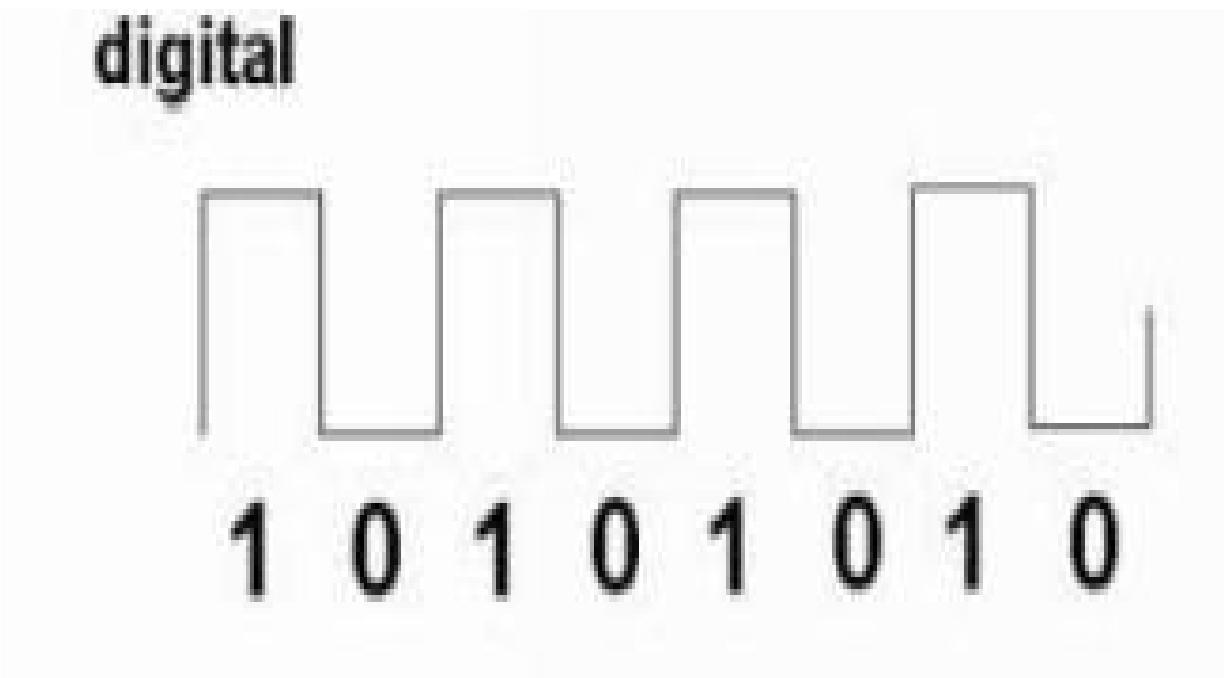
Arduino UNO board

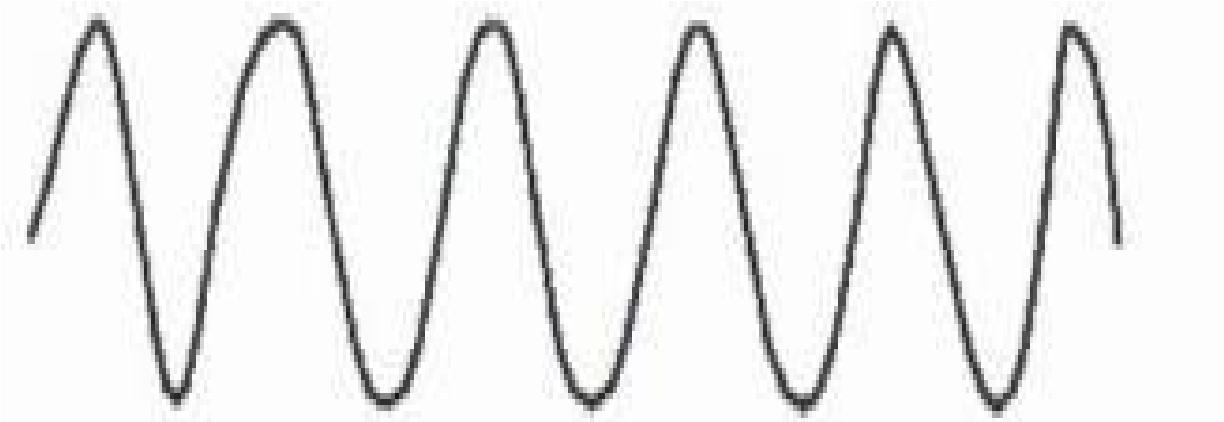
Multimeter

Sensors

Resistors

There are two types of signals:





## Analog Signal

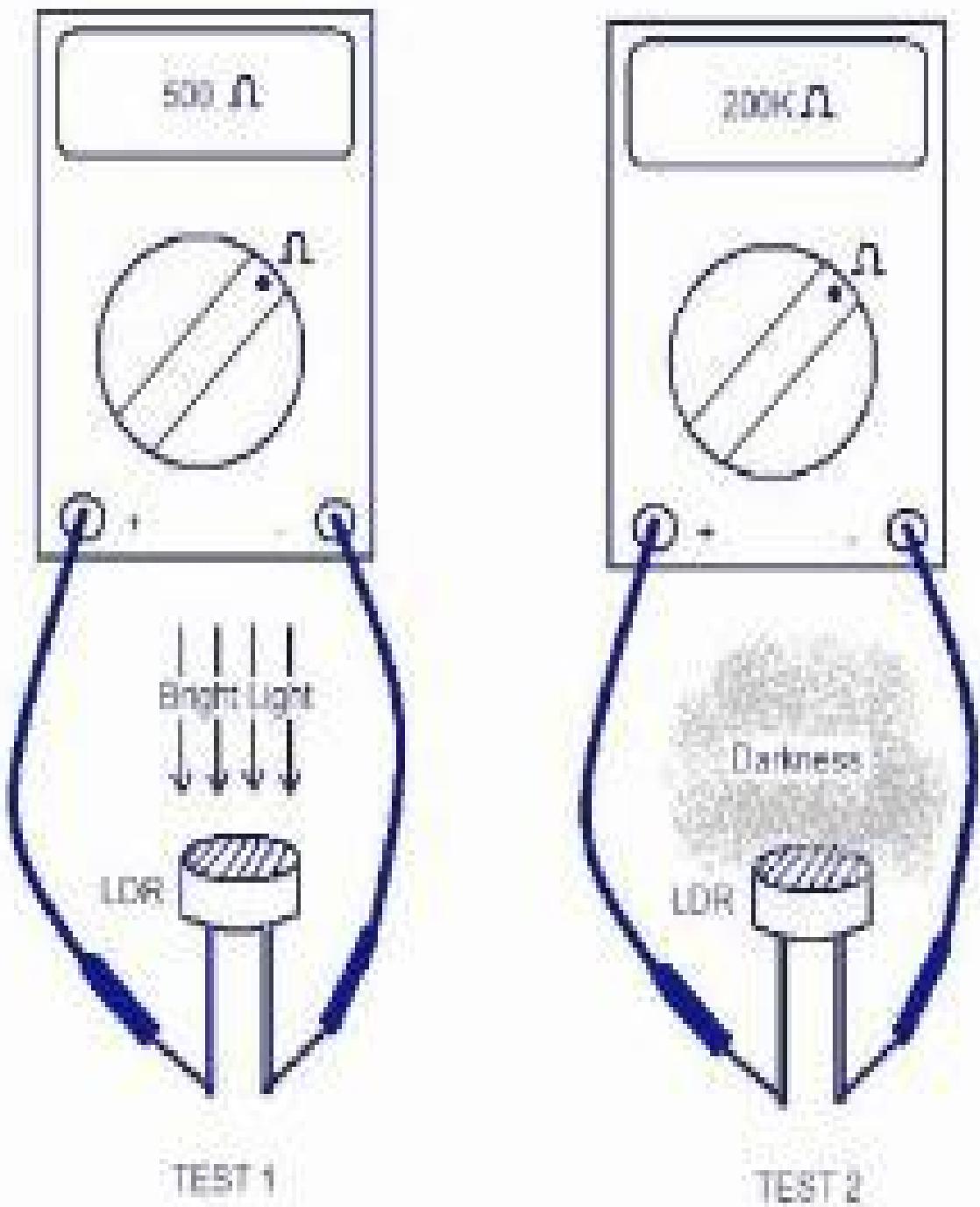
Why are analog signals important?

Analog inputs like the voltage of some sensors are a result of changing some factors. For example:

Photo – resistor: which is an electrical resistor that changes its value depending on the amount of light.



We can measure the voltage on this resistor using the multimeter.



- We can use this phenomenon to measure any other environmental factor using proper sensors that convert the factor into analog signals such as light, temperature, humidity, power, etc.

- On the Arduino UNO (ATMega 328p), there are six input pins for the analog signals it starts from A0 to A5, and it can measure voltages with 4.8 millivolts, and that means it is very accurate when measuring a lot of

applications.

A0

A1

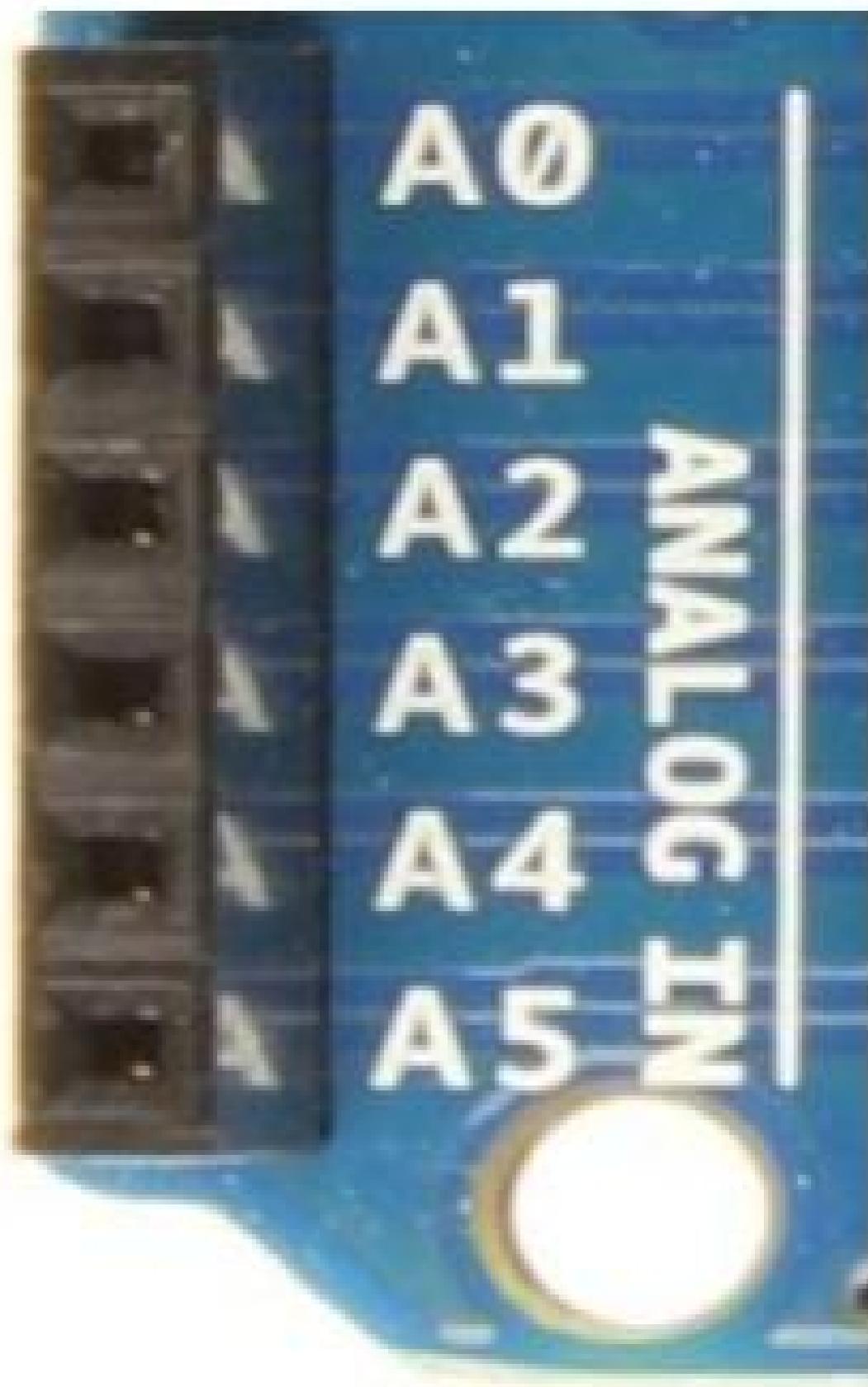
A2

A3

A4

A5

ANALOG IN

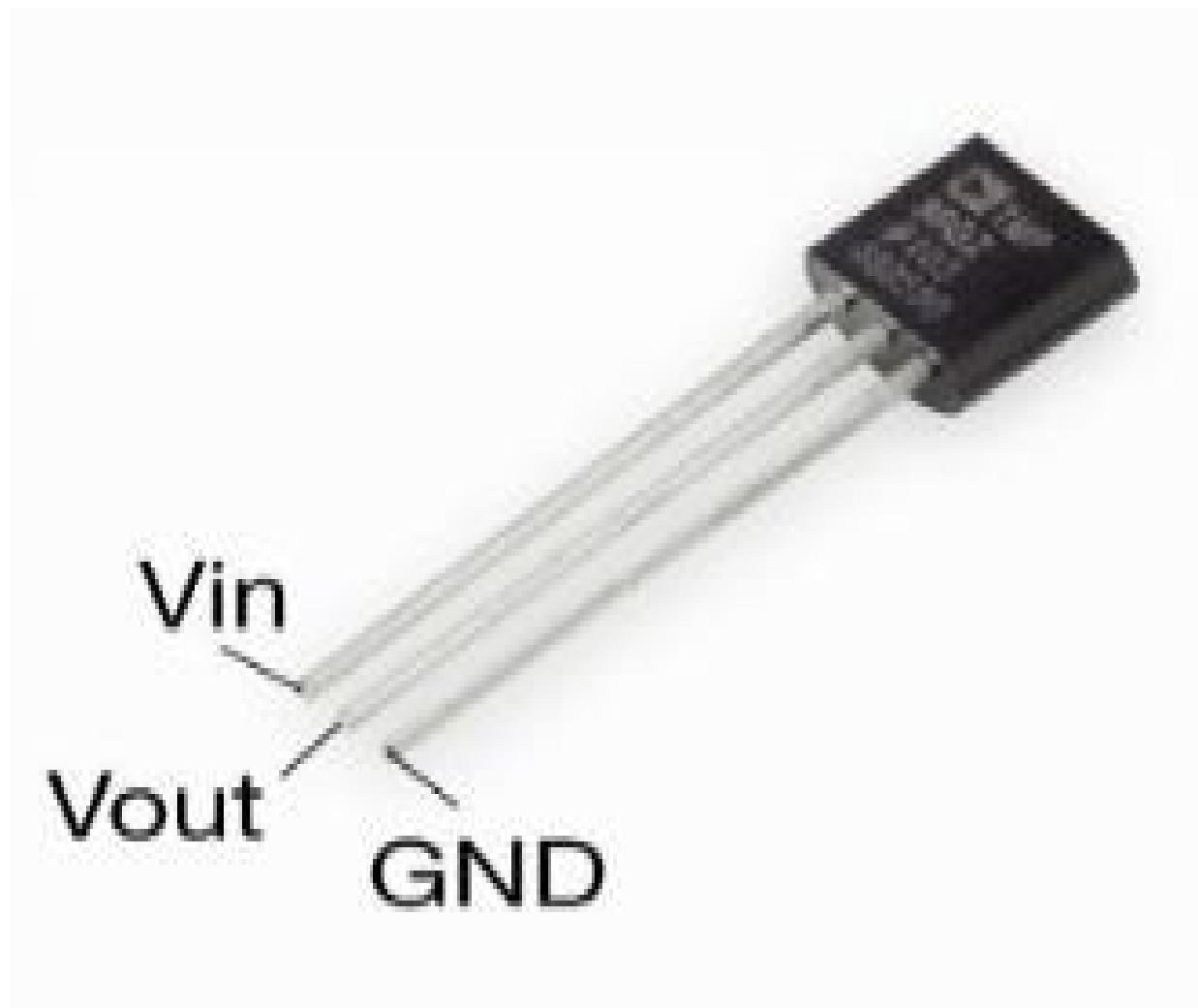


How do sensors generate analog signals?

Let's take the temperature sensor as an example: the temperature sensor contains a very sensitive transistor which is made from silicon. And as we know, silicon is highly affected by the temperature.

The temperature sensor has the following:

1. Input  $V_{in}$  (2.2v to 5.5v).
2. Signal leg  $V_{out}$  to get the measurement.
3. The ground leg  $GND$  to connect it with any ground point.



Components you will need for this example:

- Multimeter
- AAA 1.5v battery (2)
- Temperature sensor (TMP35 or TMP35 or LM35)

### *Steps*

- Bring the two AAA batteries and put them together in the battery holder so you will get 3 volts.



**Emerson** +



- Connect the red wire with that of the battery holder to the temperature Vin's leg.
- Connect the black wire of the battery holder to the temperature sensor GND leg.
- Put your multimeter to the voltage mode as shown below:



- Connect the GND leg to the black probe and connect the red probe to the Vin leg as shown.
- Note the reading of the voltage on the multimeter. It should be 0.76 volts.
- Now put your hand on the sensor (this movement will raise the temperature) and note the reading of the multimeter. The reading on the multimeter will rise and become higher.



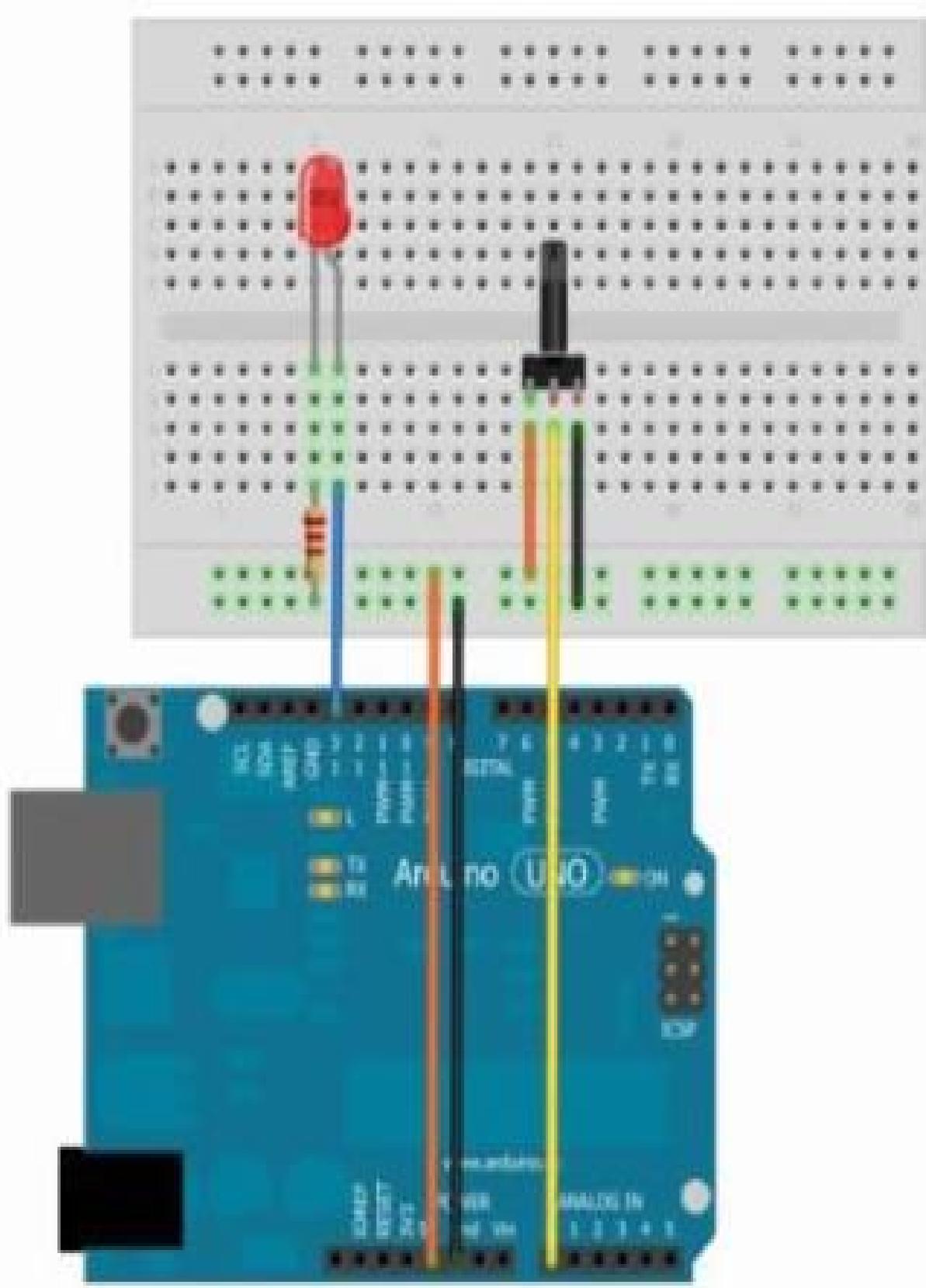
- As with any sensor work in the same manner of the temperature sensor, it behaves depending on the environmental factor and changes its internal resistor, so it changes the output voltage which can be measured by an analog sensor.

Example 4: Control light amount using a potentiometer (wiring)

Example components

- Arduino UNO board
- Breadboard
- LED
- 560 ohm resistor
- 10 k ohm potentiometer
- Wires

Connect the components as shown:



Example 4: Control light amount using potentiometer (Coding)

//create new file form the Arduino IDE and write the following code:

```
const int sensorPin = A0;
```

```
const int LedPin = 13;
```

```
int sensorValue;
```

```
void setup ()
```

```
{
```

```
PinMode (LedPin, OUTPUT);
```

```
}
```

```
void loop()
```

```
{
```

```
sensorValue = analogRead(sensorPin);
```

```
digitalWrite(LedPin, HIGH);
```

```
delay(sensorValue);
```

```
digitalWrite(LedPin, LOW);
```

```
delay(sensorValue);
```

```
}
```

*analogRead(pin number)*. This function reads the voltage as an analog signal (the microcontroller can measure voltages from 4.8 millivolts to 5 volts), and it also converts these values to digital values from 0 to 1,024. This conversion is called *analog to digital converting (ADC)*.

For example:

If the input voltage to the A0 equals the following values:

4.8millivolt = 1 in digital

49millivolt = 10 in digital

480millivolt = 100 in digital

1volt = 208.33 in digital

2volt = 416.66 in digital

5volt = 1024 in digital

```
sensorValue = analogRead(sensorPin);
```

- In this statement, the microcontroller will store the value of the sensor reading in the sensor value variable, and then the microcontroller will turn on/off the LED for a period of time equal to this variable (sensorValue).
- In this example we have used a variable resistor, so we could change the value of the resistance.



Example 5 photoresistor as a light sensor (Components)

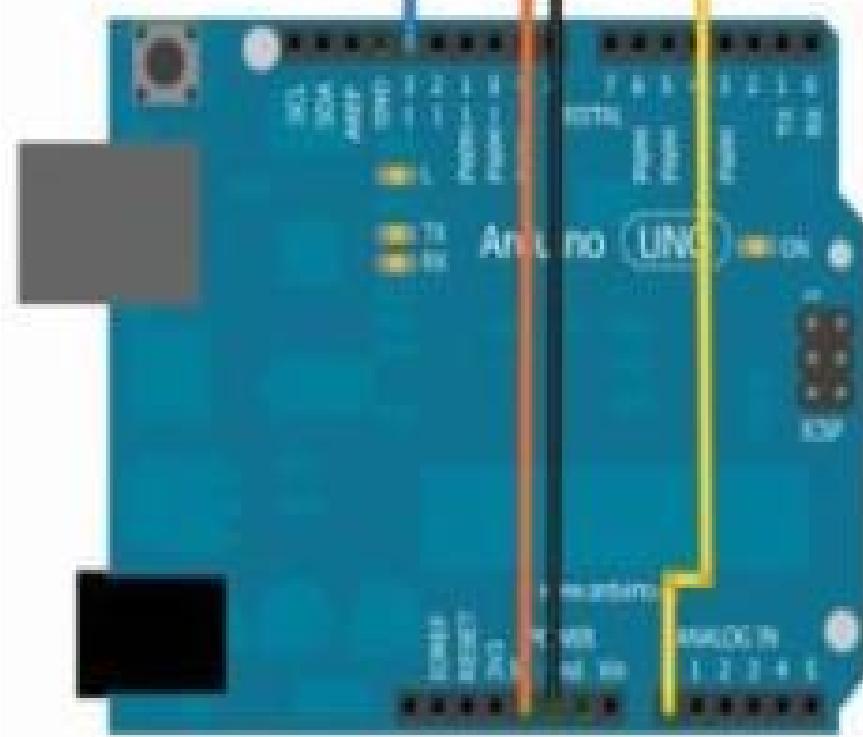
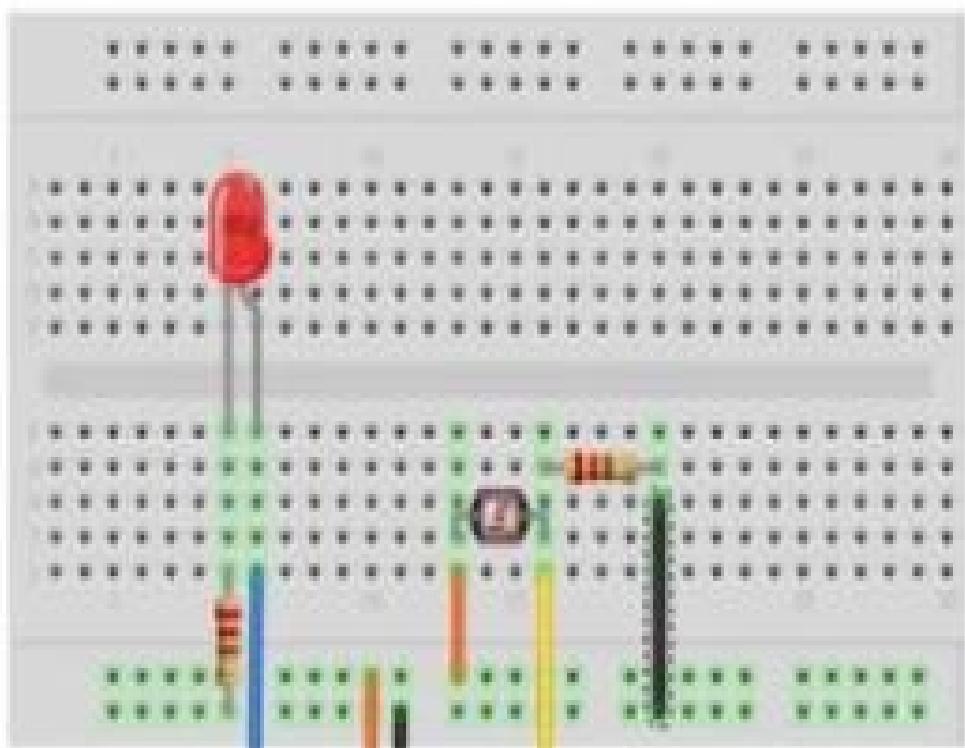
- Arduino UNO board
- Breadboard
- LED

- 560-ohm resistor

- Photoresistor

- wires

Example 5: Photoresistor as a light sensor (Wiring)



- Connect the components as shown:

Example 5: Photoresistor as light sensor (Coding)

```
// select new file from the Arduino IDE

const int lightPin = A0;

const int ledPin = 9;

int lightLevel;

void setup ()

{

pinMode(ledPin, OUTPUT);

}

void loop ()

{

lightLevel = analogRead(lightPin);

lightLevel = map(lightLevel, 0, 900, 0 , 255)

lightLevel = constrain(lightLevel, 0, 255);

analogWrite(ledPin, lightLevel);

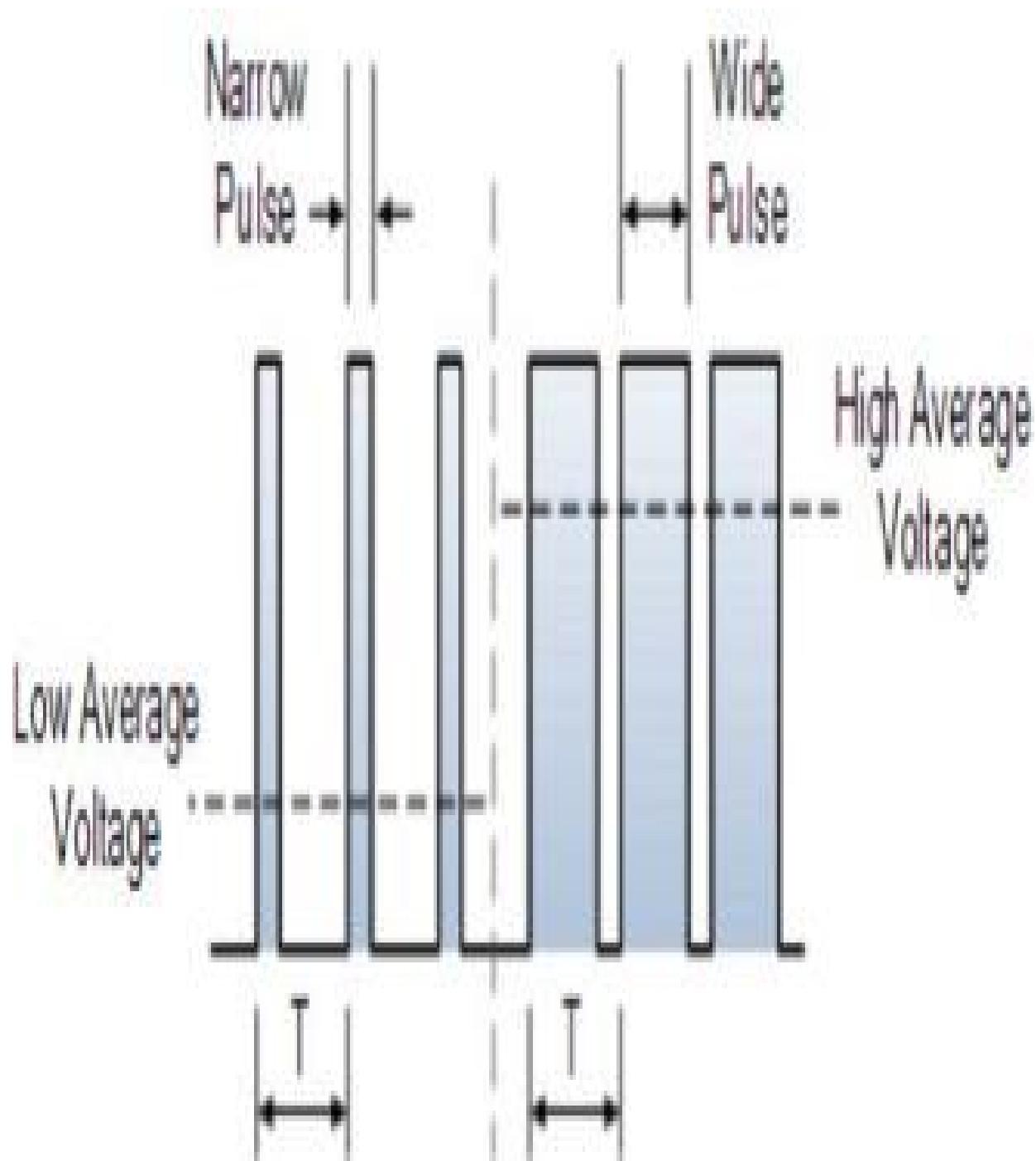
}
```

- Now you can upload this code on your Arduino board and look what will happen to the LED after focusing the light on the photoresistor. Then put your hand on the photoresistor and look what will happen to the LED.
- `analogWrite(pin number, value);`

This function generates an analog output, and this function can be applied to all of the pins with pulse width modulation (PWM).

They are pin 3, pin 5, pin 6, pin 9, pin 10, and pin 11 (any pin with  $\sim sign$ ).





How can we use it?

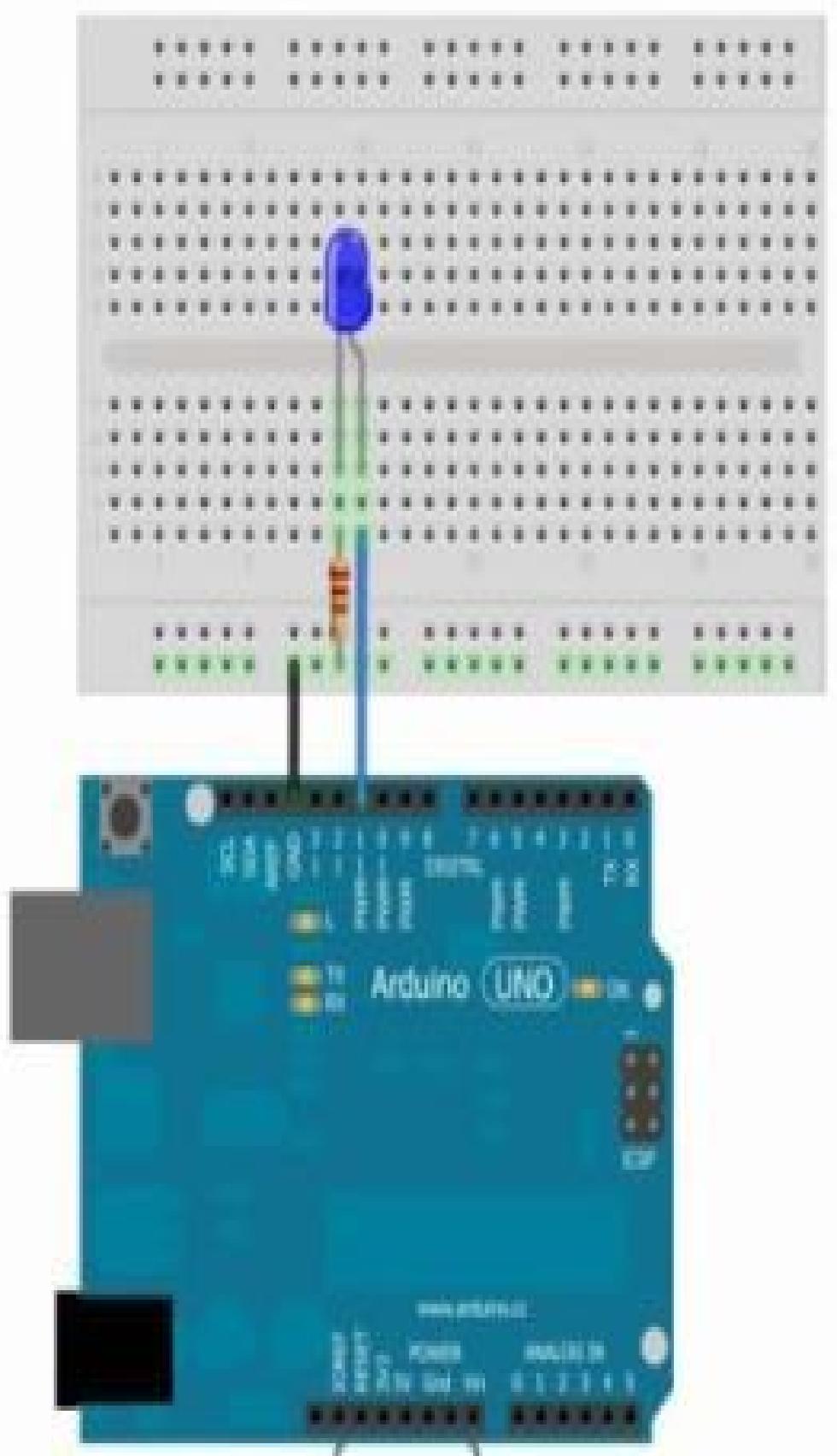
A lot of electric components are dealing with different voltage values.

For example, when you apply 3 volts to the LED, you will get a very small amount of light, and if you raise the voltage to 4 volts, you will find out that

the light will be brighter and so on.

And if you use a motor, for example, when you increase the voltage the speed of the motor will be faster.

Example 6: LED with PWM (wiring)



Connect the components as shown:

Example 6: LED with PWM (coding)

// open the Arduino IDE and select new file then write the following code:

```
const int ledPin = 11;
```

```
int i = 0;
```

```
void setup()
```

```
{
```

```
pinMode(ledPin, OUTPUT);
```

```
}
```

```
void loop()
```

```
{
```

```
for (i = 0; i < 255; i++) // LED will be lighter
```

```
{
```

```
analogWrite(ledPin, i);
```

```
delay(10);
```

```
}
```

```
for (i = 255; i > 0; i--) //LED will be darker
```

```
{
```

```
analogWrite(ledPin, i);
```

```
delay(10);
```

```
}
```

```
}
```

```
for (i = 0; i < 255; i++)
```

I = 0 → the initial value

I < 255 → to set your condition

I++ → is the iterator in this example will add 1

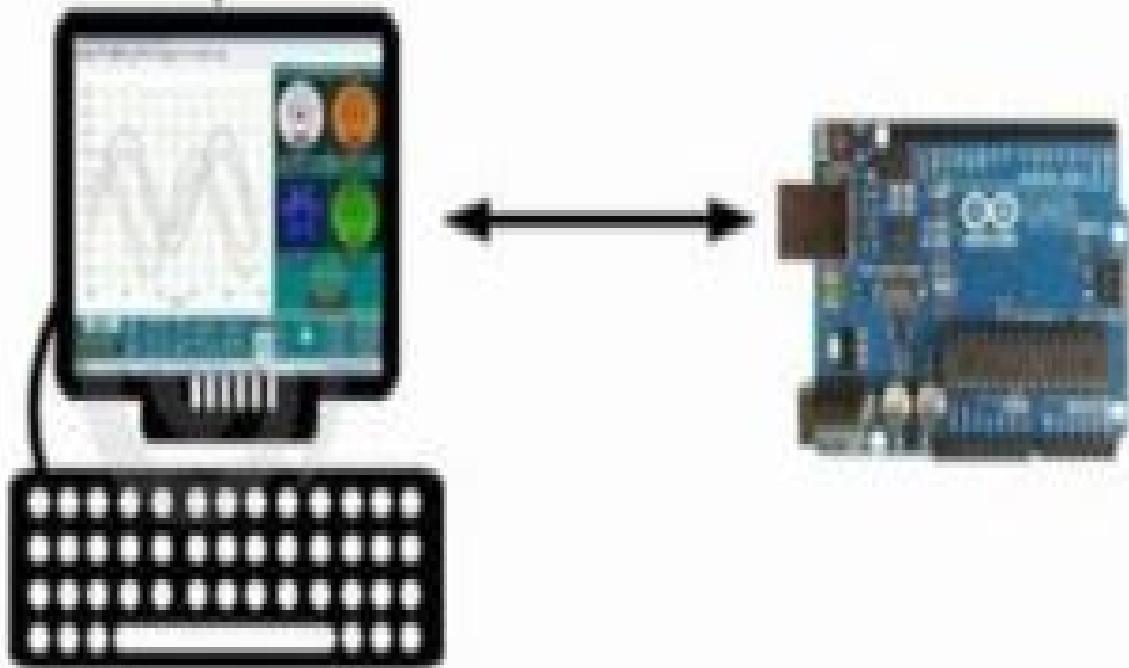
I++ → I = I +1

## Questions

To check for understanding, answer each of the questions below.

1. Describe the difference between digital and analog signals.
2. What is pulse width modulation?
3. Design a circuit to turn on/off five LEDs in sequential order.
4. Write the code for Example 3.

# Computer interfacing with an Arduino



How you choose to interface with a computer depends on the types of cables available to you. Remember that each Arduino can simply connect to a computer through a USB port. Connecting your Arduino with your computer depends on the programming language you use and add-ons you need to incorporate to let the Arduino interface smoothly with your computer.

What you will learn in this chapter:

- How to connect your Arduino with your computer

What you will need for this chapter:

□ An Arduino UNO board

□ Breadboard

□ Sensors

□ Wires

## FTDI Chips

- All of the Arduino boards have the capability of sending and receiving data to and from the computer directly through the USB port except the Mini and Lilypad Arduino boards. But you can also connect these boards with the computer using the FTDI interface, which is a small chip used to exchange the data between the Arduino or any microcontroller and the computer.

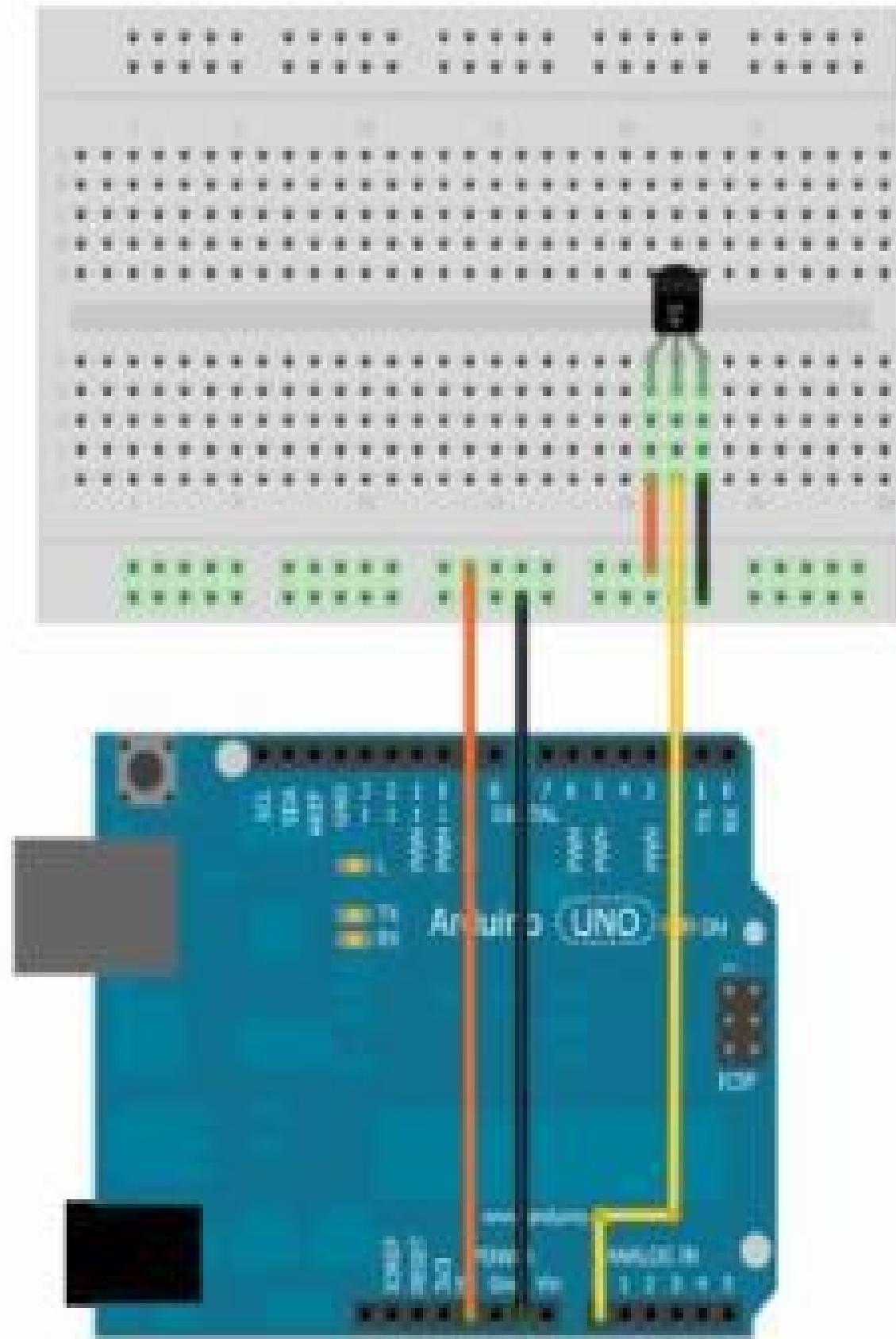


- *In the last examples*, we used the Arduino to read some sensor values, like light and temperature, to show the results on the LED.
- *In this chapter*, the serial interface will send the sensor values to the computer, and we can get the calculations easily.

Example 7: Temperature sensors with serial interface (Components)

- An Arduino UNO board
- Breadboard
- The temperature sensor (TMP 36 or LM35)
- A – B USB cable

Example 7: Temperature sensor with serial interface (Wiring)



### Example 7: Temperature sensor with serial interface (Coding)

```
const int sensorPin = A0;  
  
int reading;  
  
float voltage;  
  
float temperatureC;  
  
void setup( )  
{ Serial.begin(9600); }  
  
void loop ( )  
{  
  
reading = analogRead(sensorPin);  
  
voltage = reading * 5.0/1024;  
  
Serial.print (voltage);  
  
Serial.println(" volts");  
  
temperatureC = (voltage - 0.5) * 100 ;  
  
Serial.println("Temperature is: ");  
  
Serial.print(temperatureC);  
  
Serial.println(" degrees C");  
  
delay(1000);  
}
```

- After verifying and uploading the code, click on the Serial Monitor as shown:

File Edit Sketch Tools Help



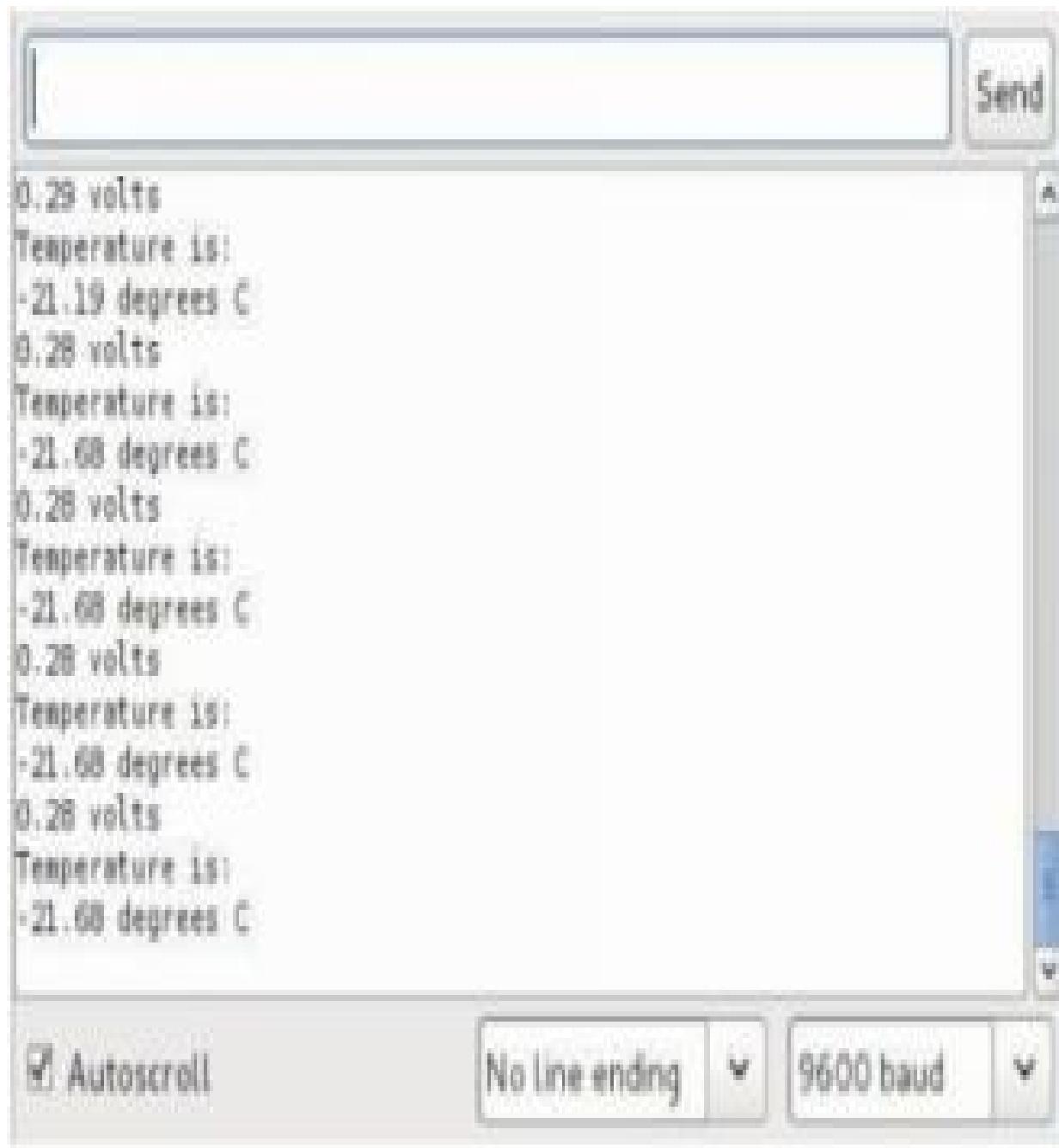
Serial Monitor



sketch\_jun18a.g

```
int sensorPin = A0;  
int reading;  
float voltage;  
float temperatureC;  
  
void setup() {  
  Serial.begin(9600);  
  
void loop() {  
  reading = analogRead(sensorPin);  
  voltage = reading * 5.0;  
  voltage /= 1024.0;  
  Serial.print(voltage);  
  Serial.println(" volts");  
  temperatureC = (voltage - 0.5) * 100 ;  
  Serial.print(temperatureC); Serial.print(" degrees C");  
  delay(1000);  
}
```

- You will see this menu that shows the temperature sensor readings.



- Now try to raise the temperature using any heat source.
- You should be aware that this sensor can handle 150 Celsius.
- (-) This symbol doesn't mean negative, but it is a temporary programming error.

## Example7: Temperature sensor with serial interface (Explanation)

Serial.begin (9600);

- We write this statement to start the communication between the Arduino and the computer through the USB port, so we can receive and send data to and from the computer.
- There are two variables in our code (voltage, TemperatureC) that have been defined with float instead of int because the temperature sensor is a very accurate sensor, and the result will be in floating points number, not integers.

reading = analogRead(sensorPin);

- This instruction is used to record the analog input in the A0 pin.

As we mentioned before that the microcontroller converts the analog signal into digital values from zero to 1024, we used this instruction:

voltage = reading \* 5/1024;

- After the conversion of digital values to voltage, we used *Serial.print (voltage);*

to send this value to the computer and show it on the Arduino IDE.

- *Serial.print ("voltage");* This instruction is used to print the word "voltage"

after its value.

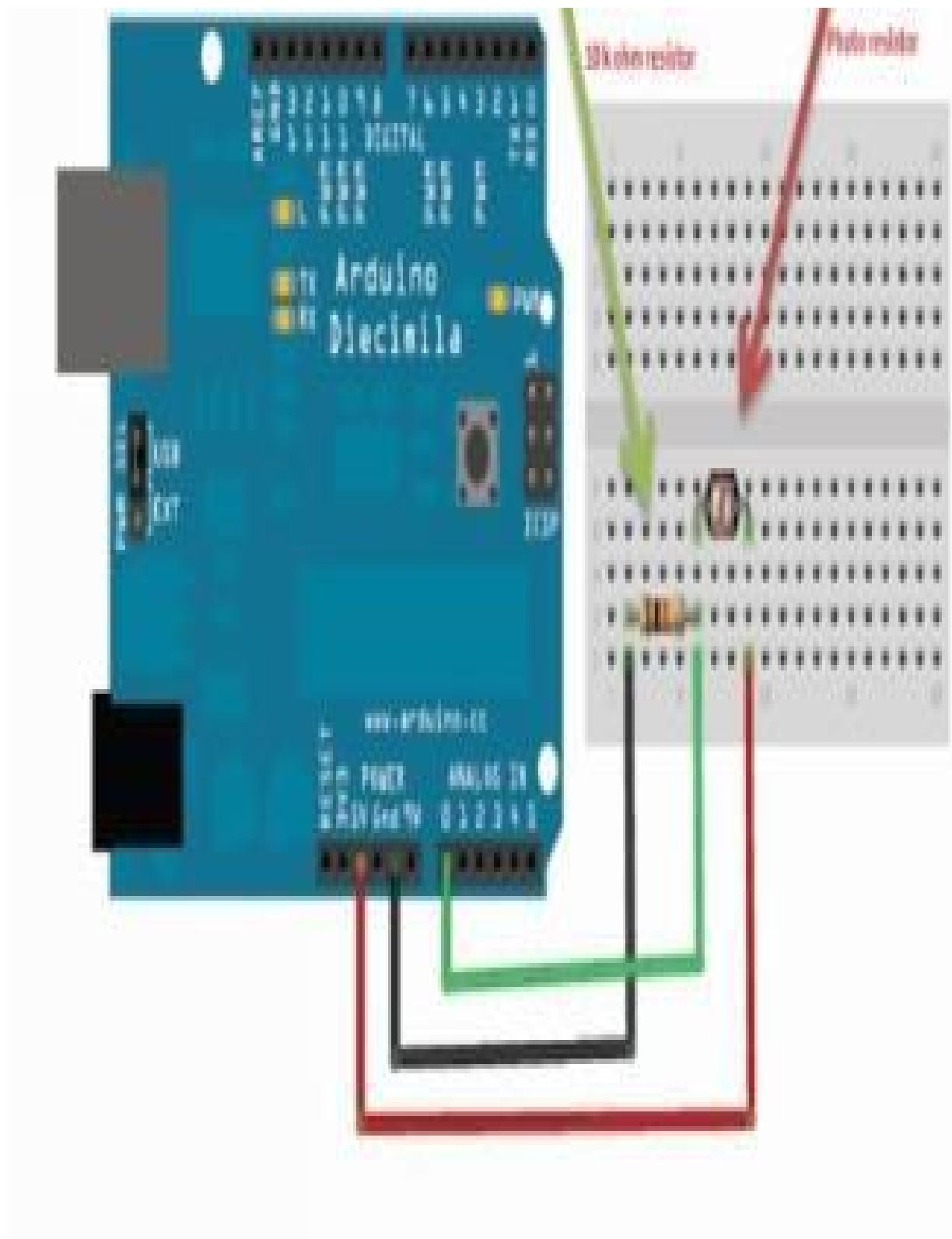
- $TemperatureC = (voltage - 0.5) * 100;$  This instruction is to convert the voltage values to temperature degrees in Celsius and print the value then the word "Temperature" and "degree C."

Serial.print(TemperatureC);

Serial.println("degree C");

- The last line of code is a *delay (1000)*; to make the microcontroller wait one second before sending the voltage and the temperature value to the computer again.

Example 8: Showing the strength of the LED light on the serial monitor (Wiring)



Example 8: Showing the strength of the LED light on the serial monitor (Coding)

```
const int photocellPin = A0;  
  
int photocellReading;  
  
void setup(void)  
{ Serial.begin(9600); }  
  
void loop(void)  
{  
  
    photocellReading = analogRead(photocellPin);  
  
    Serial.print("Analog reading = ");  
  
    Serial.print(photocellReading);  
  
    if (photocellReading < 10) { Serial.println(" - Dark"); }  
  
    else if (photocellReading < 200) { Serial.println(" - Dim"); }  
  
    else if (photocellReading < 500) { Serial.println(" - Light"); }  
  
    else if (photocellReading < 800) { Serial.println(" - Bright"); }  
  
    else { Serial.println(" - Very bright"); }  
  
    delay(1000);  
}
```

After uploading the code on the Arduino, click on the serial monitor.

The screenshot shows the Arduino IDE interface. At the top, there's a menu bar with File, Edit, Sketch, Tools, Help, and a toolbar with various icons. Below that is a tab bar with 'sketch\_jun10a'. The main area contains the following code:

```
int photocellPin = A0;
int photocellReading;
void setup(void)
{
  Serial.begin(9600);
}

void loop(void)
{
  photocellReading = analogRead(photocellPin);
  Serial.print("analog reading = ");
  Serial.print(photocellReading);

  if (photocellReading < 10) { Serial.println(" - Dark"); }
  else if (photocellReading < 200) { Serial.println(" - Dim"); }
  else if (photocellReading < 500) { Serial.println(" - Light"); }
  else if (photocellReading < 800) { Serial.println(" - Bright"); }
  else {Serial.println(" - Very bright"); }

  delay(1000);
}
```

- Now try to do the following:

- Focus the light on the photoresistor
- Cover the photoresistor with any transparent piece of clothing
- Cover the photoresistor with your hand and make sure no light is on it

- This is what you will see:

Send

Analog reading = 326 - Light  
Analog reading = 325 - Light  
Analog reading = 326 - Light  
Analog reading = 118 - Dim  
Analog reading = 76 - Dim  
Analog reading = 66 - Dim  
Analog reading = 75 - Dim  
Analog reading = 64 - Dim  
Analog reading = 100 - Dim  
Analog reading = 11 - Dim  
Analog reading = 0 - Dark  
Analog reading = 13 - Dim  
Analog reading = 0 - Dark  
Analog reading = 0 - Dark  
Analog reading = 0 - Dark



Autoscroll

No line ending



9600 baud

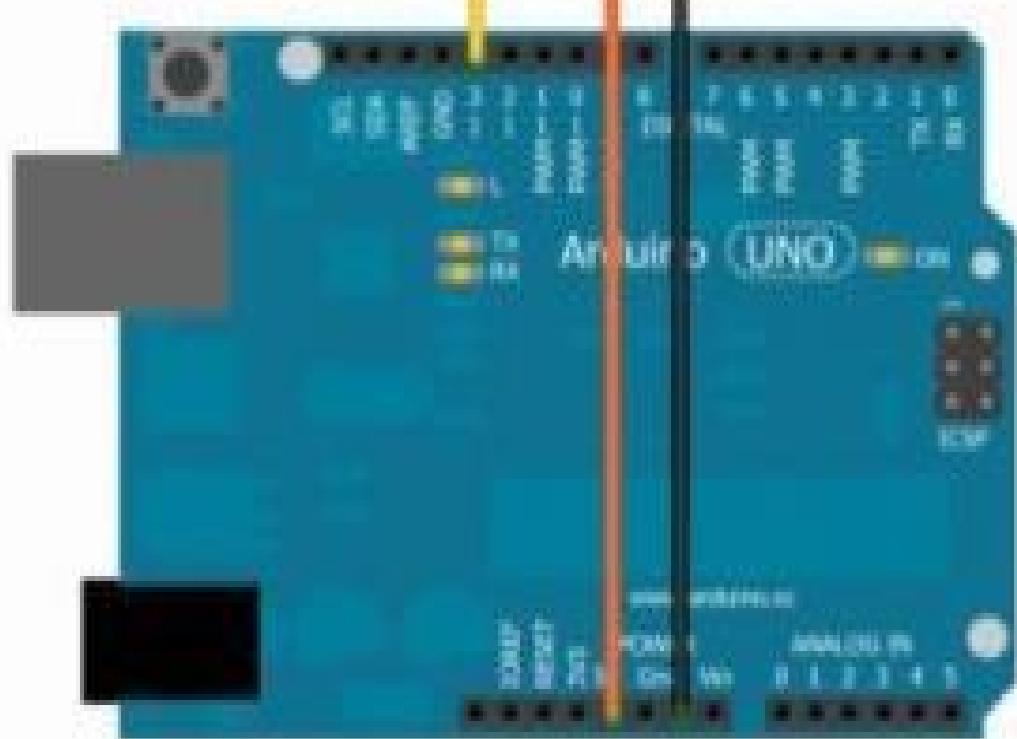
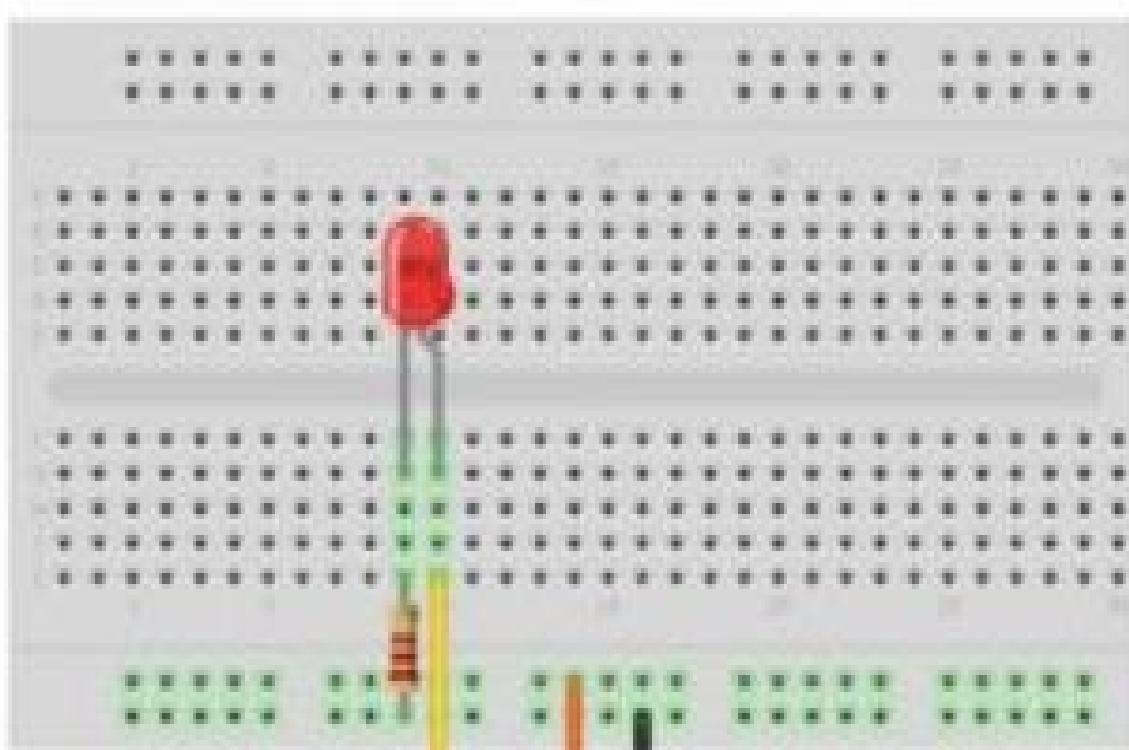


- *Dim* → the amount of light will be small
- *Dark* → there is no light
- *Light* → there is a moderate amount of light
- *Bright light* → the brightness of the light is very high

Example 9: Turn your LED on/off using your computer (Components)

- An Arduino UNO board
  - Breadboard
  - LED
  - 560-ohm resistor
  - Wires
- *In this example* will use the computer to control the LED instead of using a switch, and the Arduino will receive the command using the serial monitor through the USB port.

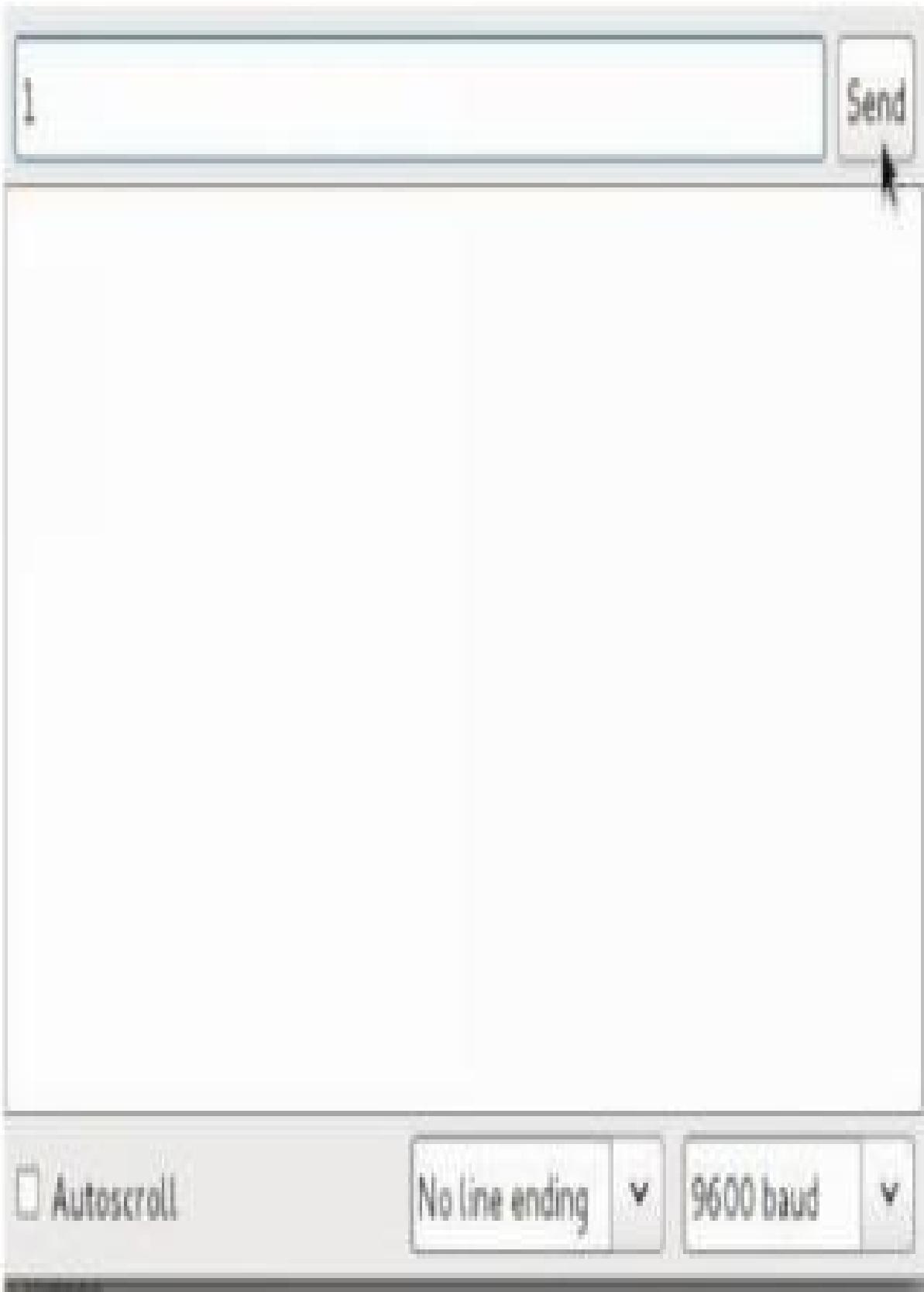
Example 9: Turn your LED on/off using your computer (Wiring)



Example: 9 turn on / off your LED using your computer (Coding)

```
int ledPin=13;  
  
int value;  
  
void setup ()  
  
{  
  
Serial.begin(9600);  
  
pinMode(ledPin,OUTPUT);  
  
}  
  
void loop ()  
  
{  
  
value = Serial.read();  
  
if (value == '1') {digitalWrite(ledPin,HIGH);}  
  
else if (value == '0') {digitalWrite(ledPin,LOW);}  
  
}
```

After the uploading of the code on the Arduino, click on the serial monitor icon and you'll find a search bar. Write "1" on it, and click send. Then write "0", and watch what will happen to the LED.





- In this example, we have used the *Serial.Read()*; instruction to read the data that was sent from the computer to the Arduino through USB, also we added the variable “value” to store the data.

Then we used the if else statement.

- if value == 1 the microcontroller will turn on the LED
- if value == 0 the microcontroller will turn off the LED

## Questions

1. How do you can make the Arduino communicate with the computer?
2. What is the FTDI Chip, and how can you use it?
3. Design a circuit to connect the Arduino with a temperature sensor and an LED.
4. Write the code for Example 3 and control the LED based on the readings of the temperature sensor.

# Catching Up (Revisiting)

In the previous installment in the Arduino series, we covered quite a few things that will help you get started as a programmer in terms of Arduino. While we aren't going to spend a terribly long time, we are going to spend a minute or two reviewing a lot of these concepts just in case this is the first book that you've read in the series. By the end of the chapter, you're going to feel like you have a firm grasp on all of the basics pertaining to Arduino and all of the underlying concepts related to it if you didn't already. If you did, feel free to skip ahead to the second chapter where we start to break more information down as it pertains to the Arduino microprocessor.

## ***Arduino***

The first thing we're going to talk about is what Arduino is. Arduino is a microprocessor board originally developed in Italy. The hardware of Arduino is all open-source, and there's a huge developer community that has developed around it. As a result, it has become an immensely popular circuit board used in a huge number of tinkering projects all around the world. These tinkering projects spread across all sorts of different industries and concepts.

The goal of Arduino is to give people an easy way to understand and tinker with the fundamentals of computing and computer-based hardware without having to shell out the expensive costs that come with normal computing.

## ***The Structure of an Arduino***

Arduino is massively extensible, and there are a number of different hardware modules that can be used with your Arduino board. These peripherals attach to the Arduino and send data to and from the Arduino through what are called pins. There are two kinds of pins: digital and analog.

These are controlled through programs which run on the Arduino. These programs are called sketches. While Arduino programs can be written in

many different languages, this book, in particular, focuses on the most common language for writing Arduino code - C.

C is a very popular programming language historically, and it is also incredible for pulling off the very specific hardware requirements that the Arduino presents. The Arduino by nature doesn't have a whole lot of processing power, so it is important that there's a language that is close enough to the hardware level that it can really easily work with data on the very low level that the Arduino demands, since the Arduino needs programs that don't use much processing power or memory at all.

Through the modules and accessories that one may connect to an Arduino, one is able to do a number of different things. This is why Arduino is a tinkerer's dream; you can do a whole lot for a very low price.

## ***Foundations of C Programming***

In order to work with Arduino properly, you need to have a bit of an idea about programming in C. As I said, we aren't going to spend a terribly long time going over everything in this chapter, but there are a number of essentials that it is important we cover for posterity's sake. We're going to talk about the basic concepts which build up programming in C so that you can work with it with immense ease and feel somewhat natural when you're finding your way around programming in Arduino.

## ***Working with Variables and Values***

Values are, to a computer, anything that mathematical operations can be performed upon. If you're familiar at all with computers, then you'll be aware that this refers to pretty much everything on a computer. Everything on a computer comes down to working with variables and values.

Values refer to anything that is ultimately parsed by a computer mathematically or that can be parsed by a computer mathematically. These are text characters, numbers, or things that the computer natively understands like binary. All of these things are values because they represent, ultimately, a mathematical value to the computer.

These values have types that refer to the value of the data before it is processed as raw data that the computer's hardware can work with. Some types are going to be used a lot more than others in Arduino programming because Arduino programming is all about efficiency. Nevertheless, we're going to spend a bit of time going over all of the types so that you have a firm and solid idea of what these types are as well as how to use them effectively.

**Byte** - This represents an integer value anywhere from 0 to 255. This only takes up 1 byte of data and can be especially useful in Arduino programming since so many things in Arduino programming are on a sequence of 0 to 255 anyway.

**Int** - This represents an integer value of the average size, roughly four bytes. It can hold relatively large values but be careful because if you get into the two million area, you're going to find yourself going far over the buffer limit for integer variables, which means that they're going to be restarting from the very lowest number that an integer can hold.

**Float** - This represents a floating-point number or a decimal. These aren't terribly common in Arduino programming, but they are more common than doubles. These can hold roughly up to 5 decimal places and be as large as about 32,000.

**Double** - This represents a double-precision floating point number. These are twice the size of floating-point numbers in terms of system memory, but they are far more accurate than normal floats and can have a larger non-decimal number than floating points do.

**Unsigned values** - These are the same size as their normal values, like unsigned ints and floats, but they don't have the capacity for negative numbers. This means that they start at 0 and can store positive numbers twice as big as normal integers and floats can, but at the price of not being able to store any negative numbers. When you're working with non-negative numbers, these are a great place to start.

**Short** - These are integer values that are half the size of integer values but twice the size of byte values. They can hold numbers into the 30,000s, but

not any bigger than that. If you're working with smaller numbers, you'll probably want to use these over integers just because they use up less memory.

**Long** - These are integer values that are twice the size of normal integers, which means they can hold numbers well into the two billion areas, but there is no default value type large enough for any number bigger than that aside from unsigned longs which can be roughly four and a half billion.

**Char** - These represent ASCII character values. These are essentially any symbols that can be parsed by a computer and generally are used in order to store and print characters. Characters can be anything from the symbolic representation of a number, like '7', or an alphabetical character like 'a,' or a symbol like '?'. Essentially, if your computer can print it, it is probably a character.

With that, we've covered all of the major data types available for you to use in C and Arduino. There are more, don't misunderstand, but these are the primary ones that you need to understand for right now.

## ***Assignment and Math***

Assigning a value to a variable is really easy. You do so with the assignment operator: `=`. Like so:

```
int myVariable = 6;
```

You can also manipulate variables in this same way.

```
myVariable = 7;
```

You can perform math operations in order to create new values. You do this by using the mathematical operators, also known as arithmetic operators. The arithmetic operators in C are like so:

`b + c`

This signifies addition, of course.

`b - c`

This signifies subtraction.

`b * c`

This signifies multiplication.

`b / c`

This signifies division.

`b % c`

This signifies the modulo. The modulo is the remainder of a given division problem. For example,  $5 \% 2$  would be 1, since  $5 / 2 = 2$  with a remainder of 1.

There are also some shorthand assignment operators. You can change the value of a single variable by adding an equal's sign to any of the above operators, like so:

`a += 1`

This would be the same as "`a = a + 1`". The meaning is consistent across all of the other symbols.

`a += 1` and `a -= 1` have shorthand forms themselves in `a++` or `a--`. `++` and `--` indicate that we're going to either increase the variable by one or decrease it by one, respectively.

With that, we've covered the mathematical operators of C and are ready to move on to other concepts.

## Arrays

Sometimes, you need to store multiple values at once. We'll be talking more about arrays more when we start to talk about strings, but for right now, we can cover the bare essentials of arrays. We already covered them in passing

in the book prior, so we aren't really looking to establish an encyclopedic knowledge of them right now, anyway. Regardless, we are going to cover them enough such that you have a refresher on them.

So, what are arrays? Arrays offer a way for you to essentially group values together by a common idea. As we have established, values are stored at random places in the computer's memory. They are then accessed in the memory whenever they're needed. Arrays serve two purposes in this arena, then.

First, they tell the computer "hey, these things are alike, and they're going to be referenced at about the same time pretty often, so we should be putting them near each other so that way the total time to get from one to another is lesser."

Then, as a result of that, they tell the computer that the values should be near one another in the computer's memory. This ensures that there is minimal travel and retrieval time from one value to the next in the computer's memory. It also enables us to perform operations that we wouldn't normally, like working through the pieces of data in memory in a procedural way as we'll talk about here in a bit.

Arrays essentially set up a contiguous or connected, areas of memory that is the size of  $n$  elements of the array times the  $s$  size of a data type. So, if a data type takes up 4 bytes of memory, and the array has four elements, it clears out and allocates 16 bytes worth of memory right next to each other.

These can then be assigned values individually according to the data type of the array. So, if you created an integer array, you could assign integers to the elements in that array.

You can declare an array like so:

```
dataType arrayName[size];
```

You can also populate it (or partially populate it, at least) by including values in brackets after your declaration of the array.

```
int myArray[3] = {0, 2, 7};
```

The indices of an array start counting at 0. You refer to a given element of an array by referring to its index. So, if you wanted to refer to the second element of an array, you'd do it like so:

```
myArray[1];
```

```
// this would be 2, since 2 is the element at index 1, which is position 2,  
within the array.
```

You can see that arrays are actually relatively easy to understand, but they're nonetheless a fundamental concept for you to work with and try to ingrain as much as possible if you want to be a good Arduino programmer. You're going to inevitably come upon this concept quite a bit in your time programming Arduino sketches, so you need to know it.

## Truth and Logic

It is now time that we rehash a concept that we perhaps didn't go into as much detail on as we should have in the book prior: truth and logic. These concepts are absolutely intrinsic to programming in general, not to mention intrinsic to Arduino programming, so it is important that you understand them.

So, let's start with a simple question - what is logic? Logic is ultimately the use of comparison to reach some particular end result.

However, it also has another definition: the combination of premises and conclusions in the pursuit of some sort of *truth*. Note that logic and truth are not mutually exclusive. The ideas of logic can be used as a foundation for nonsensical things. For example, if my argument were like so:

All dogs are blue

I have a dog

My dog is blue

These statements are logically sound just based on the fundamental structure of the argument. It is not, however, true because its premise of all dogs being blue is incorrect. However, this does mean that by extension that many things can be figured out in a logical manner and that we can use logic based upon truthful premises in order to figure out a truthful conclusion.

These logical premises, in the context of programming, are known as *comparisons*. Comparisons essentially take one thing and another thing and then compare them to something else in order to determine whether something is true.

For example, take 3 and 7; if I were to say that “3 is more than 7”, this is a logical comparison between these two values. This entire statement would be false since 3 is not more than 7.

In programming, we can do this with any given variables and values that we want so long as they are comparable. You can even overload these operators in order to define new ways for things to be comparable when you start with C++ and similar languages.

For right now, though, let’s focus on the things which allow us to compare values. These are known as *comparison operators*. The comparison operators in C and, by extension, Arduino is like so:

`s == t`

This checks to see whether or not s and t are equal.

`s < t`

This checks to see whether or not s is less than t.

`s <= t`

This checks to see whether or not s is less than or equal to t.

`s > t`

This checks to see whether or not s is more than t.

`s >= t`

This checks to see whether or not s is either greater than or equal to t.

`s != t`

This checks to see whether or not s is *not equal* to t.

One of these comparisons has two names: a *statement* and an *expression*. In programming, they're generally referred to as *expressions* in order to not confuse them with the computer science concept of the statement, but you can actually evaluate more than one of these expressions at once. This is known as *statement calculus*, and it occurs through the use of what are called *logical operators*.

There are many logical operators, but the ones in C that you most need to know are like so:

`A && B`

Checks to see if both expression A and expression B are true.

`A || B`

`!A`

Checks to see if the statement A is *not* true. If it is *not* true, then we will return true since the statement “not A” is true.

If you’re familiar at all with discrete mathematics or symbolic logic, then you’ll see quite easily how many of the concepts carry over from it. However, even if you aren’t familiar, they’re still very easy for you to grasp nonetheless.

Conditionals

Now that we've talked a bit about logic and truth, we can build on that knowledge to talk about what is called *conditional statements*. Conditional statements are one major part of control flow. Control flow is extremely prominent in computer science, and almost every application you've ever used will have some degree of control flow built into it. Control flow is, after all, the basic way that you can give your program some sort of "intelligence," if we're defining intelligence as the capacity to make decisions based off of given data.

These can take two forms: the passive and the active conditional. The passive conditional is the most basic form, so we're going to cover that first.

The passive conditional is called so because there is no obligation for the program to run the code of the conditional. For example, if the program gets to the conditional and the condition isn't met, the code is skipped altogether, and the program moves on to the next part of the program.

The passive conditional is established through the *if statement*. The if statement just evaluates whether or not a given condition is true and then will execute the code within the if statement's code block if it is true. Otherwise, the code block will be skipped entirely. The syntax for an if statement is like so:

```
if (condition) {  
    // code goes within  
}
```

This is complemented by the *active conditional*. The active conditional is parallel to the passive conditional because it forces the program to execute *some* code even if the statement isn't true. So in essence, if the statement is true, then the code within *that* code block will run. Otherwise, an alternative code block that has been written will be executed. This is done through the *else statement*.

```
if (condition) {
```

```
// code goes within  
} else {  
  
// back-up code is here  
}
```

However, you may realize that sometimes you want to test more than one condition. You can do this with the *else if* statement which is supposed to be sandwiched between your if and else statements. You can supply additional conditions to be tested if the initial condition tested doesn't turn out to be true. It will test conditions in sequence, and if none of them are true, then the else statement will execute. The syntax for an if statement is like so:

```
if (condition) {  
} else if (condition) {  
} else if (condition) {  
} else {  
}
```

With as many or as few else if statements as you really want there to be. There is no upward or downward bound so don't worry too much about that.

## Loops

Loops are an essential part of programming. In the first book we only really covered for loops, but in this book, we're going to cover both for loops and while loops. Loop logic is an essential part of our daily lives, but a lot of the time we fail to consider how important it really is to things that we do every single day.

For example, consider the act of counting from 1 to 5. You start at the number one; you say the number out loud by forming your mouth into the

proper shape and expelling air, then you add 1 to the number; this repeats until you reach the number 6. At the number 6, you see that we are now bigger than the number 5 so you no longer say the number aloud.

This is a relatively simple example, but it is an important one nonetheless because it really frames just how insidious and important loop logic is.

There are two different main forms of loops in C that you'll need to know: for loops and while loops. We're going to cover while loops first because they're far simpler in concept.

While loops are relatively simple to understand but they're harder to know when to use accurately. Much of the time, you're going to be using for loops just because they seem to have more obvious and immediate uses than while loops do.

With for loops, you have really obvious bounds, but with while loops, you don't. For this reason, while loops are best suited to those cases where you don't have an actual finite number of times for a loop to run.

A while loop simply checks a condition and then runs the code within the body of the loop for as long as that condition is met. If that condition is ever not met, then the loop will exit. Easy enough!

The syntax for a while loop is like so:

```
while (condition) {  
    // loop's internal code  
}
```

While loops are best suited to the concept of the “game loop.” These loops aren't exclusive to games, of course; they just express the idea of a game, because games will do the same thing over and over until a win or lose condition is met. When those conditions are met, the game is considered over.

The game loop is based on the idea of having either a true or false variable that is changed to the opposite when a certain condition is met. So, for example, the code may be:

```
#define TRUE 1  
  
#define FALSE 0  
  
int bHasWon = FALSE;  
  
while (bHasWon == FALSE) {  
  
    // code goes here  
  
}
```

Then have something that will change bHasWon to TRUE when the player wins. This will indicate that the loop should be terminated because the win condition has been met. Of course, this logic - again - can be used for many things aside from games. Anything with a central main menu will in fact likely use this sort of loop logic.

The other kind of loop is the *for* loop. The primary purpose of the for loop is to allow you to iterate through a given set of data with ease. For loops start with the creation of an iterator variable, which can be named whatever you want. The iteration step can be many things, but it normally is by 1 (*iterator++* or *iterator--* for +1 or -1 each time, respectively).

The syntax for a for loop is like so:

```
for (iterator declaration; condition; iteration step) {  
  
    // code goes within  
  
}
```

So to print out every number in an array, we could do the following:

```
for (int i = 0; i < (sizeof(myArray)/sizeof(myArray[0])); i++)
```

```
{  
    printf("%s\n", myArray[i]);  
}
```

Where `sizeof(myArray)/sizeof(myArray[0])` gets the number of elements within the array altogether.

## Functions

The last thing that we need to talk about and rehash before moving on to the next chapter is the concept of *functions*. Functions are a foundational concept in C programming and Arduino by extension. You're going to run into them a lot, so it is important that you understand exactly how they work. Fortunately, they aren't a very terribly difficult concept to understand! Functions simply are based around the idea of breaking something down into code which can be reused over and over.

Functions may be familiar to you through things like past math classes, where you would have something like  $f(x) = y$ .  $X$  was the argument, and the function manipulated  $x$  in order to give you the value  $y$ . Functions in computer science are relatively similar (and are very much similar to higher level functions in mathematics, but I'm not so willing to assume that everybody who reads this book has already worked with those, so I'm going to pull back on that one.)

Functions have a few basic parts. First, they have their declaration. In C, you either have to declare a new function at the start of the file, called prototyping, or you have to put it before your main function. For simplicity's sake, we'll go ahead and just prototype them at the start of our file and put them after our sketch's primary functions.

Functions also have a return type. This is the kind of value that they *give back* at the end of the function. So, for example, a function called `convertTemp` would probably give back a decimal number. Therefore, the type of function would be a *float* or a *double*.

Functions can also be written which *don't* have a return type. These functions are called *void* functions. They are valid, especially for performing certain operations that are to be done over and over but aren't particularly mathematical in and of themselves, like printing text to a serial or spinning a motor or something of the like.

Functions also have *arguments*. A function doesn't *have* to have arguments, but you can *give* a function an argument. A function's arguments are defined in its declaration. You can tell the types of the arguments as well as the placeholder names. You can then treat the arguments as variables within the body of the function and feed in the actual values when you call the function later in the program.

Again, though, a function doesn't necessarily *have* to have an argument. This isn't a requirement for a working function. Remember that as you go forward!

The syntax for prototyping a function is like so:

```
functionType functionName(arguments, if any);
```

And the syntax for actually writing a function is like so:

```
functionType functionName(arguments, if any) {  
    // code within the function  
    // return valueName if necessary;  
}
```

So, let's make a function which will return the volume of a cone as a float. This is going to need two arguments, radius and height. It will be a double since we're working with pi and want it to be as accurate as possible. We will feed in doubles as arguments, too.

We could prototype the function near the start of our program like so:

```
double volumeOfCone(double radius, double height);
```

Then at some point in the program afterward, but not within another function, we could include the actual body of our function like so:

```
double volumeOfCone(double radius, double height) {  
    return 0.33333333 * 3.141569 * (radius * radius) * height;  
}
```

We can then treat the return value of this function as a value of itself. So, we could create a double variable and assign the return value of this function to it:

```
double volumeOfConeRThreeHFive = volumeOfCone(3.00, 5.00);
```

This would return the volume of a cone with a radius of three and height of five and save it to the variable we created. We can also insert it anywhere that value is accepted, like in a printf statement or into a formatted string or something similar.

With that, we've covered the last thing that we needed to go back over before we get into some of the dense and meaty concepts within this book. In the chapters to follow, we're going to be going over much more in-depth programming concepts as we try to figure out the world of programming in Arduino at a greater level than we had before.

# More In-Depth Computer Science Topics

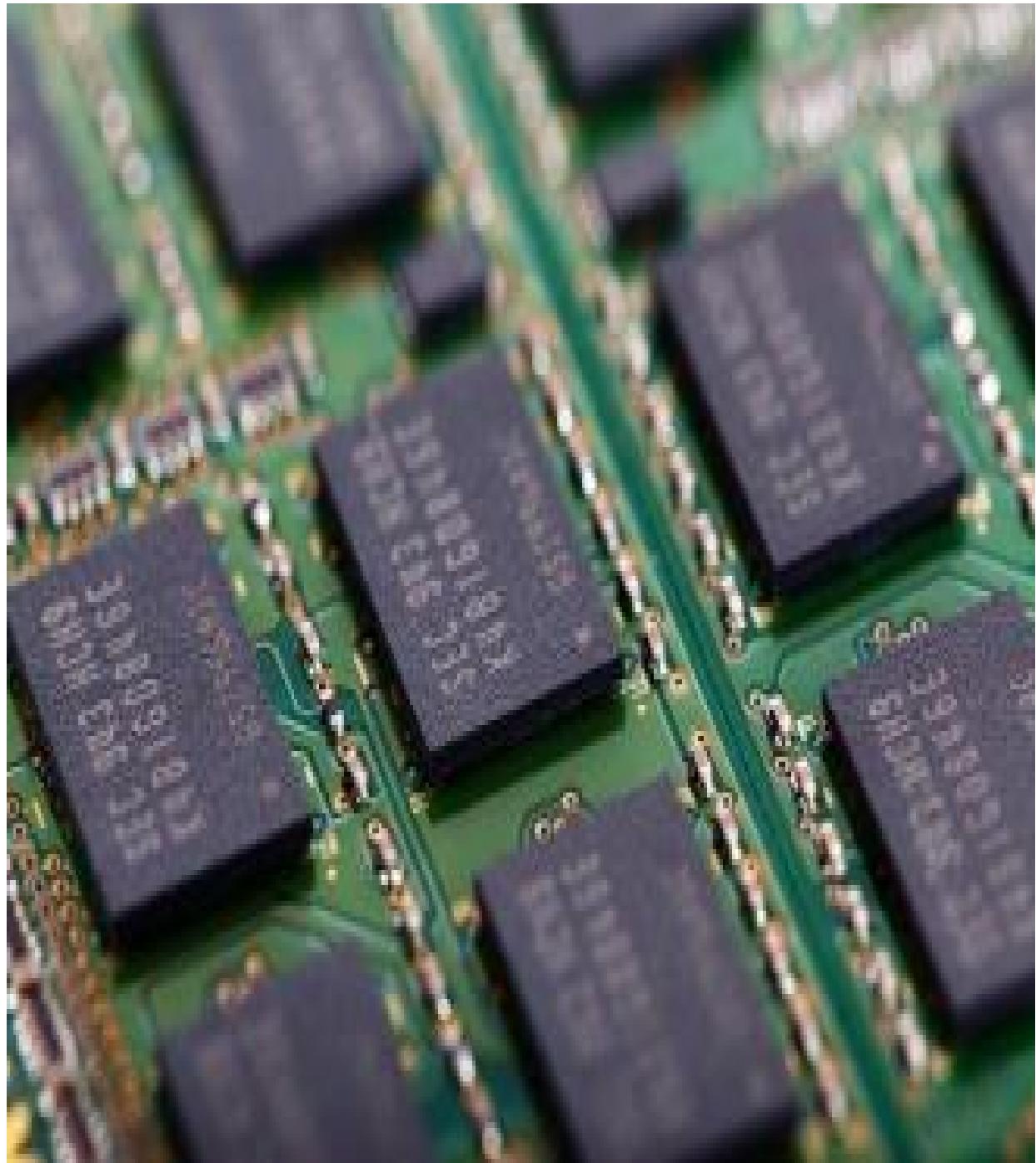
See, working with computers - especially something so precise and hardware-limited as the Arduino - can be immensely rewarding, but if you want to be good at it, you have to have a very firm understanding of a lot of underlying concepts.

It is perfectly fine, for example, to understand what variables *are*, but if you don't understand how they *work*, you may end up wasting a lot of computing power with them when you really don't mean to. And again, when you're working with something like Arduino, that's the last thing that you want to do.

So, in this chapter, we're going to be building on some of our topics that we've already discussed so that you can be an all-around better Arduino programmer and, in turn, a better C/C++ programmer. By the end of this chapter, you're going to feel as though you have a firmer grasp on a lot of different concepts.

## Memory Management and Pointers

The first thing that we're going to talk about in this chapter is the concept of memory management and pointers. This is an immensely important topic, especially when we're talking about Arduino. You don't have a lot of onboard memory to work with, so you need to make the best of what you have.



This can be a little tricky for newer programmers to grasp. In fact, it is tricky enough that in the first book, you were warned to stay away from this in particular. This is because it is a relatively high-level topic. After all, the very idea of pointers gets into some pretty low-level programming that you most likely haven't had any experience with.

Let's think back to a second to our discussion about data and data types. We talked about how you could create variables and all of the things that you can do with them. One thing we didn't really talk about, however, is how these variables work in terms of the computer's memory.

Computers store variables for running the length of a process called the *random-access memory*. You can think of random-access memory as space that can be allocated dynamically and as needed in accordance with the current demands of the program. All values which are worked with by the programmer and the program are stored, to some degree, in the random-access memory. Variables can be defined, which allocate a set space of random-access memory that is the size of the defined variable.

Variables, too, by their very nature, are essentially references to the places that a value sits within the computer's memory. When you refer to a variable, though, you aren't actually working with that *value* necessarily. For example, when you pass a variable to a function, you aren't sending the variable *itself*. Instead, you're sending a *copy* of the variable's value to be manipulated by the newer function. This copy is then disposed of when the function is finished.

In the olden days of computing, this made a bit of sense. After all, you don't necessarily want to *change* the value of some variables every time that you send those to a function. It makes sense for such a case *not to* be the default. Additionally, functions are also stored in the computer's memory in a certain way such that it makes more sense to send them values directly than to send them references to values elsewhere in memory. It makes them, in a manner of speaking, run more efficiently.

However, there are certainly cases where you would want to refer to the value of a variable itself and not just to the value that the variable *refers*. For these cases, you'd want to use pointers. Note at the same time that it is certainly possible to work through most Arduino sketches without ever having to use pointers. However, there are times where you will be working with a data structure or need to *create* a data structure, and in these cases, a working knowledge of pointers is very useful.

Not to mention that whether we're talking in terms of general programming and Arduino programming specifically, pointers are something you *need* to know because they are an important concept of *memory management*. You have to understand memory management in order to write efficient programs, and you need to at the very least understand the concept - if not for Arduino then for anything else that you want to program.

So, what are the key points to take away from all of this? Well, first and foremost, what are pointers? Pointers offer a method for you to refer to a value by its place in memory rather than just by a copy of its value. This is important why? Because it allows you to pass and work with direct values rather than simply copies of those values that you may have through variables. This is foundational to programming in C and will also probably come up sometimes during Arduino programs. While generally, you can write entire sketches without ever even using pointers, it is still good knowledge to have for when you are looking through other people's sketches and learning from the code that they're writing.

So, how do pointers *work* then? Pointers work through a combination of operators called *reference* and *dereference* operators. You can use these in order to create new pointer variables and point them toward an already existing variable.

The first thing that you're going to do is create a new pointer of to the type of value that you're wanting to point. You do this by using the reference operator \*. So, for example, let's say we had this:

```
int apples = 3;
```

And we wanted to create a pointer that would point to this variable. The first thing that we would do is create a new pointer:

```
int *ptr;
```

You can put the reference operator wherever. Most prefer to stick it on the variable name, but others prefer to put it with the type. Others still put it equidistant between the two with a space between both. It depends on the

coding conventions of whatever you're working with, but generally putting it with the variable name is a safe bet.

Afterward, what you're going to do is define to the address you want it to point. You do this through the *dereference* operator: &. You set the pointer variable *itself* to this, not the pointer variable with the reference operator. So, like this:

```
int *ptr;  
ptr = &apples;
```

Then, whenever we go and modify the ptr variable through the reference operator, it will change the *value* stored at the address that we pointed it to. Like so:

```
*ptr = 4;  
  
printf("%d", apples);  
  
// this would print out four since we changed the value at the address referred  
to by the variable apples to be 4 rather than 3.
```

You can see pretty plainly how this would have a lot of utility to you as a programmer when you're trying to pass variables between functions and work with variables in a complex manner. Being able to directly manipulate pointers like this has a lot of useful perks, too. When you're working with a platform where memory is both as limited and as crucial as the Arduino, you're going to want to have at least the *ability* to work with memory directly. There are some more essential things that you will want to, but they are an advanced topic and aren't within the scope of this book. They also aren't particularly useful to Arduino programming itself, such as direct memory allocation through the *malloc* function.

Regardless, knowing how to work with pointers will push you forward as an Arduino programmer because when you do encounter pointers in the wild or have to create functions that pass variables that need to be directly modified, you can do so with ease and not be pulling your hair out in confusion.

## Stacks

Stacks are yet another extremely important computer science concept. They're important primarily because they work in a really crucial and integral way with things like pointers and arrays. So, what are they?

This is a bit more in-depth than Arduino but, you will still inevitably run into the basic stack terminology in discussions on Arduino programming, so it is important that you have a solid idea of what the stack is and how it can be used.

"Stack" is actually a relatively versatile term. The idea of a "stack" simply refers to what it sounds like - a stack of values. You can add variables to this stack. Imagine a block tower. This is essentially how a stack works.

You can put things on top of the stack, and these things are also the first things to be *removed* from the stack. This is especially useful in algorithmic programming, but it does bear some use in Arduino programming as well. Why? Because when you're dealing with complex and limited memory structures, you're going to inevitably run into many occasions where the best path forward is to use a stack. This is because the stack is very memory-easy. It doesn't demand anything aside from the location of the last thing in the stack and the current thing in the stack, and these things are easy to use.

It also gives you an incredibly easy way to refer back to data that you've already used, so that's pretty nifty in and of itself. Stacks are, in essence, an extremely useful tool for any programmer, and there are some Arduino IDE functions that actually reference the concept of a stack and build on the concept.

The stack is built of two essential functions: pushing and popping. Both are extremely easy to understand, so we shouldn't need to spend too much time going over what they actually are.

Pushing refers to adding something *to* the stack. It is a pretty straightforward concept. You can push a value onto the stack in order to save it for later and instantly recall it without having to worry about things such as the *name* of the variable on top of the stack or its value.

Popping refers to taking something *off* of the stack. When you pop a value from a stack, you remove the thing that was most recently added to the stack and take its value however you like. If you want to return the value to the stack, just remember that you're going to have to *push* it on there again. When you pop something from the stack and remove it, the second-to-last thing that was pushed is now the first thing to be popped.

Again, the stack is a relatively simple concept, but it is nonetheless incredibly important to the overall idea of computer science as well as building and expanding your horizons in order to be better programming in general.

## Structures

One of the nuances of Arduino programming and C in general that a lot of people don't take the time to learn as a newer programmer is the idea of *structures*. Structures are a relatively well-kept secret, but they can be incredibly useful.

Perhaps you've heard the term *object-oriented programming*. Structures in C were in many ways' precursors to the idea of object-oriented programming. While they aren't able to be anywhere near as extensively programmed as object-oriented concepts are able to be, they do offer a brilliant way for a programmer to group certain ideas together into a singular structure.

So, what exactly is a structure? Providing some definition for it will help you to get a better idea of how you can use it. Many times, there are concepts in programming that would make a lot more sense if you were to bundle them by putting them together. A great example is to think of a cat. You may need to work with several different *ideas* of a cat in your program, and for that, you'd be defining all sorts of different variables even though they all have a bunch of features in common. So, instead of doing this:

```
cat1Legs = 4;
```

```
cat2Legs = 4;
```

```
cat3Legs = 4;  
cat1Color = 'brown';  
cat2Color = 'black';  
cat3Color = 'white with black spots';  
cat1Breed = 'tabby';  
cat2Breed = 'persian';  
cat3Breed = 'american shorthair';
```

You can just define all of these in a single way and then work with them at a later point by accessing their *member data*. You do this through the use of a period. The syntax for defining a structure in C is like so:

```
struct NameOfStruct {  
    // data within struct  
};
```

So, using the cat example:

```
struct MyCat {  
    int numberOfLegs;  
    String color;
```

```
String breed;  
};
```

You could then define cats like so:

```
MyCat cat1 = {4, 'brown', 'tabby'};  
MyCat cat2 = {4, 'black', 'persian'};  
MyCat cat3 = {4, 'white with black spots', 'american shorthair'};
```

You can see how this presents the programmer with a much easier way to group important data together. There's a good chance you aren't going to be using this data *super* often, but it does allow you to have such a way to *do* this. It is important that you know what it is because you will eventually see it.

All of these concepts are important to Arduino because Arduino is far more restricted in terms of memory and processing power than your home computer would be. It is important that you know how to use these concepts so that you can make the most out of your Arduino's processing ability as well as write simpler and more elegant sketches than you would have otherwise. For example, you could create a structure that held three-byte variables called r, g, and b in order to program your RGB colors alongside the same lines in a simple manner. You could define new colors doing this to access them easily later on in your program, like so:

```
struct color {  
    byte r, g, b;  
};  
  
color blue = {0, 0, 255};
```

See how simple that is? But the utility of doing such a thing is pretty plainly obvious!

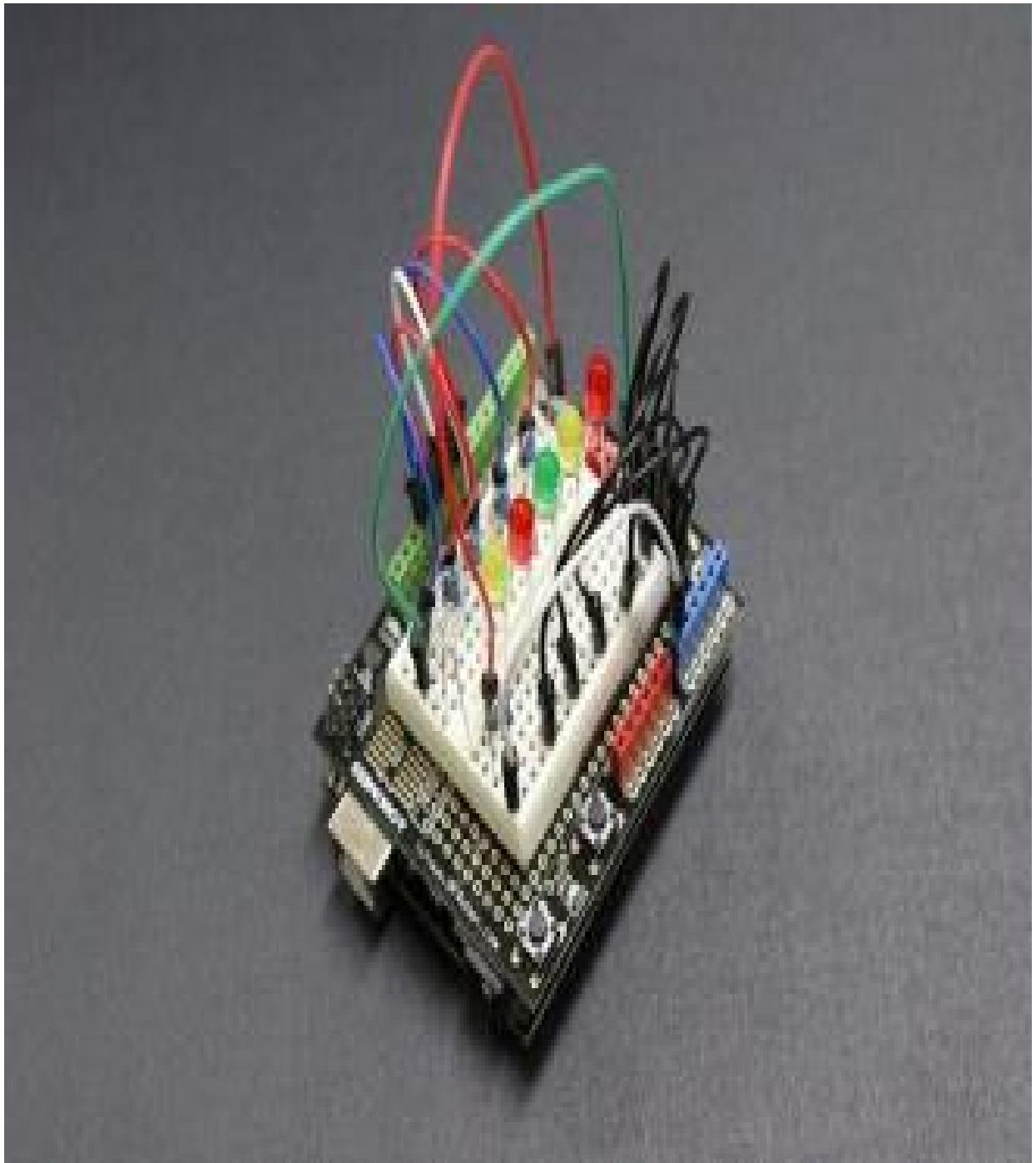
In this chapter, we've covered three major programming concepts that you're going to inevitably come across when you're working with other people's Arduino code and learning from what they've written, so it was important that I develop your ability to parse and work with these ideas.

## Arduino API Functions

In this chapter, we're going to start going into a lot of detail on functions that are provided by the Arduino API. The first book had a lot to do with the bare fundamentals of programming. This is great and all, but we didn't really get too much experience with the Arduino API itself. Our goal now is to get some experience with the numerous functions that are provided by the Arduino interface for programmers to use.

The Arduino API is the rich set of different things that are provided to the hopeful Arduino programmer to give them more options in their programming. The Arduino team has done a fantastic job of providing a full-featured API that gives the programmer a large variety of different things that they can do within the context of their Arduino tinkering.

We're going to be dividing these by section and going into a lot more information on each function, so settle in tight. What you get from this chapter is a full reference on Arduino programming that you can swing back to whenever you need.



## Digital Input and Output

There are a number of functions defined by the Arduino API in order to allow you to work with digital pins. This section is dedicated to those functions, three in particular.

## `pinMode(pin, INPUT - OUTPUT - or INPUT_PULLUP)`

This allows you to specify a given pin and then designate whether that pin will act as an *input* or act as an *output*. Newer Arduino models are able to have pins enabled through pullup resistors using the INPUT\_PULLUP mode.

## Analog Input and Output

On top of the digital pins, you also have your analog pins. These functions are intended to read voltage from a given pin, always between 0 and 5 volts. The upper range will be closer to 1023 while the lower will be closer to 0.

### `analogRead(pin)`

This will read the voltage from a given pin and return it as an integer from 0 to 1023.

### `analogReference(type)`

This will configure the voltage to be used as a reference depending upon the type of your Arduino.

Most of the time, you can specify “type” as DEFAULT or INTERNAL. There are a few cases where you’ll want to specify a different reference voltage.

### `analogWrite(pin, value)`

This will write a given voltage valued from 0 to 1023, with 1023 being 4.99999 volts and 0 being 0 volts.

## Advanced Input and Output

These don’t really fall under either the digital or analog categories, but they’re more advanced input and output categories that will allow you to do more in general with your Arduino board.

### `tone(pin, frequency, OPTIONAL duration)`

This allows you to specify a given frequency and then generate a square wave of that frequency on a given pin.

*noTone(pin)*

This will stop the tone being generated by the tone function.

*pulseIn(pin, value)*

This will allow you to read the pulse of a given pin. If the pin is fluctuating from high to low, then it will return the time in microseconds between the high and low. Because pulses may not be completely even, you can actually specify whether you want it to read the HIGH pulse (the time for the HIGH value to change to LOW) or the LOW pulse (the time for the LOW value to change to HIGH).

*pulseInLong(pin, value)*

This is just like the function before, but instead of returning an integer number of microseconds, it will return a *long* number of microseconds, which essentially offers a much larger time dimension for which you can receive data.

*shiftIn(dataPin, clockPin, bitOrder)*

This will send a byte's worth of data to a given pin, bit by bit. The data pin is the pin where you're going to be sending each bit, the clock pin is the pin which will designate that dataPin has read data, and the bitOrder can be either MSBFIRST or LSBFIRST (Most significant or least significant bit first, respectively.)

*shiftOut(dataPin, clockPin, bitOrder, value)*

This is much like the function before, but it allows you to *send* data to a pin one bit at a time. Everything else is the same, but you can send out data using the *value* argument. The value argument must be of the type *byte*.

Time

These functions are intended to help you in working with time-sensitive things in the Arduino scope.

### *delay(value)*

This allows you to pause your sketch for a certain amount of time specified by the integer *value* in milliseconds.

### *delayMicroseconds(value)*

This is functionally the same as the `delay()` function except for the fact that it uses microseconds instead of milliseconds.

## Math

Believe it or not, programming sometimes involves a lot of math. The math functions in the Arduino API are similar in many ways to those math functions defined by the C math library, but they keep you from having to import any additional mathematical libraries. Even if you don't use much math in your program, you'll still benefit from knowing that these exist because you never know when you might need them.

### *abs(value)*

This function returns the absolute value of a given number or the distance between zero and a given number on a number line.

### `constrain(variant, lowerBound, upperBound)`

This allows you to create a function such that a number will always be within the lower and upper bound.

### `map(number, fromMin, fromMax, toMin, toMax)`

This will map a number from one range to another range.

### `max(number1, number2)`

Will return the highest number of *number1* or *number2*. Simple enough!

`min(number1, number2)`

Pretty much the exact opposite of the `max` function. This will return the lowest of the two numbers.

`pow(base, exponent)`

This will allow you to take a given number and then raise it to an exponent. C, which doesn't have a built in exponential operator, makes great use of this function.

`sq(number)`

This will return the square of a given number. A shorthand for `pow(number, 2)`.

`sqrt(number)`

This will calculate the square root of a given number.

`cos(angle)`

This will compute the cosine of a given angle, with the angle to be specified in radians.

`sin(angle)`

This will compute the sin of a given angle, with the angle to be specified in radians.

`tan(angle)`

This will compute the tangent of a given angle, with the angle to be specified in radians.

## Characters

While they will be rare, it is important that you have a set of functions primed for you to use whenever you're working with character sets.

*isAlpha(character)*

This will return whether or not the character is alphabetical.

*isAlphaNumeric(character)*

This will return whether or not the character is either alphabetic or numeric.

*isAscii()*

This will return whether or not the character is an ASCII character.

*isControl()*

This will return whether or not a character is a control character.

*isDigit()*

This will return whether or not a character is a number.

*isGraph()*

This will return whether or not the character is something that has visual data. A space, for example, does not have visual data.

*isHexadecimalDigit()*

This will return whether or not the character is hexadecimal.

*isLowerCase()*

This will return whether or not the character is lowercase.

*isPrintable()*

This will return whether or not the character can be printed to the console.

*isPunct()*

This will return whether or not the character is a punctuation mark.

*isSpace()*

This will return whether or not the character is a space.

*isUpperCase()*

This will return whether or not the character is in upper case.

*isWhiteSpace()*

This will return whether or not the character is a whitespace character, like a tab, space, or line break.

## Random Numbers

These functions will allow you to create random numbers in your program. Note that computers can never be truly random and spontaneous; all things are based on inputs, and nothing will ever be without these inputs in a computer. As a result, the random function *must* be seeded.

*randomSeed(number)*

This starts the random number generator. You feed a number in, and it starts at some random point within the sequence of the pseudo-random number generator's numerical sequence.

*random(OPTIONAL minimum, maximum)*

This will act as the bounds to your random number generation. The maximum value is the highest random number that you will allow, and the minimum input is the lowest value you will allow. If you don't specify a minimum, then the minimum will be assumed to be 0.

## Bitwise Functions

These functions allow you to work with bits and bytes, which are the smallest pieces of data that a computer will work with. You can, theoretically, work with smaller values (in terms of overall computing power

required), but these are the smallest practical values that you're going to work with while programming for Arduino.

*bit(bitNum)*

This will return the value of a given bit.

*bitClear(variable, bit)*

This will set the given *bit* of a specified numeric *variable* to 0.

*bitRead(variable, bit)*

This will give back the *bit* of a specified numeric *variable*.

*bitSet(variable, bit)*

This will set a given *variable*'s *bit* as position denoted by *bit* to 1.

*bitWrite(variable, bit, 0 or 1)*

This will set the *bit* at the given position within the *variable* to either 0 or 1, depending on what you say.

*highByte(value)*

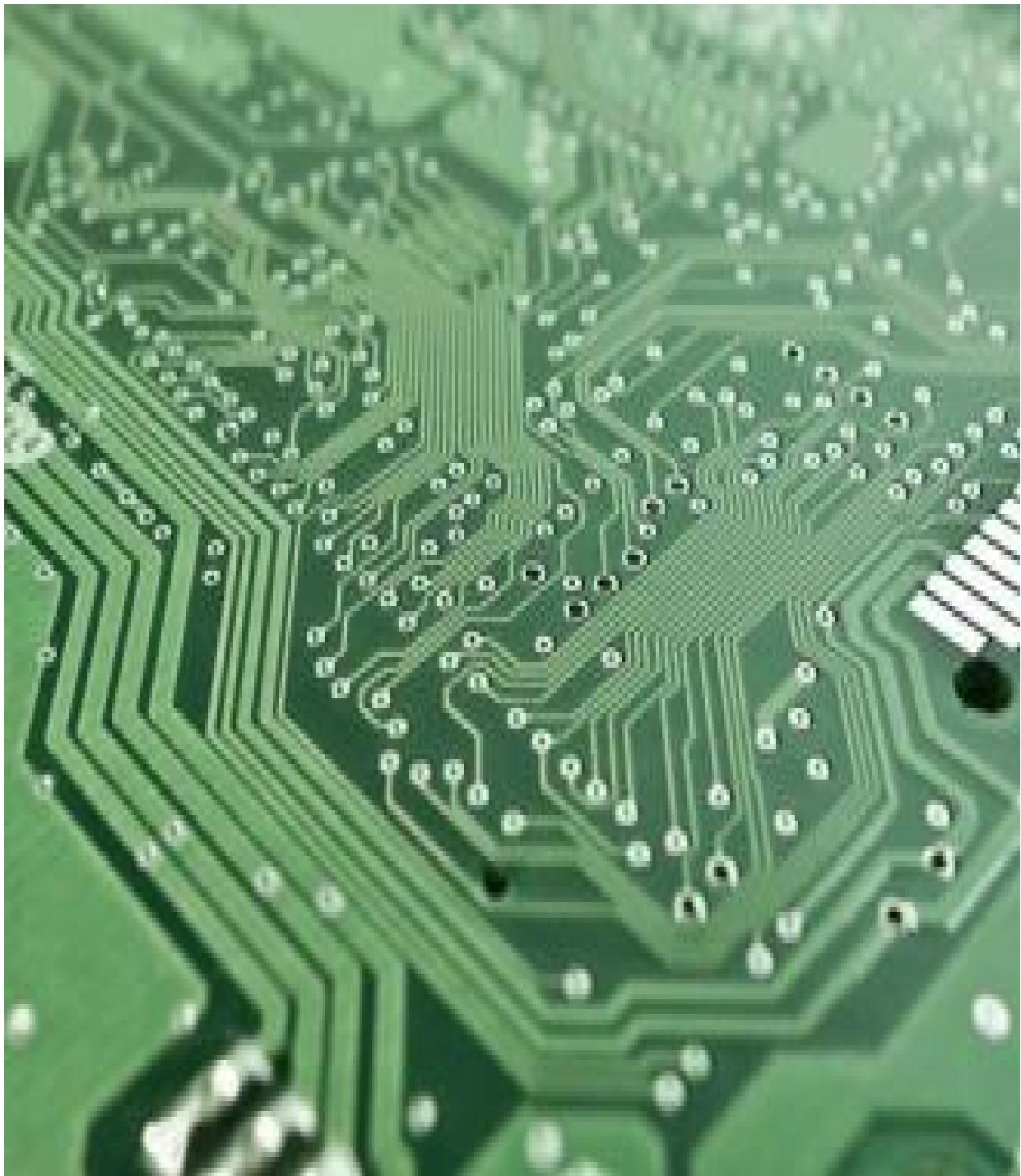
This will return the highest byte of a given value.

*lowByte(value)*

This will return the *lowest* byte of a given value.

## **Using the Stream class (And Working with Strings)**

This deserved its own chapter. While this also deals heavily with using the Arduino API in an effective manner, this is such a broad lesson that we really needed to break it into its own chapter so that we could properly discuss it.



The stream class is a relatively simple concept to grasp. The stream class in itself is based on using reading information from a certain source and using this within your sketch. Because the stream is about reading data, it is necessary that we also talk about working with the keyboard and mouse in this chapter even though these aren't related *intrinsically* to the stream class.

When you're working with data, especially reading in data, you're going to inevitably find times that you're going to need to work with sets of characters like words or sentences or anything of that nature. The idea of *strings* presents you the opportunity to do this.

Strings are basically just sets of character values that are linked together as an array. Therefore, they're contiguous in memory, and the computer sees them as one large and interconnected unit. Working with strings means learning to manipulate these units to the best of your ability.

It is really simple in and of itself. Strings are essentially just *character arrays*. This means that we're technically working with what are called *C-style Strings*, which are basically strings that have a very low level of abstraction. For example, in a lot of more modern and higher-level languages, strings aren't revealed in their character as a character array; they're rather treated as a more abstract object, even if they *are* a character array at their core.

A string is composed of the  $n+1$  characters, where  $n$  is the number of letters within the string in a general sense. So, for example, the size of a string for the word "hello" would be six characters. The reason it is  $n+1$  is that the string ends in a null terminating character, \0, which indicates that the end of the array has been reached and properly terminates it.

You can define a string in the same way that you would an array. You can also make them bigger than the string that they're going to contain. When you define an array, you may give it a value right off the bat, but you can also just define their size and expand them at a later point. This also makes strings, in one manner or another, dynamic and able to be changed at a later point in the program by rewriting the data within the string.

This information is of great use to you as a programmer because strings are a fundamental part of any sort of program that handles information, especially those which handle file input and output.

We've already spent a bit of time rehashing information from the book prior, but just for the sake of clarity, we're going to go ahead and define a string:

```
char myString[6] = "hello";
```

You can then refer to this entire string at a later point by the name of the character. Most of the data that is worked with by the Arduino will be worked with in terms of bytes, and most actual textual data will be worked with in terms of C strings because characters are tremendously easy to parse.

It is important that we cover all of this so that we can actually develop an idea of how to treat strings in the context of Arduino programming alongside everything that we're going to be working on through this book.

## Serial

While you can't necessarily implement the stream class itself, you can implement its *derivatives*, and this is where you start to find a whole lot of utility. The serial class is an extension of the stream class that allows the Arduino board to communicate with other devices such as a computer.

Serial is enacted through both the Serial port on the Arduino as well as the USB link to the computer. In this section, we're going to be outlining all of the different functions which make up the Serial class so that you can make the absolute most of this invaluable resource.

`if (Serial)`

*Serial.begin(rate)*

You're already familiar with this function. It allows you to start the serial transmission of data. You can specify the specific rate of data transmission in bits per second.

*Serial.end()*

This allows you to end serial communication. You can later restart the serial communication by calling the `Serial.begin()` function if you wish. While the serial communication is disabled, you can use the serial pins for generalized entry and exit of data.

### `Serial.find(string)`

This will search for the given string within the data provided by the Serial. If the string is found, the method will return true. If the string is not found, the method will return false.

### `Serial.findUntil(string, OPTIONAL endString)`

This will look for the string within the serial buffer until either the string is found or a specified terminating string is found. If the target string is found, then the method will return true. If the terminating string is found or if the method times out, it will return false.

### `Serial.flush()`

This will allow you to halt processes until all data being sent to the serial has been sent. Straightforward!

### `Serial.parseFloat()`

This will return the first floating point number to be provided by the serial stream. It will be brought to an end by any character that isn't a floating point.

### `Serial.parseInt()`

This will return the first integer number to be provided by the serial stream. It will be brought to an end by the first character that isn't a digit.

### `Serial.peek()`

This will return the very next character to be imported by the serial buffer. However, it will not remove the character from the buffer. This makes it fundamentally different from the `Serial.read()` method we'll be getting to momentarily. This means that you can simply see what character is coming next.

### `Serial.print(value, OPTIONAL format)`

You can specify the format, optionally. Otherwise, integers will print as decimals by default; floats will print to two decimal places by default, and so forth.

You can send characters or strings as is to the print statement and it will print them without any issue.

`Serial.println(value, OPTIONAL format)`

This will allow you to print out values just like you would with the normal print method

`Serial.read()`

This will read in the data which is coming in through the serial port. Simple enough! It is added to an incoming stream of serial data called the serial buffer. When you read from this buffer, the information is destroyed, so be sure to save the data to a variable if you need to reuse it at some point.

`Serial.readBytes(serialBuffer, numberOfBytes)`

This will read in characters from the serial port to a buffer. You can determine the number of bytes that are to be read. Your buffer must be either a char array or a byte array.

`Serial.readBytesUntil(terminatorCharacter, serialBuffer, numberOfBytes)`

This will read in characters from the serial either until the given number of bytes has been read or until a given terminating character has been read. In either case, the method will terminate.

`Serial.write()`

This will write data to the serial port; however, this particular method only sends binary data to the serial port. If you need to send ASCII data, you should use the print method instead.

`Serial.serialEvent()`

Whenever data comes to be available for use by the serial port, this function will be called. You can then use the `Serial.read()` function in order to read in data from the serial port.

With that, we've covered a lot of the particular functions related to the serial class and how it pertains to programming with the Arduino API. The next thing that we're going to need to work with is the Ethernet class.

## User Defined Functions

One of the ways you can help keep your code neat, organized, and modular (reusable) is to use functions in your code. Additionally, they help make your code smaller by making certain sections reusable. Functions are like tools that were created to serve a particular function, as the name suggests.

While we have already encountered a few user-defined functions, we will cover them in greater detail now and explain some of the features we may have glossed over when we encountered them last time. Let's look at the declaration of a function now:

```
float employeeEarnings (float hoursWorked, float payRate) {
```

float result;// this will be the value we return when this function is called. It should match the datatype before our function name.

```
    result = hoursWorked * payRate
```

return result// return tells the function to send a value back once to where it was called

```
}
```

This function clearly takes two arguments, hoursWorked, and payRate, both of which are ‘floats.’ It does some simple math on them and then returns a float as a value. Return means to terminate the function and send back whatever value is placed after the word return, usually a variable, as the result of some calculations.

Let's see us call this function now to get an employee's earnings:

```
void loop () {
```

```
    floathoursWorked = 37.5;
```

```
    float payRate= 18.50;
```

```
    float result = employeeEarnings (hoursWorked, payRate)
```

```
// result will be 693.75
```

First, the function must be declared outside of any other functions. This means you need to write the code for the function you are creating outside of either `setup()` or `loop()`, or any other user-defined function.

Let's see another example that sample sketch that could be used to smooth sensor readings:

```
int sensorSmoothing (analogPin) {  
    int sensorValue = 0;  
  
    for (int index = 0; index < 5, index++)  
        digitalWrite(LED_BUILTIN, HIGH); //Turn on LED for  
                                         smoothing  
  
    sensorValue = sensorValue + analogRead(analogPin)  
  
    delay(100)// 100 millisecond delay between samples  
  
}  
  
digitalWrite(LED_BUILTIN, LOW);//turn off LED  
  
sensorValue = sensorValue / 5// average the values over five samples  
  
return sensorValue;  
  
}
```

This kind of function can be used for smoothing the data input of many sensors if they are prone to jittery inputs. This will average the samples to give a more consistent flow of data. You can see that this code is very similar to our last example:

```
void loop () {
```

```
int sensorPin = 0;// analog pin 0  
int sensorValue = sensorSmoothing (sensorPin);  
}
```

Here, when we try to initialize our sensorValue variable it will call the sensorSmoothing() function on analog pin 0, and return the average result over five samples)

Functions do not always need to have parameters or return variables either. Sometimes functions can return no value and have no parameters. All they do is execute a few lines of code and then terminate bringing the compiler back to place in the code they were called.

## Conclusion

The next step is to get out there and start making your own sketches! Go to your local hobby store to get some ideas or go to the community to see what new projects you might want to try. After you have an idea where you might want to go next, (robots are pretty fun!) join the community! Seriously, it is a lot of fun to build projects with friends and compare them with each other. If you feel like you don't know where to start, don't worry! There are many online sources that share coding and techniques to improve your game. Many online sites also have forums specifically tailored to helping people like you learn and show off what they have done. It is also a fantastic way to learn and grow as a hobbyist.

If you want to get started but are feeling strapped for cash, there are options. Like we've said above, there are fairly cheap modules for purchase on the Arduino site and others. Also, cheaper programming languages are available, and some are even free. Learning these languages are actually easier than you would think. If you know one programming language, the rest are easier to understand and write. There are also books at your local library that will help you learn how to code. Many libraries offer interlibrary loans, which means that you can learn about programming from books from all around you! Best of all, learning from a book from the library is absolutely free! If you have any questions about this, remember that you can go online and find others who have worked with Arduino and know how to get you started.

Arduino board can be programmed to light or fade a LED etc. The syntax used in Arduino programming is similar to C++. If you are good at C++, then programming in Arduino will be easy for you. The variables in Arduino are initialized within the setup() function. The loop() section has the block to be run repeatedly. When working with Arduino pins, you must specify the pin you need to work with. The pins are normally identified with numbers as each has a unique number. After getting the board, you have to setup Arduino IDE on your computer. This is where you will be writing your Arduino code before uploading bit to the board. Arduino code is commonly known as a sketch. You must get a source of power. However,

some board types must be configured to allow power to be drawn from a computer. The effect of a sketch on the board will be seen after uploading it to the board, in which one has to click the Upload button.

There are many sensors and additions to each Arduino, so make sure you check out which you would like to employ. The great part of owning an Arduino is that you'll get the chance to try many experiments. You're not just limited to what the sensors can read, either. Come up with some ways on your own to make the machines work for you. Try programming simple requests first—like setting up blinking lights or figuring out how Arduino can monitor inputs—and see what you can do from there. I have included many Arduino codes for you to use, but feel free to find some of your own! There are many guides for help.

You can also check out some more advanced concepts we didn't have a chance to touch on here such as headers, classes, changing the clock speed for the chip, adding cores, adding libraries, there is so much that you can do with this chip, it really is incredible. Pick a direction that interests you and see where it takes you. I hope that this guide has offered you some small inspiration to go out there and try new things and see what your sketch designing skills are capable of.

Finally, thank you for finishing this book! Arduinos are a fun way to get started on your programming journey. Since you've purchased this book, we hope you've grown, and if you found this book useful in any way, a review on Amazon is always appreciated!

## References

Arduino Reference. (2019). Retrieved from <https://www.arduino.cc/reference/en/language/structure/comparison-operators/lessthan/>

What is an Arduino? - learn.sparkfun.com. (2019). Retrieved from <https://learn.sparkfun.com/tutorials/what-is-an-arduino/all#the-arduino-family>

# C#

*The Ultimate Beginner's Guide to Learn C#  
Programming Step by Step*

**Ryan Turner**

**© Copyright 2019 – Ryan Turner  
All rights reserved.**

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

**Legal Notice:**

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

**Table of Contents**

[Introduction](#)

[Chapter 1: What is C#](#)

[Chapter 2: Detailed Overview](#)

[Environment](#)

[Program Structure and Basic Syntax](#)

[Chapter 3: Demystifying Data Types](#)

[Chapter 4: Working with Variables](#)

[Chapter 5: What is Type Conversion?](#)

[Chapter 6: The Need for Operators](#)

[Chapter 7: The Conditional Statements](#)

[Chapter 8: Loops](#)

[Chapter 9: C# Methods](#)

[Chapter 10: The Array World](#)

[Chapter 11: Classes](#)

[Chapter 12: Structure](#)

[Chapter 13: Encapsulation](#)

[Chapter 14: Inheritance](#)

[Chapter 15: Polymorphism](#)

[Chapter 16: Regular Expressions](#)

[Chapter 17: The Process of Handling Exceptions](#)

[Chapter 18: File I/O](#)

[Chapter 19: Delegates](#)

[Chapter 20: Multithreading in C#](#)

[Chapter 21: Event](#)

[Chapter 22: Hints and Important Resources](#)

[Conclusion](#)

[Bibliography](#)

# Introduction

Although there is nothing easy about learning a new language of any sort, the aim of this book is to allow you to pick up and understand as quickly and easily as possible. The goal of this is to provide you with complete and comprehensive skills that are able to help you retain important knowledge and guide your programming and coding experience in C#. In order for you to learn in the most efficient manner, I hope that you will also do the practice that will reinforce the lessons we explore in this book.

The book will cover a basic description of what C# is, and finally give you some helpful hints and resources to reinforce any subjects or skills where you may need a little more assistance.

This language is one that will mirror some components of other computer programming languages—it is almost invaluable experience if you are familiar with other languages themselves. Although this book is not quite just a beginner's manual that introduces you to the art of programming, it will reiterate and acknowledge a lot of these skills to help you achieve your maximum programming results in the quickest amount of time you can. Keep in mind that this language provides the programmer with many abilities, and this book may refer to things in terms of gameplay programming I hope you are also familiar with.

Although I provide you with the information and some basic skills to practice, it is up to you to put these skills into practice. I highly recommend you find a strict routine to abide by that involves research, study of the material and practice in your own virtual environment that familiarizes you with these skills. As commonly known, you can guide a horse to the water but you can't force it to drink!

I can give you the knowledge and resources but it will be up to you to take on the challenge of synthesizing and using them to create your own programming and codes.

With an object-oriented and structured language such as C#, I think you will find it easy to learn and user-friendly. This high-level language builds on and, in some ways, complements the languages of C, C++ and even Java. However, even if you are a master of these languages, this new task will have just a steep of a learning curve as any of the other programming languages do! Throughout this process, I encourage you to use your own knowledge and instincts to apply the concepts and strategies in this book. Remember, everyone is a beginner before they are a master!

# Chapter 1: What is C#

You have come across and purchased this book in order to learn how you can become a C# programmer.

Throughout this book, you will learn basic rules of how to apply C# within your own programming projects. C# (pronounced as “see sharp”) is an object-oriented computer programming language that was both designed and developed by Microsoft.

This is a fairly new language and is considered to be very functional and object-oriented in nature. Although C# is most certainly not the same language as C or C++, there are definitely some similarities between the languages and it could even be considered helpful to have an insight or knowledge and experience in either or both of those languages.

Just as with all programming languages, having knowledge of how they work on a basic and functional level is helpful. All languages have one thing in common, they are working towards telling the computer what and how to set up systems, databases or how and when to run and execute programs.

This particular language has roots stemming from a programmer, Anders Hejlsberg, who was a member of the development team of .Net Framework for Microsoft<sup>[97](#)</sup>. Although some people tend to criticize C# as being somewhat of a “rip off” of Java, you will see throughout the course of this learning manual that (if you have a knowledge of Java that is) there are distinct differences in the content and structure of the language that make it different.

With a design that was formulated for an environment that has the ability to use high-level languages and consists of executable code (also referred to as Common Language Infrastructure or CLI) it appeals to many who are interested in this easy-to-learn language.

It boasts efficiency as one of its objects, so there are quite a few advantages to using it. Throughout this book, I will introduce you to how this object

and component-oriented language can be easily acquired as a new language for you to use.

Along with being a structured language, you can count on efficient results from your programming and coding within the C# language. I will assist you with optimizing your performance, in order to obtain your desired output from the work you put into programming and coding in this language.

Although C# was developed by using the Microsoft platform, the language itself is not just limited to it. Another great concept of C# is that this versatility allows you to take the object-oriented language to different platforms to create multidimensional and available programs and software.

As I am sure you are familiar with other languages, certain aspects of languages benefit certain desired results. There are many benefits, that will be covered in this guide to learning C#, that assist a user with portability, typing, meta programming, property, memory access and methods and functions—among an array of other positive attributes we will discuss and show examples of in the coming chapters.

This language has only been around, in official versions, since 2002, and therefore could be considered a newer programming language. However, do not let that lead to you believe it cannot and does not compete with other languages, such as Java, that have been around a little longer.

Developers of this language really aimed to improve upon and add different features, initially designed specifically for the .Net framework but now have been expanded, and were successful in this with many features—one many tend to like is the Syntax used.

There is a possibility for this language to be used to focus on gameplay. It is important to gamers that, not only is the game interesting and holds one's attention, they also want the experience itself to be enjoyable.

As you are probably somewhat of a programmer yourself (or at least appreciate the capabilities of computers), after reading this book I expect

you will have a great understanding of just how hard gaming programmers work in order to create the experience you have as an end result.

Although I do not think reading this book alone will allow you to create a masterful game (you need a lot of practice as well!)—I do believe that you will understand concepts to a greater degree that are essential when you are dealing with the many components of graphic programming and design. Keep in mind, throughout the book, that you are creating and fine-tuning a potentially endless world through your coding and programming skills in this language. The object-oriented component really helps to reinforce this idea.

In the coming chapters, you will learn both how and why C# will be very useful to you and how you can easily pick up and become fluent in all the basics within less than a month.

With a standard library that rivals any of the other languages, by the end of this book you will know how to find the source code you may need and execute it to produce fairly consistent and exquisite results. Another positive feature of this language is the Boolean conditions, which are favored by many, along with the Windows integration that allows you to access one of the most popular operating systems with ease.

As a final note on the very simplistic explanation of what C# is in this initial chapter, I just want to note a few general concepts of programming and coding that are good to keep in mind throughout this process.

First of all, remember that computers do not have intuition. This means that they will execute tasks exactly as you tell them. A computer will not infer a message from your code, as a human might. Always keep this in mind when you are having any struggle.

Also remember that a simple mistake in spacing or a wrong letter can cause a whole lot of headache. If you come to, what may seem like, an impossible impasse—perhaps you need to take a quick break and come back to look at your coding again.

Sometimes we cannot see our own mistakes when we have been staring at a screen for a long period of time.

Finally, remember that you are not the first person to have a struggle with this language and will certainly not be the last. The last chapter of this book will be a guide to further helpful resources and hints as to address any struggles I foresee you having in the future as you begin to learn this language.

This book sets up reasonable expectations for you to learn C#. I encourage you to try to learn in that exact manner. It is important for you to not overwhelm yourself with too much content in a short period of time.

Conversely, don't allow too much time to lapse before coming back to your work. You may forget subtle nuances that may drive you crazy. Allow yourself to have reasonable expectations, which this book is designed to do.

Lastly, enjoy yourself! Allow yourself to take pride in the new skill you are learning and get ready for a crash course in basic C# mastery!

## Chapter 2: Detailed Overview

Since you are a beginner, we want to focus on building a strong foundation that gives you a full picture of the basics of C#.

Essentially, by the end of the first week you will have a firm grasp on the environment in C#, the program structure, basic syntax and data types, type conversion, variables, constants and operators. If you are unfamiliar with the terminology or methodology used, I will try to give a quick, basic review of the terms. If this does not satisfy jogging your memory, you may need to find a quick refresher from Google or another resource. I would also like to recommend a free interactive shell program you can try out code on - [https://www.tutorialspoint.com/compile\\_csharp\\_online.php](https://www.tutorialspoint.com/compile_csharp_online.php), this should help you complete any training exercises included in the book.

### Environment

As seen, C# was designed to be integrated with Microsoft and .Net Framework by the developers. As one would guess, this is a large influence on the environment you find within C#. An important part of understanding C# comes from the understanding you have of how the two interact with each other. First, here is a brief synopsis of .Net framework to help you better understand how the two interact.

As the latter part of the name framework suggests, the .Net framework is a platform in which one can run applications. The .Net framework is a platform for software development developed by Microsoft. The framework was created for development of applications that can run on the Windows platform. The first version was announced in 2002 and given the name .Net framework 1.0. Since then, a lot of updates have been made on the framework up-to-date.

We can use the .Net Framework for creation of both web-based and form-based applications. We can also use it to develop web services. Other than C#, the .Net framework also supports the visual basic programming language. This means that the programmer has the option of choosing the language in which to develop their application in.

There are three different types of applications that the framework can assist you with writing—web services, web applications and windows applications. When one is using the framework applications, it can actually be used by a variety of programming languages, as it is a multi-platform application. The multitude of languages, from Virtual Basic, Jscript, C++ and COBOL, also have the ability to communicate with each other within the framework. Another great attribute of this framework is the extensive code library that can be used by a language, for our purposes we will be talking about C#.

One of the great things about any language or framework is the potential for a library of codes you can utilize. For the .Net framework and C#, there is a huge library that you can access and utilize for your coding projects. One advisory statement I always give people is that although these libraries are excellent resources, be sure you understand the hows and whys of the coding itself to ensure that you are able to fix an issue, should it arise.

Microsoft itself also offers a great tool for its users in order to assist their development in their C# programming projects.

There are two excellent ones that the company even offers for free, great resources for writing any kind of application, from simple to complex. The names of the two I would recommend (because of their free price!) are Visual C# 2010 Express and Visual Web Developer.

If these options are not something you feel comfortable with or enjoy using, you can always use a text editor in order to write the source code you may need. I recommend the other options because I find them to be a step up in helpfulness from a simple code editor. However, lots of people prefer that method—it's really up to you and the comfort level you feel with a program. Test them out and see which methodology works better for you.

You can download the two Microsoft programs I spoke up from a website called Microsoft Visual Studio.

Finally, you can also run C# on Linux or Mac OS, which is a great ability to code across different platforms. However, in order to do so, you obviously need a non-Microsoft framework to do so on your different operating

system. I recommend “mono”—which is an open source option for use on both Linux and Mac.

With the basics of the environment, go download or open the programs you would like to get familiar with to start your coding process. Play around with the software—or you can sleep on your new knowledge and you can pick up your project for the next phase!

### **IDE (Integrated Development Environment)**

An IDE is a program that allows you to write your program. Most .Net developers prefer using Visual Studio Community as the IDE. It allows you to create, debug and run your applications. This IDE can be used for development of both web-based and form-based applications. To use this IDE, you must download and install it on your computer. You can find it in the URL given below:

<https://www.visualstudio.com/downloads/>

You are allowed to choose between the Visual Studio Community Forum, which is free, and Visual Studio Professional Edition, which comes with a 30 day trial period, and after the expiry of that period, you must pay for it.

Once the download is complete, double click the setup file to begin the installation of visual studio. Follow the on-screen instructions until the installation of visual studio is complete.

After that, you will be ready to start creating your own C# applications.

### **Program Structure and Basic Syntax**

Now that we have established a little about how the environment was created for C#, let’s look at the actual structure and how you will interact with programs through the syntax of C#.

When we are looking at how C# is structured, we are looking at 7 basic components that make up the program. These are that there are class methods and attributes, namespace declaration, a class, a main method, comments and statements and expressions. Now, what do these look like for

you in terms of your programming and coding? Why don't we take this "Hello World" example and break it down into parts to see this in action.

```
using System98;  
namespace HelloWorldApplication  
{  
    class HelloWorld  
    {  
        static void Main(string[ ] args )  
        {  
            /* my first program working with the wonderful C# */  
            Console.WriteLine("Hello World");  
            Console.ReadKey();  
        }  
    }  
}
```



99

So hopefully this looks somewhat familiar to you and, although you may not quite understand the meaning, you know all the symbols and spaces are important.

When we are using C#, the first line is always going to consist of using System, although most of the time you will see more than one statement with the first key word being using.

The next line in C# will be used as a declaration of the namespace. In case you are unfamiliar, we should think of the terminology of namespace as a descriptor of the specific group or collection of classes.

In our Hello World example, we are looking at that phrase as the class (HelloWorld). Following the name space line, you will come to the class declaration. The class is going to inform you of the method of the class.

It is a typical characteristic of methods that there are multiple within one singular class. It is important to be familiar with the methods of a given class because you can then be more aware of the behavior of the class you are dealing with. In the Hello World example, we are looking only at one single term, or main method.

Each class has a main method that you can find in the line following the class declaration. Essentially, when you are looking at this line you can expect the main method to tell you what is going to happen when the code is

executed. This point is often also referred to as the entry point in C# programs. When you see the `/*...*/` in the following line, you are looking at code that will be ignored by the actual compiler and instead is added in the program under the comments.

The last two phrases we are looking at in this example are `Console.WriteLine("Hello World")` and `Console.ReadKey()`. These lines are telling the system to both display your message and control how and when the program runs and closes. These are obviously both important parts of how and why the program would work and function and you want to make sure of a couple of key components that could affect your coding, just due to similar grammatical type errors you may find in the English language.

To run the code, just click the Run/Execute button or press the F5 key and the project will be executed. The code should return:



The code begins with the *using* statement. What this statement does is that it adds the *System* namespace into the program. C# programs normally have multiple *using* statements.

In the next line, we have a *namespace* declaration. A namespace is simply a collection of many classes. Above, we have the *HelloWorld* namespace in which we have the *Hello* class.

In the next line, we have the class declaration. The *Hello* class has method and data definitions that the program will be using. A class normally has numerous methods. The purpose of a method is to state the behavior of a class. However, our *Hello* class above only has one method, the *Main* method.

In the next line of code, we have the definition of the *Main* method, which marks the entry point for all for C# programs. It is the Main method that

states what happens after execution of a class.

Next, we have the `/* ... */` line. This is a comment, and the C# compiler will ignore the line. The line is only meant to increase the readability of the code by human readers.

Next, we have the `Console.WriteLine(...)` line. `WriteLine` is a method that belongs to the `Console` class that is defined in the `System` namespace. This line will print the *Hello World!* message on the screen.

Lastly, we have the `Console.ReadKey();` statement. This line is for VS.NET users. The line will make the program to wait for a key press and prevent the screen from running then closing quickly once the program has been launched from the Visual Studio .NET.

Other than running the program from the Visual Studio, it is possible for you to run a C# program from the command prompt of the operating system. To do this, follow these sequences of steps:

- Open your text editor then add the code given above to it. An example of a text editor is Notepad.
- Save the file with the name *helloworld.cs*
- Launch the command prompt of the operating system then navigate to the directory you have saved the file.
- Type the command `csc helloworld.cs` on the prompt then press the enter key to compile the code.
- If the code has no errors, the command prompt will take you to the next line then generates the file *helloworld.exe*. This file has executable code.
- Type *hello world* on the prompt to execute the program.
- The output *Hello World!* Will be printed on the screen.

You want to make sure that you note a subtle difference (if you are familiar with Java) that the class name and program file name can be different. Also, C# is both case-sensitive and, rather than ending sentences with a period or other notation, they are ended using a semicolon. Also, always remember that the main method is the point at which the program execution starts.

These four components are good to note in order to avoid annoying and tedious problem-solving when mistakes occur with your coding.

Play around with the above example within the virtual environment. Now we will discuss a little about the syntax of C# and how some of the concepts we have talked about with the structure really develop and exemplify the syntax. For starters, the `/*...*/` that allows you to comment on a program is an important feature to remember.

In addition to this, it is good to remember that when naming a class, the first character in the name can only be a letter, not a numerical digit. Although you can make up the class itself following the initial digit with letters and numbers, always remember this and that symbols are always not allowable in the naming of a class. Finally, the name cannot contain a C# keyword. There are two different categories of keywords, contextual and reserved, below I will list them all in their respective categories.

**Contextual Keywords**— get, partial, from orderby, let, dynamic, set, join, descending, ascending, into select, alias, group, remove, global and add.

**Reserved keywords**— case, decimal, event, for, int, new, continue, enum, in, float, byte, namespace, in, long, else, fixed, const, break, finally, lock, implicit, double, class, is, if, bool, do, false, checked, base, goto, internal, extern, delegate, as, char, foreach, explicit, interface, default, abstract, catch, try, internal, is, interface, sealed, null, switch, object, short, protected, this, operator, sizeof, is, public, throw, out, readonly, lock, true, long, red, out, static, override, namespace, string, return, typeof, params, new, struct, uint, sbyte, volatile, unchecked, throw, stakalloc, ref, private, while, void, virtual and sealed.

Important Note: Make sure that all of your programs have formatted codes. This way, you can guarantee the readability of your codes.

**A Breakdown of Different keywords<sup>100</sup> in C#**

Keywords are special predefined reserved words and are each assigned with a unique meaning. These keywords can be organized into categories useful for better understanding. Below is a list of keywords categorized into different types.

### **1) Keywords for class, method, field and property**

- abstract
- extern
- internal
- new
- const
- override
- protected
- private
- public
- sealed
- readonly
- static
- virtual
- void

### **2) Keywords for type conversions**

- explicit
- implicit
- as
- is
- operator
- sizeof
- typeid

### **3) Keywords useful for program flow control**

- if
- else
- for
- foreach
- in
- case
- break

- continue
- return
- while
- goto
- default
- do
- switch

#### **4) Keywords used for built in types and enumerations**

- bool
- char
- class
- byte
- decimal
- enum
- double
- float
- interface
- long
- int
- object
- sbyte
- short
- string
- uint
- struct
- ulong
- ushort

#### **5) Keywords used for exception handling**

- try
- catch
- throw
- finally
- checked
- unchecked

#### **6) Keywords used as literals, method passing parameters**

- true
- false
- null
- this
- value
- out
- params
- ref

## 7) Keywords useful in function pointers, object allocation, unmanaged code

- delegate
- event
- new
- stackalloc
- unsafe

### ***How to Format Codes Properly***

Some rules to follow when formatting your codes include:

- You may indent methods within the class definition. C# allows you to use any number of tab characters (i.e. the character you'll get after hitting the Tab key of your keyboard) when formatting source codes.
- You should indent the method's contents within its own definition.
- Place the opening brace (i.e. {) right under the class or method to which it belongs. Also, it would be best if it will be the only character on its line.
- Make sure that the closing brace (i.e. }) is vertically aligned with the opening brace.
- The names of your class should begin with an uppercase letter.
- The names of your variables should start with a lowercase letter.

- The names of your methods should begin with an uppercase letter.
- Indent codes that are written inside another code.

## Class Names

In C#, each program has one or more class definitions. Programmers often store each class definition inside a separate file that corresponds to the class's name. Also, the extension of these files should be “.cs”. You can compile different classes into a single file without experiencing any technical problems. However, you'll have a hard time navigating your codes.

## More Information about C#

The C# programming language is advanced, modern, multipurpose, and object-oriented. It has some similarities with other languages such as C, C++, and Java. The main difference is that C# offers easier programming and simplified syntaxes.

A C# program has one or more .cs files, which hold data types and class definitions. You need to compile those .cs files to get executable codes (i.e. files that end with .dll or .exe). For instance, if you will compile the HelloCSharp program, you'll get an executable file named “HelloCSharp.exe”.

## The Things You Need to Create C# Programs

Before you can write programs using C#, you need two important things—a text editor (e.g. Notepad) and the .NET framework. The text editor will help you in writing and editing codes. The .NET framework, on the other hand, will help you in compiling and executing your programs.

### **What is .NET?**

Basically, .NET is a framework designed for the development and execution of computer programs. Most of the modern Windows systems have .NET as a built-in framework. Thus, you won't have to install any software onto your computer if you are using Microsoft's modern operating systems.

If you are using an old Windows OS, you need to download the .NET framework first. You may go to this [site101](#) and get the most recent version of

.NET.

Important Note: Make sure that your computer has .NET before reading the rest of this book. Otherwise, you'll have problems compiling and executing the sample programs that you'll see later.

### ***Text Editors***

You should use a text editor to write, edit, and save source codes. You can use any text editor installed on your computer. If you don't want to download additional programs, you may simply use "Notepad" (i.e. the pre-installed text editor of Windows computers).

### **Compiling and Executing Programs**

The actions you need to undertake include:

1. Generate a ".cs" file and name it as "HelloCSharp".
2. Write the source code inside that .cs file.
3. Compile the code to get a .exe file.
4. Run the resulting program.

Turn on your computer and let's start writing some codes.

Important Note: This eBook assumes that you are using a Windows computer.

# Chapter 3: Demystifying Data Types

Now let's talk about data and the different types you will see in C#, which we have three basic categories of. These categories are value, pointer and reference types. If you are familiar with programming, you are definitely familiar with the different types of values that can be associated with data types.

## Attributes

Data types have the following attributes:

- Name (e.g. int, char, bool, etc.)
- Size (e.g. 1 byte)
- Default Value (e.g. 1, 2, 3, etc.)

## The Different Types of Data in C#

C# supports the following data types<sup>[102](#)</sup>:

- Boolean
- Characters
- Objects
- Floating-Point
- Decimals
- Integers
- Strings

Since these data types are pre-installed into the C# language, programmers refer to them as “primitive types”.

Value types of data are, as the name would suggest, data types that consist of the value themselves. These are terms like Boolean data, byte or int. A good test to run within your virtual environment is to get the size of a given data type using the term `sizeof(type)`, only replace type with the data type you wish to know the size of. A complete list of the value data types are

decimal, double, bool, float, char, byte, long, sbyte, short, uint, int, ushort and ulong.

Within the reference type of data, we are looking at (you probably guessed it!) a reference to a type of data. These types of data are able to tell you where memory is located. There can even be multiple references to multiple variables. His type of data, we are looking at string, dynamic and object as our key words to describe the reference data types.

The last type of data, pointer, points to a different variable in order to direct you to the appropriate location. In order to reinforce that you do indeed know what a pointer is, here is a generic sample of what a declaration of a pointer variable should look like—

```
type *var-name;
```

Go ahead and play around with the variables in your own dictionary and see what kind of output you can get for these. There is one other important phrase I think you should be aware of in your programming and coding. This is when you use the /unsafe command—

```
csc /unsafe darkprog.cs
```

In the above example, the program we would be specifying as unsafe is “darkprog.cs.

C# is a strongly typed language; hence it expects you to state the data type any time you are declaring a variable. Let us explore some of the common data types and how they work:

1. bool- this is a simple data type. It takes 2 values only, True or False. You need to use “bool” (i.e. a C# keyword) to declare this data type. Boolean values only result in “true” or “false”. In the C# programming language, Boolean values are automatically set as “false”. Programmers use this data type to store the result/s of logical statements. It is highly applicable when using logical operators like *if* statement.

2. int- this stands for *integer*. It is a data type for storing numbers with no decimal values. It marks the most popular data type for numbers. Integers also have several data types within C#, based on the size of the number that is to be stored.
3. string- this is used for storage of text, which is a sequence of characters. C# strings are immutable, meaning that you cannot change a string once it has been created. If you use a method that changes a string, the string will not be changed but instead, a new string will be returned.
4. char- this is used for storage of a single character. You need to use “char” (i.e. another C# keyword) to declare this type. Write your “char” values within a pair of apostrophes (e.g. “I”).
5. float- this is a data type used for storage of numbers that have decimal values in them.

## *Objects*

Programmers regard objects as the most special type of data in the C# language. Basically, objects serve as the parent of other data types within .NET. This data type, which requires the “object” keyword during its declaration, may accept any value from other data types. You may think of objects as addresses that point to certain parts of your computer’s memory.

## *Real Type – Floating Points*

In the C# language, “real type” refers to the actual numbers you’ve learned in mathematics. A floating-point real data type can either be “float” or “double”.

1. Float – Programmers call this subtype “single-precision real integer”. The default value of a float number is 0.0F (the “f” at the end is not case-sensitive). The letter “f” indicates that the value is a “float”.

2. Double – Programmers use the term “double-precision real integer” when referring to this subtype. Its default value is 0.0D (i.e. the letter “d” at the end is not case-sensitive.). C# automatically tags real numbers as “double” so you don’t really need to add “d” at the end of the value.

### *Real Type – Decimals*

This programming language supports “decimal arithmetic”, a mathematical approach that uses decimal values to represent numbers. This approach preserves its accuracy regardless of the values it works on.

C# uses the “decimal” data type to represent this kind of real number. This data type is 128 bits long and is accurate up to the 29<sup>th</sup> decimal value. Because of its excellent accuracy, these are used by programmers for financial computations (e.g. payments, taxes, interest, etc.).

### *Integers*

This data type consists of 8 subtypes, which are:

- SBYTE – An sbyte is a signed integer that is 8 bits long. That means it can contain up to 256 values (i.e.  $2^8$ ). Additionally, sbytes can be either negative or positive.
- BYTE – This subtype contains unsigned integers that are 8 bits long. It is almost identical to sbyte. The only difference is that byte can never be negative.
- SHORT – These integers are signed and 16-bits long.
- USHORT – These are signed integers whose length is 16 bits.
- INT – This is probably the most popular data type in C#. Programmers use this data type because it is perfectly compatible with 32-bit processors and big enough for typical computations. It is a signed integer whose length is 32 bits.

- **UINT** – This 32-bit integer is unsigned. Thus, it can only be positive.
- **LONG** – This integer is 64 bits long. It can be either positive or negative.
- **ULONG** – This is similar to “long” integers. The only difference is that “ulong” can only assume positive values.

### *Strings*

A string is a set or sequence of characters. C# requires you to use “string” (i.e. a keyword) when declaring values that belong to this data type. You have to write each string between a pair of quotation marks. This language allows you to perform various operations on strings (e.g. concatenation, character replacement, character search, etc.).

The `sizeof()` method allows us to know the size of a variable or data type. The size is returned in the form of bytes. Consider the following example<sup>[103](#)</sup>:

```
using System;
namespace TypeApp {
    class IntType {
        static void Main(string[] args) {
            Console.WriteLine("Size of int: {0}", sizeof(int));
            Console.ReadLine();
        }
    }
}
```

The code should return:



Which means that an integer takes a storage size of 4 bytes?

# Chapter 4: Working with Variables

You can use a variable to store, retrieve, and modify changeable data. As a programmer, you need to use variables<sup>104</sup> in storing and processing information.

## The Characteristics of a Variable

A variable has the following characteristics:

- Name (also known as “Identifier”)
- Data Type
- Value (i.e. the information you want to store).

Variables are sections of computer memory that have a name. They store values and allow a computer program to access their contents. You may place a variable inside the stack (i.e. the working memory of your computer program) or within the program’s dynamic memory.

Characters, Booleans, and Integers are known as value types since they keep their data within the program’s stack. On the other hand, strings, arrays, and objects are known as “reference types” because they don’t hold the values inside them. Instead, they serve as markers that point to the part of computer memory where the data is stored.

## How to Name a Variable

You need to name your variables before using them. The variable’s name serves as an identifier: it helps the program in locating and specifying the variable. C# allows you to choose variable names freely. However, just like any other language, C# has some rules regarding variable names. These rules are as follows:

- You may use an underscore (i.e. “\_”), letters, and numbers to create a variable name.

- You can't use a number to start the name. Thus, FirstSample is valid while 1stSample is not. camelCase for local variables, such as cost, orderDetail, dateOfBirth, and firstName
- You can't use C# keywords in naming your variables. This simple rule prevents compilation and runtime errors in your programs. If you really want to use keywords for your variables, you may begin the name using the "@" symbol (e.g. @bool, @int, @char, etc.).
- A meaningful or descriptive name that is neither too long nor too short to identify the information stored in a variable just by looking at it
- Can contain the letters a–z and A–Z, the numbers 0–9, and the underscore (\_) character—other symbols are not allowed
- No spaces and cannot start with a number

### Valid Names

- cost, name, order, order1, \_order1, income
- order\_Detail, orderDetail, dateOfBirth, hourlyRate, firstName, first\_Name, isValid

### Invalid Names

- 1 (*number*)
- class, while, if, protected (*keyword*)
- 1order, 1name (starts with a number)

### How to Declare a Variable

You should declare a variable before using it. When declaring a variable:

1. Specify the variable's data type (e.g. char)
2. Specify the variable's name (e.g. sample)
3. Set the variable's initial value. This step is completely optional.

The format for variable declarations is:

*data\_type name = initial\_value;*

Here are some samples of valid variable declarations:

*int year = 2000;*

*string = yourname;*

*char = test;*

Important Note: You need to end each of your C# statements using a semicolon. If you'll forget to add even a single semicolon character, your programs will generate undesirable and/or unexpected results.

### How to Assign a Value

Just like any other computer language, C# allows you to assign a value to a variable. This process, known as “value assignment”, requires you to use the equals sign. Programmers refer to this symbol as the “assignment operator”. Write the variable's name on the left side of the operator. Then, specify the value you want to assign on the other side. Check the following examples:

*test = x;*

*yourname = “John Doe”;*

### How to Initialize a Variable

In programming, “initialization” is the process of assigning a value to a new variable. Thus, you'll set this “initial value” while declaring a variable.

### The Default Value of a Variable<sup>105</sup>

If you won't assign any value for your variable, C# will perform an automated initialization (i.e. it will assign a default value to the empty variable). The following tables will show you the default value of each data type:

| Data Type | Default Value |
|-----------|---------------|
| sbyte     | 0             |
| byte      | 0             |
| short     | 0             |
| ushort    | 0             |
| int       | 0             |
| uint      | 0u            |
| long      | 0L            |
| ulong     | 0u            |

| Data Type | Default Value |
|-----------|---------------|
| float     | 0.0f          |
| double    | 0.0d          |
| decimal   | 0.0m          |
| bool      | false         |
| char      | '\u0000'      |
| string    | null          |
| object    | null          |

The code given below will show you how to declare variables and assign values.

*// This is an example*

*byte years = 10;*

*short days = 31;*

*decimal pi = 3.14;*

*bool isittasty = true;*

```
char sample = x;  
string animal = "dog";
```

Important Note: Lines that begin with two forward slashes (i.e. //) are “comments”. In programming, comments are descriptive text added to the code to improve its readability. Language compilers ignore these lines. Thus, comments won’t affect the functionality of your computer programs.

### **Creating an identifier**

If you look through some of the suggestions that come with Microsoft, you will find that they will recommend that you use a Camel notation when you are working with the variables, and it will recommend that you work with the Pascal notation when you want to work with the methods. Working with the Camel notation will mean that you will keep the first letter of your name as lower case. If you are creating an identifier that has a compound word then you will make sure that the first letter of your second word will start with an uppercase letter:

|                    |                        |
|--------------------|------------------------|
| <i>payment</i>     | <i>completePayment</i> |
| <i>mathematics</i> | <i>firstClass</i>      |

The Pascal notation will take things a bit differently. This notation style requires you to start the first word using an uppercase letter. The first letter in all of the other words in the sequence should also be written in uppercase as well. Some examples of using the Pascal notation include:

|                    |                   |
|--------------------|-------------------|
| <i>WriteLine()</i> | <i>ReadLine()</i> |
| <i>Start()</i>     | <i>Main()</i>     |

When it comes to naming these identifiers, you can work with numbers and underscores as well. But, if for some reason you can’t begin the name of these identifiers with a number, you can write out something like seven books, but you can’t write out ‘7books.’

Now, these notations are not required, and you can change them around as much as you would like. But this is considered a proper coding practice if

you're working with C# so it is best to follow these rules so the code will work exactly the way that you want it.

As you can see, C# is a pretty easy language to learn. You do not need to worry about a lot of rules that are too complicated to remember like the other programming languages, and yet you will still have a lot of the power and the flexibility to get the most out of coding.

## Definite Assignment

C# enforces a definite assignment policy. This means that you will need to initialize the local variable with the value before using it. Suppose the following is written:

```
Console.WriteLine(name);
```

**Error:** Use of unassigned local variable ‘name’

If you try to use a variable that hasn't been declared, your code won't compile. In this case, the compiler will tell you that something is wrong. Trying to utilize a variable minus having a value assigned to it also causes an error.

In order to initialize the variable with a value, you simply have to use the assignment operator “=”. The variable name is located to the left side of the operator and on the right side is the value in the following manner.

```
name = "Pirzada"; age = 35; weight = 70; isMarried = true;
```

On declaring a variable with a type, it cannot be redeclared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an **int** and then assign it a Boolean value of True/False.

```
age = true;
```

**Error:** Cannot implicitly convert type ‘bool’ to ‘int’

If I try to assign a **string** to an **age** variable, it is declared an **int** datatype.

```
age = "Pirzada";
```

OR

```
age = name;
```

The above statement will give a compile-time error because the **string** value cannot be assigned to an **int** type variable.

You can combine the declaration and initialization statements at the same time on the same line as shown below, which is more convenient.

```
string name = "Pirzada"; int age = 35; int weight = 70; bool isMarried = true;
```

You can also use a mathematical expression.

```
int wowExp = (3 + 2) * 4; Console.WriteLine(wowExp);
```

OUTPUT:

20

The right side is being evaluated and then assigned to the variable on the left. Meaning  $3+2$  is 5 and  $5 * 4$  is 20, which will be assigned to the **wowExp** variable.

You can also assign the same value to multiple different variables all at the same time.

```
int a, b, c; a = b = c = 786; Console.WriteLine(a); Console.WriteLine(b);  
Console.WriteLine(c);
```

OUTPUT:

786

786

786

## Using Type Inference

C# 3.0 introduced the implicitly typed variable with the **var** keyword. Now you can declare a local variable without giving an explicit or real type. The variable still receives a type at compile time, but the type is provided by the compiler.

Actually, the **var** keyword instructs the compiler to infer the type of variable from the expression on the right side of the initialization statement. The compiler is given the task to determine and assign the suitable type.

```
var variableName = variableValue; // Syntax
```

**var** is optional, and it's just for convenience.

Let's use the same variables as above.

```
var name = "Pirzada"; var age = 35; var weight = 70; var isMarried = true;
```

Now the variables are **implicitly typed local** variables, meaning that you don't have to explicitly specify the type. The compiler is tasked to determine and assign the suitable type. A **string** variable is indicated by double quotes, **char** variable is denoted by single quotes, and a **bool** type is denoted by true and false values.

**Suggestion:** Using **var** is convenient, but use it only when the type is obvious from the right side of the assignment.

**Double** is used to denote a literal number with a decimal point unless you add the **M/m** suffix to indicate a **decimal** variable or the **F/f** suffix to indicate **float** variable.

```
var value = 24899.45m; var number = 20.8f;
```

To initialize **number**, you need to add **f** as a suffix after 20.8 to explicitly tell the compiler to change 20.8 to a **float**. Similarly, to initialize **value**, you need to add **m** as a suffix to change 24899.45 into a **decimal** type.

A few rules you need to follow:

- **var** can only be declared and initialized in a single statement. Otherwise, the compiler doesn't have anything from which to infer the type.
- **var** is not supposed to be utilized on fields at class scope.
- The initializer cannot be null and must be an expression.
- Multiple, implicitly typed variables cannot be initialized in the same statement.
- An object cannot be initialized unless a new object is created in the initializer.

## Default Value Expressions

Default value expressions are especially useful in combination with generic types when you don't know in advance what the default value for the given type will be.

default(T) Expression: Old Way

A default value expression **default(T)** returns the default value of a type **T**, where **T** is a reference type or a value type.

```
T variableName = default(T); // syntax
```

### Example:

```
var name = default(string); var age = default(int); var weight = default(int);  
var isMarried = default(bool);
```

default literal: New Way

The **default literal** is a new feature in C# 7.1 that is used to get the default value of the specified data type when the statement is executed. This feature works for value types as well as reference types.

**Note:** **default literal** is equivalent to `default(T)` where `T` is the inferred type.

```
string name = default; int age = default; int weight = default; bool isMarried  
= default; Console.WriteLine(name); Console.WriteLine(age);  
Console.WriteLine(weight); Console.WriteLine(isMarried);
```

In the output below, a null value is printed as a blank line.

OUTPUT for default literal & `default(T)`:

0 0 False

**new** Operator:

Using the **new** operator invokes the default constructor of a value type. This allows you to create a variable using the **new** keyword, which automatically sets the variable to its default value.

**Note:** All values in C# are an instance of a specific type.

```
var age = new int(); var isMarried = new bool(); Console.WriteLine(age);  
Console.WriteLine(isMarried);
```

OUTPUT:

0

False

# Chapter 5: What is Type Conversion?

In general, an operator works on arguments that belong to the same type. However, the C# language offers a wide range of data types that you can use for certain situations. To conduct any process on variables that belong to different types, you should convert one of those variables first. In C#, data type conversion<sup>[106](#)</sup> can be implicit or explicit.

Each C# expression has a data type. This data type results from the literals, variables, values, and structures used inside the expression. Basically, you might use an expression whose type is incompatible for the situation. When this happens, you'll get one of these results:

- The program will give you a compile-time error.
- The program will perform an automated conversion. That means the program will get the right expression type.

Converting an “s” type to “t” type allows you to treat “s” as “t” while running your program. Sometimes, this process requires the programmer to validate the type conversion. Check the examples below:

- Converting “objects” to “strings” will need verification during the program’s runtime. This verification ensures that you really want to use the values as strings.
- Converting “string” values to “object” values doesn’t need any validation. That’s because the “string” type is a branch of the “object” type. You can convert strings to their “parent” class without losing data or getting any error.
- You won’t have to perform verification while converting int values to long values. The “long” data type covers all of the possible values of the “int” type. That means the data can be transformed without any error.

- Converting “double” values to “long” values involves validation. You might experience loss of data, depending on the values you’re working on.

Important Note: C# has certain restrictions when it comes to changing data types. Here are the possible type conversions supported by this language:

### **Explicit Conversion**

Use this conversion if there’s a chance of information loss. For instance, you’ll experience information loss while converting floating-point values to integer values (i.e. the fractional section will disappear). You might also lose some data while converting a wide-range type to a narrow-range type (e.g. long-to-int conversion, double-to-float conversion, etc.). To complete this process, you need to use the “type” operator. Check the examples below:

*class Example*

{

*static void Main()*

{

*double yourDouble = 2.1d;*

*System.Console.WriteLine(yourDouble);*

*long yourLong = (long)yourDouble;*

*System.Console.WriteLine(yourLong);*

*yourDouble = 2e9d;*

*System.Console.WriteLine(yourDouble);*

```
int yourInt = (int)yourDouble;  
  
System.Console.WriteLine(yourInt);  
  
}  
}
```

If compiled and executed correctly, that program will give you these results:



## Implicit Conversion<sup>107</sup>

You can only perform this conversion if data loss is impossible. This conversion is referred to as implicit because it doesn't require any operator. The C# compiler will perform implicit conversion whenever you assign narrow-range values to variables that have a wide-range.

## String Conversion

C# allows you to convert any data type to “string”. The compiler will perform this conversion process automatically if you will use “+” and at

least one non-string argument. Here, the non-string argument will become a string and the “+” operator will produce a new value.

The implicit type conversion is performed to do actions like conversions that are from a base class to a derived class. These are performed in what is referred to as a “type safe” manner. The other type of conversion, explicit, require a cast operator in order to complete their function. There are 16 different functions within C# that are able to be used as a built in type of the conversion methods. Here are some examples you should practice becoming familiar with within the C# environment.

- ToChar- allows you to take a type and convert it to a single Unicode character
- ToByte- allows you to get a byte by converting it from a type
- ToDateTime- allows you to get a date time structure by converting a type (either string or integer)
- ToBoolean- allows you to get a Boolean value from converting a type
- ToDouble- allows you to get a double type from converting a type
- ToDecimal- allows you to get either an integer or decimal type converted from a floating point
- ToInt64- allows a 64 bit integer to be converted from a type
- ToInt32- allows a 32 bit integer to be converted from a type
- ToInt16- allows a 16 bit integer to be converted from a type
- ToSbyte- allows a signed byte type to be converted from a type
- ToString- allows a string to be converted from a type

- ToSingle- allows a small, floating point number to be converted from a type
- ToUInt64- allows an unsigned bit integer to be converted from a type
- ToUInt32- allows an unsigned long type to be converted from a type
- ToUInt16- allows an unsigned int type to be converted from a type
- ToType- allows a specified type to be converted from a type

For your exercise today, I encourage you to go find programming that does each and every one of these functions/conversions in order to get more familiar with what they involve and how they can assist you in your own programming.

Remember that although your coding may be long, your output will be short—but should always produce the answer you were trying to achieve from the coding in question that you have done.

# Chapter 6: The Need for Operators

In this chapter, we will take some time to talk about the operators<sup>[108](#)</sup> that you can work with inside the C# language.

By reading this material, you'll know how these operators work and how you can use them in your source codes. These operators are pretty simple to understand and use but this doesn't mean they're useless, in fact, they can help make sure that your code will perform the tasks that you require it to complete. Without using some of these operators, your program will lack functionality, and won't work all that well. These operators may be simple to understand, but they really add a ton of power to enhance the overall capability of your code.

Operators are important no matter which type of coding language that you are working with. We will go into detail on the different operators separately so that you can understand how each one is supposed to work and how you can work with each one to fulfill a variety of purposes and accomplish different tasks.

## Operators – The Basics

Operators help you in processing objects and data types. They accept inputs and operands to produce a result. The C# language uses special characters (e.g. “.”, “+”, “^”, etc.) as operators. Also, C# codes can take up to three different operands.

## The Different Types of Operators

C# programmers divide operators into the following types:

### Assignment Operator

While there are some other options that you can go with when it comes to working with the assignment operators, this is the most common way that you will use these operators. In addition, there are a few different types of assignment operators that you can work with, but the most common one is the equal sign. Some of the operators that you can work with inside of your C# code includes the following:

- =

The function of this operator is to allow you to perform simple assignment operations. It can assign the value to a variable that you are working on at that time. For example, writing `int sample = 100` will tell the program that you want to assign 100 to the variable that is called ‘sample.’ It won’t perform any additional tasks on this variable or on the value involved.

- +=

This is the additive assignment operator. Its function is to add up the values of your two operands and then assign the sum to your left-hand operand.

- -=

Programmers will often refer to this as the ‘subtractive’ assignment operator. Its function is to subtract the value of the operand on the right side from the one on the left side and then assign the difference to the left-hand operand.

- \*=

The function of this operator is to multiply the values of each operand and then assign the product to the left-hand operand.

- /=

The function of this operator is to divide up the two variables and then take the result and assign it to the variable on the left.

When you are working with the assignment operators, make sure that you have both operands that belong to the same type of data. If you find that the various operands you are using are not compatible, some issues could arise which can result in an error in the program if you try to execute it and will require additional time and effort to fix.

In C#, you need to use the equals sign (i.e. =) to assign values. Here’s the syntax that you need to use when assigning a value:

*name\_of\_operand = expression, literal, or second\_operand*

Here are some examples:

```
int y = 99;
```

```
string hello = "Good morning.;"
```

```
char sample = 'z';
```

Important Note: C# allows you to cascade (i.e. use several times within a single expression) the assignment operator. That means you won't have to enter multiple equals signs if you are creating variables of the same data type. The examples given below will illustrate this rule:

```
int d = 1, e = 2, f = 3;
```

```
string hi = "Good Morning.", bye = "Farewell.", thanks = "Thank you.";
```

### Logical Operators

A logical (also known as “Boolean”) operator accepts and returns Boolean values (i.e. true or false). Here are the four logical operators of C#:

- “&&” – Programmers refer to this operator as “Logical AND”. It will give you “true” if both of the operands are true.
- “^” – This is known as the “Exclusive OR” operator. It will give you true if one of the operands is true.
- “||” – This is the “Logical OR” operator. You will get true if at least one of the operators is true.
- “!” – This operator, known as the “Logical Negation” operator, reverses the value of an operand.

Study the code to understand the use of these operators<sup>[109](#)</sup>:

```
class Test
```

```
{
```

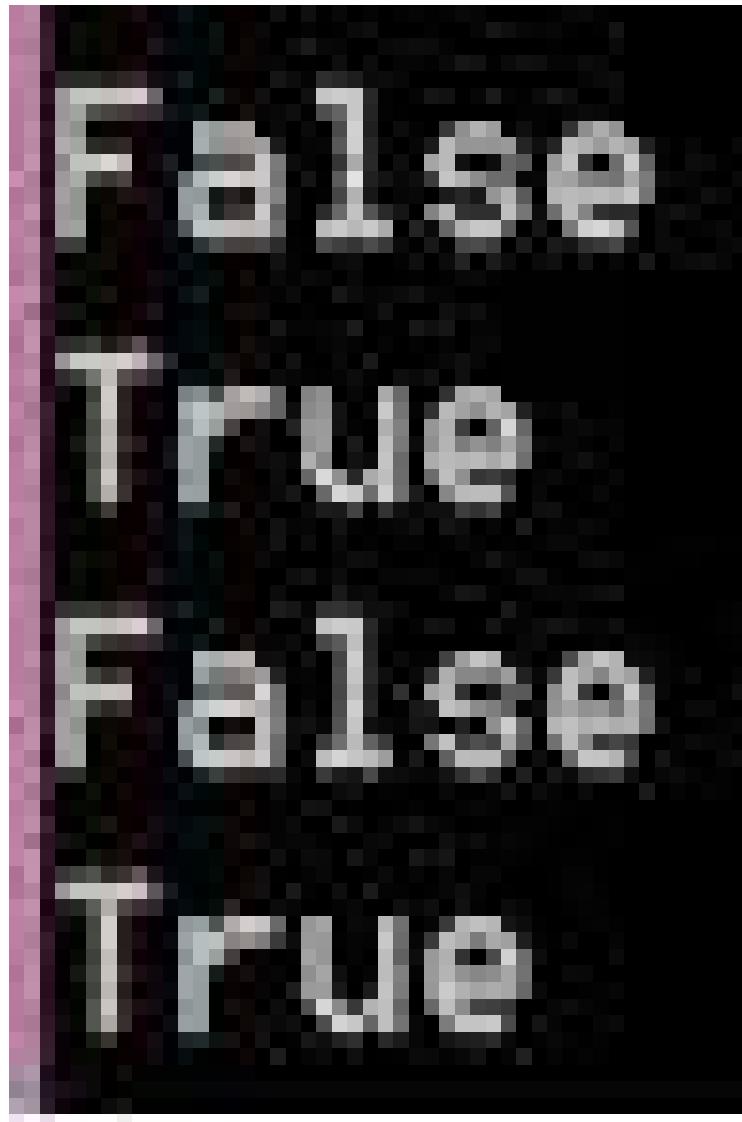
```
static void Main()
{
    bool x = true, y = false;

    System.Console.WriteLine(x && y);
    System.Console.WriteLine(x || y);
    System.Console.WriteLine(!x);
    System.Console.WriteLine(x ^ y);

    // This is an example.

}
```

Once you have compiled and run this code, you will see the following results:



### Arithmetic Operators

These operators help you in performing math operations. This category consists of the following operators:

- “+” – Use this operator to perform addition using two operands together or values (e.g. `int x = 1 + 1`)
- “-” – This operator allows you to perform subtraction in your codes. It deducts the value of the right operand from that of the left operand (e.g. `int x = 2 - 1`)

- “\*” – With this operator, you can multiply the values of two operands (e.g. `int r = 6 * 6`)
- “/” – Use this operator to divide number values in your C# codes. Basically, “/” divides the value of the left-hand operand by that of the right-hand operand.
- “++” – This is known as the “increment operator”. It increases the value of an operand by one. Unlike the ones discussed above, this operator works on a single operand. Also, you may write it before or after the operand. (e.g. `99++`, `1++`, `++5`, etc.)
- “--” – This operator is the exact opposite of the increment operator. You can use it to decrease an operand’s value by one. You may write it before or after the operand you want to modify.
- “%” - This is often called the remainder or is commonly known as the ‘modulo’ operator. Its function is to divide the left-hand operand by the right operand and then returns the remainder.

The code given below will show you how these operators work:

```
class Test
```

```
{
```

```
static void Main()
```

```
{
```

```
    double d = 100, e = 3;
```

```
    System.Console.WriteLine(d + e);
```

```
    System.Console.WriteLine(d - e);
```

```
System.Console.WriteLine(d / e);  
System.Console.WriteLine(d * e);  
System.Console.WriteLine(++d);  
System.Console.WriteLine(--e);  
  
// This is an example.  
}  
}
```

Once you compile and execute that code, you'll see this on your command prompt:



### Binary Operators

These operators work on binary numbers. In the IT world, all of the information is expressed as a sequence of zeros and ones. That means you

need to master binary operators if you want to be a successful programmer.

A binary operator works exactly like a logical one. Actually, you may think that binary and logical operators use different inputs but conduct the same processes. A logical operator uses Boolean values to produce results. A binary operator, on the other hand, works on numbers presented as binary digits. Here are the binary operators you'll encounter while using C#:

- “&” – This is called “Binary AND”. It indicates the position/s where both of the operands have “1”.
- “^” – Programmers refer to this operator as “Binary Exclusive OR”. It will place “1” in each position where the values are different.
- “|” – This is the “Binary OR” operator. It indicates the position/s where any of the operands has “1”.
- “~” – This acts as the negation operator for binary values. Just like “!”, it reverses the current value of a variable.
- “<<” – This is the “Binary Left Shift” operator. It moves the bits of a binary number to the left. This movement depends on the value that you'll assign (e.g.  $4 << 1$ ,  $4 << 3$ ,  $4 << 2$ , etc.).

### *Relational/Comparison Operators*

The next type of operator that you can use is known as the relational operator. These operators are very useful because they allow you to compare the values of two of your operands. Each of these operators gives a Boolean value as the final output. Because of this function, these operators are the best to use when you want to create some conditional statements. We will have a look at some of the relational operators that you can work with to make your C# code, but for these examples assume that ‘ $e = 150$ ’ and that ‘ $d = 100$ .’

- “==” – This is the Equality operator. It checks whether the two operands are equal (e.g.  $x == y$ ). The function of this the operator

is to allow you to check the equality of two values. If the two values end up being equal, the operand will tell you it is true. Otherwise, the operand will tell you it is false. For example, saying the  $d == e$  would show up as false.

- “ $>$ ” – The function of this operator is to check whether the operand on the left is greater than the operand on the right. If it is, then the operator will tell you it is true. For example, saying that  $e > d$  would be true.
- “ $<$ ” – This is actually less than an operator, it allows you to check whether the operand on the left side is less than the operand on the right side. It will give you “true” if the left-hand operand’s value is less than that of the right-hand operand (e.g.  $y < x$ ).
- “ $\geq$ ” – With this operator, you can determine whether the value of the left-hand operand is greater than or equal to that of the right-hand operand. The function of this operand is that it will say the value of the operand on the left side is truly greater than or equal to, the operand on the right side. Otherwise, it will tell you the statement is false. For example, saying that  $e \geq d$  evaluates as true.
- “ $\leq$ ” – This operator will give you true if the value of the left-hand operand is less than or equal to that of the right-hand operand.
- “ $\neq$ ” – The function of this operator is to allow you to test the inequality of two values/operands. If the values end up not being equal, it will tell you this is true. For example,  $e \neq d$  would result in the operator saying it’s true.

When you are working with these relational operators, always remember that you will get a Boolean result each time. What this means is that the answer you get will be either true or false. You also have to check if you are using two equal signs when you are working with the equality operator. If you end up getting these operators mixed up with your assignment operator,

you will probably get an error message, and that program will not work out the way that you want.

### Conditional Operator

C# users refer to “?” as the conditional operator. Basically, this operator uses the result of a Boolean expression to determine which statement should be processed. This is called “ternary operator” because it works on three operands. Here, the first value should be Boolean, while the remaining values need to belong to the same data type (e.g. characters, strings, numbers, etc.).

The syntax of this operator is:

*first\_operand ? second\_operand : third\_operand;*

Here's how it works: If “first\_operand” is true, the program will work on “second\_operand”. If “first\_operand” is false, however, the program will work on “third\_operand”.

# Chapter 7: The Conditional Statements

C# has conditional statements, mostly used for controlling the flow of execution. The conditional statements expect the programmer to specify a condition or a set of conditions and the corresponding set of statements to be executed if a condition is found to be true. The programmer can also specify the set of statements that are to be executed if the condition is not true.

Let us discuss the various conditional statements supported in C#.

## **if Statement**

This statement is used to evaluate a Boolean expression before a set of statements can be executed. If the condition stands true, then there will be execution of one set of statements, otherwise, another set of statements will be executed. Here is the syntax for this statement<sup>[110](#)</sup>:

```
if(boolean_expression) {
    /* statement(s) to execute if above boolean expression is true */
}
```

Here is an example:

```
using System;
namespace DecisionMaking {
    class IfStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 5;
            /* check a boolean condition via if statement */
            if (x < 10) {
                /* to be printed if the condition is true */
                Console.WriteLine("x is less than 10");
            }
            Console.WriteLine("The value of x is : {0}", x);
            Console.ReadLine();
        }
    }
}
```

The code returns the following output:

x is less than 10

The value of x is : 5

The condition was found to be true, that is, the value of variable x is less than 10, hence, and the statement just below the condition was executed. What if the condition was false?

```
using System;
namespace DecisionMaking {
    class IfStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 15;
            /* check a boolean condition via if statement */
            if (x < 10) {
                /* to be printed if the condition is true */
                Console.WriteLine("x is less than 10");
            }
            Console.WriteLine("The value of x is : {0}", x);
            Console.ReadLine();
        }
    }
}
```

The code returns the following:

# The value of x is : 15

The condition was found to be false, hence the statement outside its block was executed.

### **if-else Statement**

This is simply a combination of an *if* statement with an *else* part. The *if* part is executed if the condition is true, while the *else* part is executed when the condition is false. The two parts cannot all be executed at once. Here is the syntax for the statement:

```
if(boolean_expression)
{
    // code executes if the condition is true.
}
else
{
    // code executed if the condition is false
}
```

Consider the example given below:

```
using System;

namespace DecisionMaking {
    class IfElseStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 5;

            /* a check of the boolean condition */
            if (x < 10) {
                /* if the condition is true, the following will be printed */
            }
        }
    }
}
```

```

        Console.WriteLine("x is less than 10");
    } else {
        /* if the condition is false, the following will be printed */
        Console.WriteLine("x greater than 10");
    }
    Console.WriteLine("The value of x is : {0}", x);
    Console.ReadLine();
}
}
}

```

The code returns the following:

```
x is less than 10
The value of x is : 5
```

The *if* condition evaluated to a true, hence the statement within its block has been executed. What if the condition was false?

```

using System;
namespace DecisionMaking {
    class IfElseStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 15;
            /* a check of the boolean condition */
            if (x < 10) {
                /* if the condition is true, the following will be printed */
                Console.WriteLine("x is less than 10");
            } else {
                /* if the condition is false, the following will be printed */
                Console.WriteLine("x greater than 10");
            }
        }
    }
}

```

```
        }
        Console.WriteLine("The value of x is : {0}", x);
        Console.ReadLine();
    }
}
```

The code gives the following output:

```
x greater than 10
The value of x is : 15
```

The *if* condition evaluated to a false, hence the statement within the *else* block has been executed. You must have noticed that part outside the two blocks has been executed in both cases. That is what happens.

### **else...if Statement**

In some cases, we may be in need of checking a multiple number of conditions. In such a case, we can use the *else if* statement. It takes the syntax given below:

```
if(boolean_expression a) {
    /* Runs if the boolean expression a is true */
}
else if( boolean_expression b) {
    /* Runs if the boolean expression b is true */
}
else if( boolean_expression c) {
    /* Runs if the boolean expression c is true */
} else {
    /* to run if none of above conditions is true */
}
```

Here is an example:

**using System;**

```
namespace DecisionMaking {
    class ElseIfStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 5;

            /* checking the boolean condition */
            if (x == 1) {
                /* print this statement if the condition is true */
                Console.WriteLine("Value of x is 1");
            }
            else if (x == 2) {
                /* print this statement if the if else if condition is true*/
                Console.WriteLine("Value of x is 2");
            }
            else if (x == 3) {
                /* print this statement if the if else if condition is true */
                Console.WriteLine("Value of x is 3");
            } else {
                /* print this statement if none of the above conditions is true */
                Console.WriteLine("All conditions are false");
            }
            Console.WriteLine("Exact value of x is: {0}", x);
            Console.ReadLine();
        }
    }
}
```

The code prints the following result:

# All conditions are false

## Exact value of x is: 5

All the conditions were found to be false; hence the block of code outside the conditions has been executed. Let us set the value of variable x to 2 and see what happens with the code:

```
using System;
namespace DecisionMaking {
    class ElseIfStatement {
        static void Main(string[] args) {
            /* defining a local variable */
            int x = 2;
            /* checking the boolean condition */
            if (x == 1) {
                /* print this statement if the condition is true */
                Console.WriteLine("Value of x is 1");
            }
            else if (x == 2) {
                /* print this statement if the if else if condition is true*/
                Console.WriteLine("Value of x is 2");
            }
            else if (x == 3) {
                /* print this statement if the if else if condition is true */
                Console.WriteLine("Value of x is 3");
            } else {
                /* print this statement if none of the above conditions is true */
                Console.WriteLine("All conditions are false");
            }
            Console.WriteLine("Exact value of x is: {0}", x);
            Console.ReadLine();
        }
    }
}
```

```
    }  
}  
}
```

The output is shown below:

```
Value of x is 2  
Exact value of x is: 2
```

An *else if* condition evaluated to a true hence the statement within its block was executed. The block of code outside all the conditions was also executed. You can play around with the code by modifying the value of x to various values and see what will happen.

### Nested if Statements

C# allows us to nest conditional statements. We can nest both the *if* and the *else if* statements, which means that we use them inside another *if* or *else if* statement. The following syntax demonstrates how we nest the *if* statement in C#:

```
if( boolean_expression a) {  
    /* to execute if the boolean expression a is true */  
    if(boolean_expression b) {  
        /* to execute if the boolean expression b is true */  
    }  
}
```

The *else if* statement can be nested using the syntax given above.

Consider the example given below:

```
using System;
```

```
namespace DecisionMaking {
    class NestedIf {
        static void Main(string[] args) {
            /* defining local variables */
            int x = 1;
            int y = 2;

            /* checking the boolean condition */
            if (x == 1) {
                /* if the condition is true, check the if condition below */
                if (y == 2) {
                    /* if the condition is true, print the following */
                    Console.WriteLine("Value of x is 1 and b is 2");
                }
            }
            Console.WriteLine("Exact value of x is : {0}", x);
            Console.WriteLine("Exact value of y is : {0}", y);
            Console.ReadLine();
        }
    }
}
```

The code returns:

```
Value of x is 1 and b is 2
Exact value of x is : 1
Exact value of y is : 2
```

Both *if* conditions evaluated to a true, hence the statement within the nested *if* were executed. If either or both conditions evaluated to a false, then this

statement could not have been executed. Here is an example:

```
using System;
namespace DecisionMaking {
    class NestedIf {
        static void Main(string[] args) {
            /* defining local variables */
            int x = 1;
            int y = 2;
            /* checking the boolean condition */
            if (x == 1) {
                /* if the condition is true, check the if condition below */
                if (y == 3) {
                    /* if the condition is true, print the following */
                    Console.WriteLine("Value of x is 1 and b is 2");
                }
            }
            Console.WriteLine("Exact value of x is : {0}", x);
            Console.WriteLine("Exact value of y is : {0}", y);
            Console.ReadLine();
        }
    }
}
```

In the above code, the nested *if* condition checks whether the value of variable y is 3, which is false. The statement within this block will not be executed. The C# compiler will skip this section and proceed to execute the code outside the conditions. It returns the following:

```
Exact value of x is : 1
```

```
Exact value of y is : 2
```

## **switch Statement<sup>11</sup>**

This statement is the same as using multiple *if* statements. It is created with a list of possibilities, an action for every possibility and a default section to be execution in case any of the options doesn't evaluate to a true. Here is syntax for this statement:

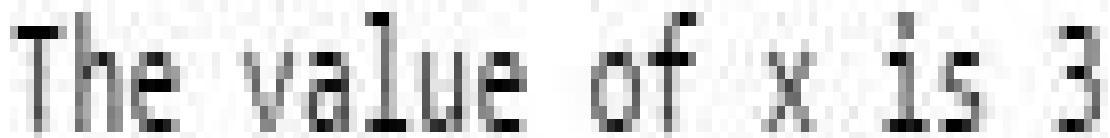
```
switch(expression)
{
    case <value_1>
        // code
        break;
    case <value_2>
        // code
        break;
    case <value_N>
        // code
        break;
    default
        // code
        break;
}
```

Here is an example:

```
using System;
public class SwitchStatement
{
    public static void Main()
```

```
{  
int a = 3;  
switch (a)  
{  
    case 1:  
        Console.WriteLine("The value of x is 1");  
        break;  
    case 2:  
        Console.WriteLine("The value of x is 2");  
        break;  
    case 3:  
        Console.WriteLine("The value of x is 3");  
        break;  
    default:  
        Console.WriteLine("Unknown value of x");  
        break;  
}  
}  
}
```

The code returns the following:



The value of x is 3

The value of variable was initialized to 3. After executing the code, the *case* code for 3 will be matched, hence the statement within its block will be executed. If none of the *case* conditions is true, then the *default* section will be executed. This is demonstrated below:

```
using System;  
public class SwitchStatement  
{  
    public static void Main()
```

```
{  
int a = 5;  
  
switch (a)  
{  
    case 1:  
        Console.WriteLine("The value of x is 1");  
        break;  
    case 2:  
        Console.WriteLine("The value of x is 2");  
        break;  
    case 3:  
        Console.WriteLine("The value of x is 3");  
        break;  
    default:  
        Console.WriteLine("Unknown value of x");  
        break;  
}  
}  
}
```

The code returns the output given below:



Unknown value of x

The case label within the switch statement has to be unique. The switch statement is usable with expressions of any type including strings, integers, bool, char, enum etc. Here is how to use it with a string:

```
using System;  
public class SwitchStatement  
{  
    public static void Main()
```

```
{  
string firstName = "Nicholas";  
switch (firstName)  
{  
    case "Samuel":  
        Console.WriteLine("Your first name is Samuel");  
        break;  
    case "Nicholas":  
        Console.WriteLine("Your first name is Nicholas");  
        break;  
    case "Bismack":  
        Console.WriteLine("Your first name is Bismack");  
        break;  
}  
}  
}
```

The code returns the following output:

Your first name is Nicholas

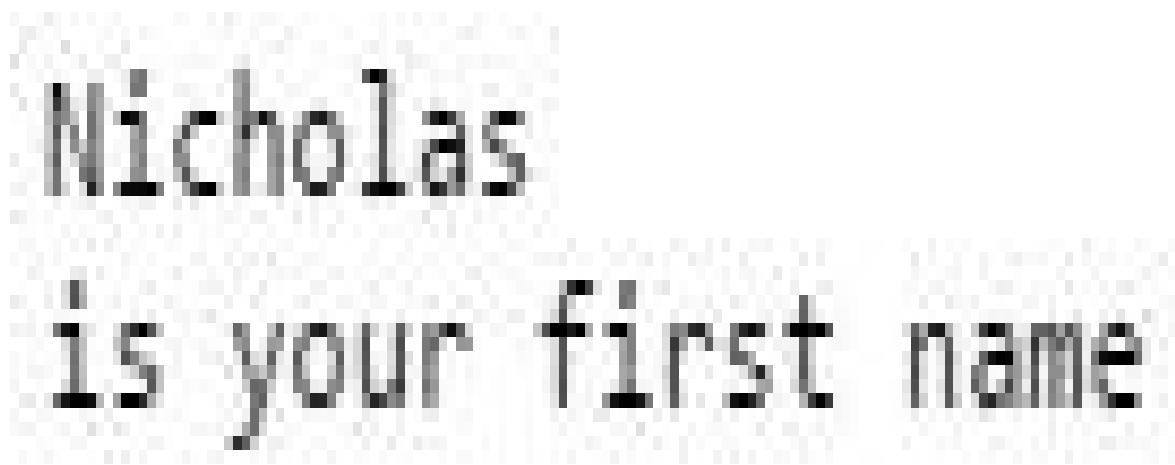
### Goto in switch<sup>112</sup>

In some cases when using the *switch* statement, we may need to skip to a certain case. This can be done using either a *jump* or *goto* statement. Consider the example given below:

```
using System;  
public class SwitchStatement  
{  
    public static void Main()  
    {  
        string firstName = "Nicholas";  
        switch (firstName)  
        {
```

```
case "text":  
    Console.WriteLine("is your first name");  
    break;  
case "Samuel":  
    Console.WriteLine("Samuel");  
    break;  
case "Nicholas":  
    Console.WriteLine("Nicholas");  
    goto case "text";  
    break;  
case "Bismack":  
    Console.WriteLine("Bismack");  
    break;  
}  
}  
}
```

The code returns the following output:



```
Nicholas  
is your first name
```

## Nested Switch

A switch statement can be created within another switch statement. This gives us a nested switch. Here is an example:

```
using System;  
public class NestedSwitch  
{
```

```
public static void Main()
{
    int x = 10;
    switch (x)
    {
        case 10:
            Console.WriteLine(10);
            switch (x - 1)
            {
                case 9:
                    Console.WriteLine(9);
                    switch (x - 2)
                    {
                        case 8:
                            Console.WriteLine(8);
                            break;
                        }
                        break;
                    }
                    break;
                case 20:
                    Console.WriteLine(20);
                    break;
                case 125:
                    Console.WriteLine(25);
                    break;
                default:
                    Console.WriteLine(30);
                    break;
                }
            }
        }
}
```

The output is as follows:

# Chapter 8: Loops

As a programmer, you'll use a loop statement to run certain code blocks repeatedly. Loops play an important part in C# programming. Hence, you should study this material carefully as well if you want to master this language in a short while.

## Loops – The Basics

While creating programs, you often need to execute code sequences multiple times. Typing the same blocks of code several times can be extremely boring. Thus, you need to find a quick and simple way to repeat codes. Fortunately, the C# language supports loop statements.

A loop is a tool that runs code fragments repeatedly. You can use it to run codes for a certain number of times or as long as a given condition is fulfilled. C# offers different types of loop statements. Let's analyze each type in detail:

### The While Loop<sup>[113](#)</sup>

This is the simplest loop statement in C#. Its syntax is:

*while (the\_assigned\_condition)*

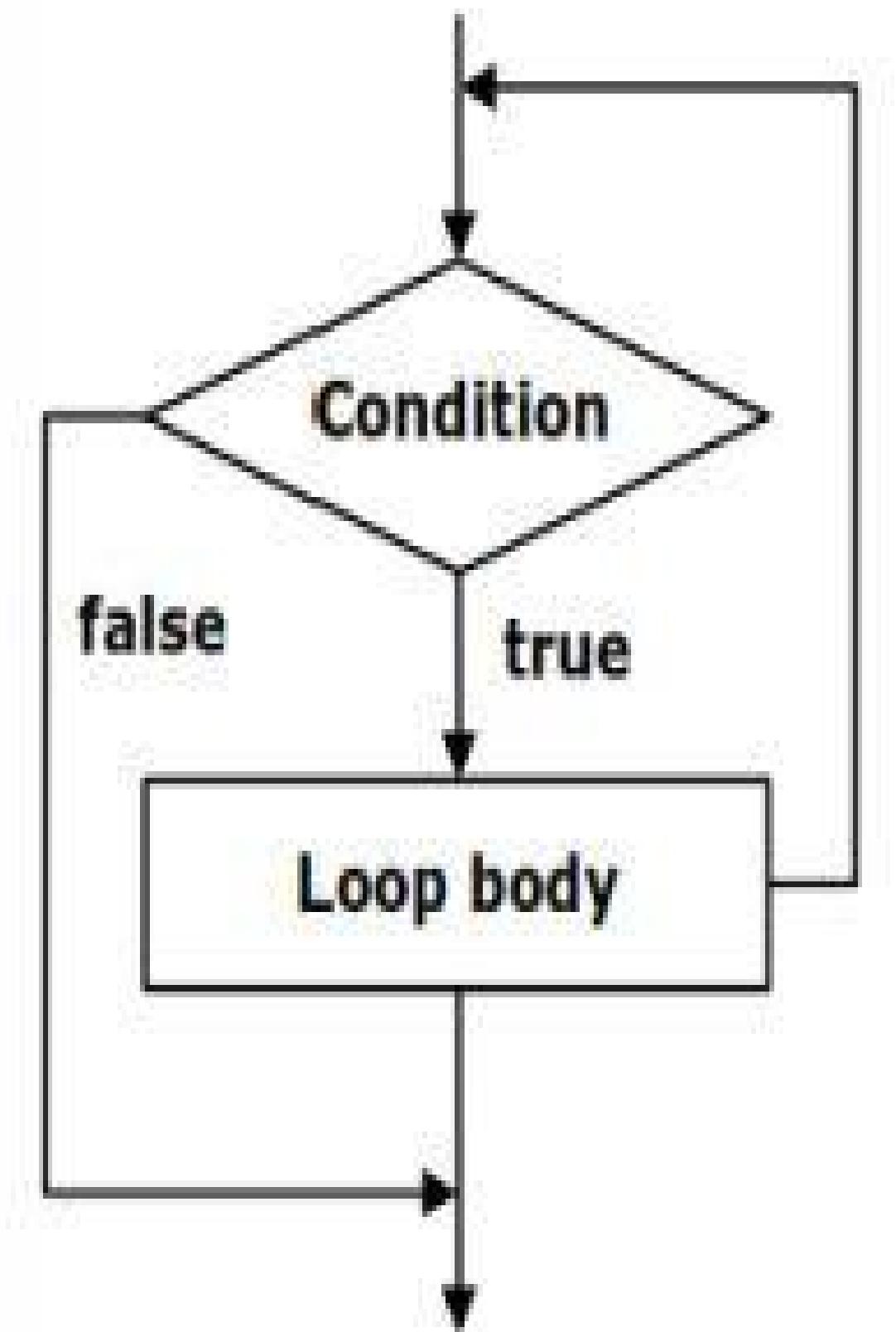
{

*the\_loop's\_body;*

}

In this syntax, “the\_assigned\_condition” is an expression that produces a Boolean value. This condition dictates how many times the “body” will run. The “body”, on the other hand, is the statement (or group of statements) that you want to execute.

The diagram given below will illustrate how while loops work:



When working on a “while” loop, a C# program checks the value of the Boolean expression. If the value is “true”, the program will run all of the statements within the loop’s body. Then, the program will check the Boolean expression to see its value. If the value is still true, the program will rerun the loop’s body and go back to the first step (i.e. check the expression’s value). This process will go on until the Boolean expression evaluates to false. Once this happens, the program will process the code blocks right after the loop.

Important Note: Your C# program won’t run the while loop’s body if the Boolean expression is false. Thus, if the Boolean value is false when you launched the program, your in-loop statements will never be executed.

The following example will illustrate how while loops work:

```
int timer = 10;
```

```
while (timer >= 0)
```

```
{
```

```
    System.Console.WriteLine("Time remaining: " + timer);
```

```
    timer--;
```

```
}
```

If you will compile and execute this code, your command prompt will print this data:

```
Time Remaining: 10
Time Remaining: 9
Time Remaining: 8
Time Remaining: 7
Time Remaining: 6
Time Remaining: 5
Time Remaining: 4
Time Remaining: 3
Time Remaining: 2
Time Remaining: 1
Time Remaining: 0
```

### **The Break Operator**

You can use “break” (a C# operator) to exit a loop prematurely. Programmers use this operator if they don’t want to wait for the loop’s natural termination. Basically, a loop will end once it encounters a break

operator. This situation forces the program to jump to the code fragments right after the loop.

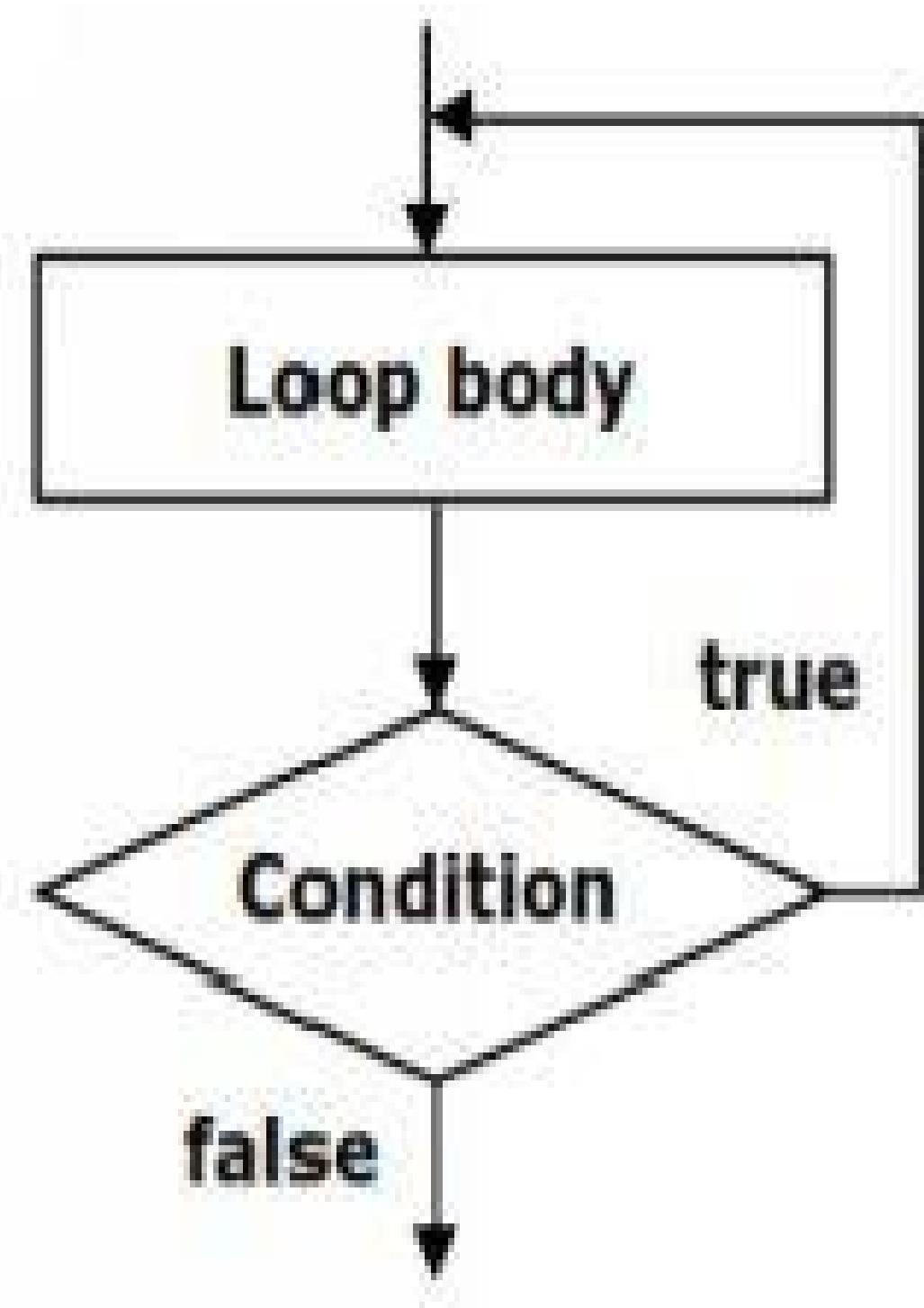
In C#, you can only use this operator inside the loop's body. That means "break" will only work while the loop is running.

### The Do-While Loop<sup>[114](#)</sup>

A do-while loop is almost identical to a while loop. The only difference between these loops is that the former analyzes the Boolean value after running the loop's body. Thus, you can rest assured that your codes will run at least once even if the Boolean expression is false. Here is the syntax of the do-while loop:

```
do  
{  
    statements;  
} while (Boolean expression)
```

Do-while loops work according to this pattern:



The program will execute all of the statements within the loop's body. Then, it will check the condition (i.e. the Boolean expression). If the condition evaluates to true, the program will execute the loop's body again and perform the “value inspection”. This process will go on until the condition

becomes false. Thus, your codes might run forever if your assigned condition always evaluates to true.

## The For Loop<sup>[115](#)</sup>

In general, “for” loops are more complex than do-while and while loops. However, for loops can help you perform difficult tasks using fewer C# codes. The syntax of a “for” loop is:

*for (initializer; Boolean\_expression; update)*

```
{  
    the_loop's_body;  
}
```

A “for” loop has an initializer (i.e. the initial value of the counter), a Boolean expression, a C# statement that updates the counter, and the loop’s body.

The “counter” (i.e. the initial value assigned to the loop) serves as a “for” loop’s most distinctive feature. In most cases, the counter’s value increases until it reaches the final value (e.g. 1 to 10). Also, programmers usually know how many times the “for” loop will iterate.

You may include one or more variables in your “for” loops. These variables may move in descending or ascending order. When writing programs, you may combine ascending and descending variables in a single “for” loop. In addition, an ascending variable can go from 1 to 1024 because “for” loops support arithmetic operations (e.g. addition, multiplication, etc.).

Important Note: All of the parts of a “for” loop are optional. You can create an infinite “for” loop by leaving blank spaces on the syntax. Here’s an example:

```
for ( ; ; )  
{  
    // Loop body  
}
```

At this point, you’re ready to learn more about the different sections of the “for” loop.

### The Initialization

A “for” loop can possess an initializing fragment:

```
for (double sample = 1; ...; ...)
```

```
{
```

```
    // You can use the “sample” variable here.
```

```
}
```

```
    // You can’t use the variable here.
```

C# programs execute this fragment once, right before running the “for” loop. Often, programmers use an initializing fragment to create a counter (also known as “loop variable”) and assign its initial value. This counter is available and usable when inside the loop’s body. C# allows you to declare multiple variables using a single initializing fragment.

### The Condition

Obviously, a “for” loop needs a conditional expression. Here’s a sample:

```
for (double sample = 1; sample < 5; ...)
```

```
{
```

```
// This is the loop's body.
```

```
}
```

Computer programs evaluate the conditional expression before running the loop’s body. If the result is “true”, the body of the loop will run; otherwise, the involved program will jump to the statements written after the current loop.

### The Update Statement

This is the final part of a “for” loop. Basically, an update statement updates the loop’s counter. Let’s use the code snippet given above:

```
for (double sample = 10; sample > 1; sample--)
```

```
{
```

```
// This is the loop's body.
```

}

The program executes this part after running the loop's body. Keep in mind that this statement updates the counter's value.

### *The Loop's Body*

This part consists of C# statements. It can utilize the variables you declared in the loop's initializing fragment. Check the example below:

```
for (double sample = 10; sample > 1; sample--)
```

```
{
```

```
    System.Console.WriteLine(sample);
```

```
}
```

### *The Continue Operator*

In some cases, you need to stop the active iteration without ending the loop itself. You can accomplish this task using the C# operator called “continue”. The example given below will illustrate how this operator works:

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
    if (i % 7 == 0)
    {
        continue;
    }
    sum += i;
}
Console.WriteLine("sum = " + sum);
```

This code computes the total of all the odd numbers within the range (1 to n), which produce remainders when divided by 7.

### The “Foreach” Loop<sup>116</sup>

Just recently, C# introduced the concept of “foreach” loops (i.e. extended “for” loops). This loop concept is also available in other programming languages such as C, VB, C++, PHP, etc. With this programming tool, you

can run all of the elements of a list, array, or other groups of values. A “foreach” loop takes all of the existing elements even in non-indexed data groups.

When writing this kind of loop, use the following syntax:

```
foreach (type name_of_variable in name_of_group)  
{  
    the_statements;  
}
```

As you can see, this loop is much simpler than a typical “for” loop. A “foreach” loop can help you scan all of the objects inside a collection quickly. It’s no surprise that countless programmers use this loop in writing their codes.

### The Nested For Loop

C# allows you to place a “for” loop inside another “for” loop. The loop at the innermost part of the code runs the most number of times. The one at the outermost section, on the other hand, gets the least number of repetitions. You should use the following syntax when “nesting” for loops:

```
for (initializing_fragment, condition, update_statement)  
{  
    for (initializing_fragment, condition, update_statement)  
    {  
        statements  
    }  
}
```

}

Here, the program runs the initial “for” loop, executes its body, and triggers the nested loop. The program will check the second loop’s condition and execute the codes inside it until the evaluation becomes false. Then, the program will make the necessary adjustments on the loops’ counters. The program will repeat the entire process until all of the conditions evaluate to false.

# Chapter 9: C# Methods

This chapter will focus on the C# methods. Here, you'll learn how to declare and utilize methods in your own programs. You need to memorize the lessons contained in this chapter to be an effective C# programmer.

## Methods – The Basics

For most programmers, methods are core aspects of any program. Methods can solve problems, accept user inputs (known as parameters), and produce results.

A method is simply a group of statements that have been put together to perform a common task. Every C# class has at least one method, which is the Main method. For you to be able to use a method, you must define it, and then call it to perform the action it was intended to perform. Methods complete their tasks by representing the data conversions done by the program. Additionally, methods are the areas where the actual processes are completed. This is the main reason why C# programmers consider methods as the basic units of any computer program.

## The Benefits Offered by Methods

In this section, you'll discover the major benefits offered by C# methods. After reading this material, you'll know why you should use these tools in writing your programs.

### Better Structure and Code Readability

Programming experts claim that you should use methods while writing computer programs. Methods, even the simplest ones, can improve the structure and readability of your C# codes.

Here's an important fact that you need to know: programmers spend 20% of their time on writing and checking their computer program. The remaining 80% is spent on maintaining and improving the software. Obviously, you'll have an easy time working on your program if your codes are readable and well-structured.

## Prevention of Redundant Codes

Methods can help you avoid redundant codes in your programs. Redundant or duplicated codes often produce undesirable results in computer applications.

## Better Code Repetition

If your program needs to use a certain code fragment multiple times, it's an excellent idea to transfer the said fragment into a C# method. You can invoke methods multiple times, which means you can repeat important codes without retyping them.

### **Declaring, Implementing, and Invoking C# Methods<sup>117</sup>**

*Method Definition* - A method definition is simply a declaration of the elements that form the structure of the method. Here is the syntax for method definition in C#:

```
<Access_Specifier><The_Return_Type><Your_Method_Name>
(Parameter_List) {
    Method Body
}
```

At this point, you need to know the processes that you can perform on an existing method<sup>118</sup>. These processes are:

- Declaration – In this process, you will link a method to a program. This process allows you to call the method in any part of your program.
- Implementation – This process involves entering codes to complete a certain task. The codes involved here exist inside the method you're using.
- Invocation – This is the process of calling a declared method. Here, you'll use the method to solve a problem or perform an action.

## **Method Declarations**

In C#, you need to declare methods inside a class. Additionally, you're not allowed to “nest” methods (i.e. write a method inside the body of another method). The perfect example for this is Main(), the method you've used multiple times. The code given below will illustrate this:

```
class DeclaringMethods  
{  
    static void Main()  
    {  
        System.Console.WriteLine("Hi C#!");  
  
    }  
}
```

### The Syntax

Use the following syntax when declaring a method:

*static data\_type\_of\_the\_result name\_of\_method (list\_of\_parameters)*

Let's analyze this syntax using the Main() method (i.e. *static void Main()*). Main() uses “void” as its return type since it doesn't generate any result. The word “Main” serves as its name. The parentheses, on the other hand, act as containers for the parameters that the programmer will provide.

Important Note: You have to follow this syntax when writing a C# program. Don't change the placement of the method's parts.

C# doesn't require you to include parameters in your declarations. That means you can leave the parentheses empty (e.g. Main()).

### The Method's Name

You need to indicate the method's name during declarations, invocations, or implementations. Here are the rules that you need to remember when naming your methods:

- Make sure that the initial letter is in uppercase.
- Begin each word with an uppercase letter (e.g. SampleMethod, NewMethod, MainLine, etc.).
- Use verbs and nouns as names of your methods.

Important Note: The rules discussed above are completely optional. Follow these rules if you want to have excellent structure and readability in your C# codes.

### **Method Implementations**

Declaring a method isn't enough. You also need to implement your methods if you want them to run inside your programs. Programmers use the term "body" when referring to the implementation of a method.

#### *The Body*

You'll find the method's body inside a pair of curly braces. This body consists of commands, expressions, and statements that you want to execute. Thus, the body is an important part of any method.

Important Note: Keep in mind that you cannot nest methods in C#. Don't write methods inside other methods.

### **Method Invocations**

Basically, this is the process of running the statements within the method's body. Invoking a method is easy and simple: you just have to indicate its name, add a pair of curly braces, and terminate the line using a semicolon. Here's the syntax that you should use:

*name\_of\_method();*

The sample code given below will show you how to invoke a method:

```

class Animals

{    static void Main()

{

    Console.WriteLine("I love dogs.");

}

}

```

If you'll compile and execute that code, your command prompt will print the following message:

*I love dogs.*

### The Places Where You Can Invoke a Method

C# allows you to invoke methods in the following areas:

- Inside the program's main method (i.e. Main()).
- Inside another method.
  - Inside the method's own body. This technique is called "recursion".

### **Recursive Method Call<sup>119</sup>**

It is possible for a method to call itself. This process is referred to as *recursion*<sup>120</sup>. Consider the factorial example given below:

```

using System;
namespace MethodApplication {
  class NumberChecker {
    public int factorial(int x) {
      /* declaring a local variable */
      int answer;
      if (x == 1) {

```

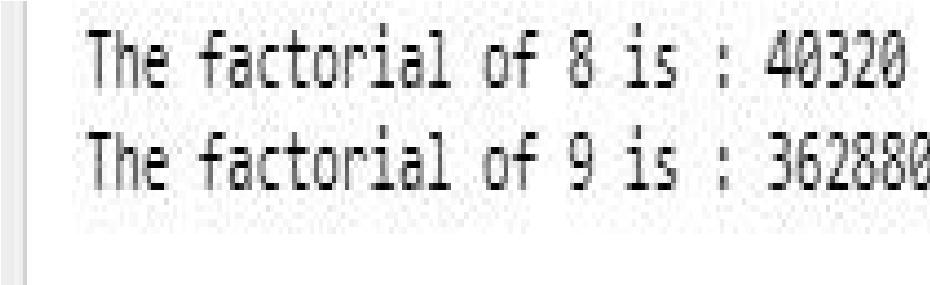
```

        return 1;
    } else {
        answer = factorial(x - 1) * x;
        return answer;
    }
}

static void Main(string[] args) {
    NumberChecker n = new NumberChecker();
    //let's now call the factorial method {0}", n.factorial(8));
    Console.WriteLine("The factorial of 8 is : {0}", n.factorial(8));
    Console.WriteLine("The factorial of 9 is : {0}", n.factorial(9));
    Console.ReadLine();
}
}
}
}

```

We have created the factorial function and instance of the class named *n*. The code will return the result given below:



```

The factorial of 8 is : 40320
The factorial of 9 is : 362880

```

The code was able to give us the factorial of 8 and 9. Note that the factorial of a number is the multiplication of all the numbers below it except 0. For example, the factorial of 8 is  $1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$ , which gives 40320 as shown in the above result.

## Passing Parameters to Methods<sup>[121](#)</sup>

If a method was defined with parameters, then parameters should be passed to it during the call. Three methods can be used for passing parameters to methods. Let us discuss these methods.

### Pass By Value

This method involves copying the actual value of an argument to the formal function parameter. This is the default mechanism of passing parameters to a

method. With this mechanism, when calling a function, a new location is created in the memory for every value parameter. The values for the actual parameters are then copied into them. This means that the changes that are made to the parameter inside the method will have no effect on the argument.

Let us demonstrate this by an example:

```
using System;
namespace MethodApplication {
    class NumberChecker {
        public void swap(int a, int b) {
            int temp;
            temp = a; /* saving the value of a */
            a = b; /* putting b into a */
            b = temp; /* putting temp into b */
        }
        static void Main(string[] args) {
            NumberChecker n = new NumberChecker();

            /* defining a local variable */
            int x = 10;
            int y = 20;

            Console.WriteLine("Before swapping, value of x is : {0}", x);
            Console.WriteLine("Before swapping, value of y is : {0}", y);

            /* let's swap the values by calling the function */
            n.swap(x, y);

            Console.WriteLine("After swapping, value of x is : {0}", x);
            Console.WriteLine("After swapping, value of y is : {0}", y);

            Console.ReadLine();
        }
    }
}
```

The code gives the following output:

```
Before swapping, value of x is : 10
Before swapping, value of y is : 20
After swapping, value of x is : 10
After swapping, value of y is : 20
```

What we did is that we changed the values within the function. However, the above output shows that this change did not take effect; hence it has not been reflected above.

That is how we pass parameters by value in C#.

### **Pass By Reference**

A reference parameter references a memory location of a variable. When parameters are passed by reference, no creation of a new memory location, unlike what happens in the pass by value. Reference parameters actually reference the same memory location as the actual parameters being supplied to the method.

To declare reference parameters, we use the *ref* keyword. Let us demonstrate this using an example:

```
using System;
namespace MethodApplication {
    class NumberChecker {
        public void swap(ref int a, ref int b) {
            int temp;

            temp = a; /* saving the value of a */
            a = b;   /* putting b into a */
            b = temp; /* putting temp into b */
        }
    }
}
```

```
static void Main(string[] args) {
    NumberChecker n = new NumberChecker();

    /* defining a local variable */
    int x = 10;
    int y = 20;

    Console.WriteLine("Before swapping, value of x is : {0}", x);
    Console.WriteLine("Before swapping, value of y is : {0}", y);

    /* let's swap the values by calling the function */
    n.swap(ref x, ref y);

    Console.ReadLine();
}

}
```

The code returns the following:

```
Before swapping, value of x is : 10
Before swapping, value of y is : 20
After swapping, value of x is : 20
After swapping, value of y is : 10
```

In the pass by value, the values were not swapped. However, in this case, the values have been swapped as shown above. The above change is a reflection

is reflected in the *Main* function.

## Pass By Output<sup>[122](#)</sup>

A return statement can help us to return a single value only from a function. However, by use of *output parameters*, it is possible for on to return two values from a function. Output parameters are the same as reference parameters, with the difference being that they transfer data out of the method instead of into it.

Here is an example demonstrating this:

```
using System;
namespace MethodApplication {
    class NumberChecker {
        public void getValue(out int a) {
            int temp = 20;
            a = temp;
        }
        static void Main(string[] args) {
            NumberChecker n = new NumberChecker();

            /* defining a local variable */
            int x = 10;

            Console.WriteLine("Before calling the method, value of x : {0}",
x);
            /* call the function to get value */
            n.getValue(out x);

            Console.WriteLine("After calling the method, the value of x is :
{0}", x);
            Console.ReadLine();
        }
    }
}
```

It returns the following:

Before calling the method, value of x : 10

After calling the method, the value of x is : 20

The variable that is supplied for the output parameter should be assigned a value. Output parameters are very useful when one wants to return values from a method via the parameters without assigning initial value to the parameter. The following example will help you understand this better:

```
using System;
namespace MethodApplication {
    class NumberChecker {
        public void getValues(out int a, out int b ) {
            Console.WriteLine("Enter in your first value: ");
            a = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("Enter in your second value: ");
            b = Convert.ToInt32(Console.ReadLine());
        }
        static void Main(string[] args) {
            NumberChecker n = new NumberChecker();

            /* defining a local variable */
            int x , y;

            /* call a function to obtain the values */
            n.getValues(out x, out y);

            Console.WriteLine("After calling the method, the value of x is :
{0}", x);
            Console.WriteLine("After calling the method, the value of y is :
{0}", y);
            Console.ReadLine();
        }
    }
}
```

You will be prompted to enter the two values, so do those. You will then be provided with their values before and after calling the method.

# Chapter 10: The Array World<sup>[123](#)</sup>

We can define an array as a special data type that stores a fixed number of values in a sequential manner and by use of a special syntax<sup>[124](#)</sup>. All the array elements must belong to the same data type such as a string, integer, double etc. You can think of an array as a collection of variables of the same type stored in contiguous memory locations. You use an index to get access to a given element within an array.

In the memory, the lowest address identifies the first element in the array while the highest address identifies the highest element in the array.

## Declaring Arrays<sup>[125](#)</sup>

The declaration of arrays in C# is done using the following syntax:

**datatype[] arrayName;**

The *datatype* helps us specify the data type of the elements that are stored in the array, not forgetting that all the array elements must belong to the same data type. The square brackets [] help in stating the rank of the array, where the rank denotes the size or the number of elements to be stored in the array. The *arrayName* denotes the name of the array. The following are valid examples of array declarations:

**int[] intArray; // may be used for storing int values**

**bool[] boolArray; // may be used for storing boolean values**

**string[] stringArray; // may be used for storing string values**

**double[] doubleArray; // may be used for storing double values**

**byte[] byteArray; // may be used for storing byte values**

**Employee[] customDepartmentArray; // may be used for storing instances of Employee class**

## Array Initialization

When an array has been declared, it doesn't mean that it has already been initialized in the memory. After initializing an array, it is possible to assign

values to it.

Array initialization can be done at the time of its declaration using the *new* keyword. When this keyword is used, an instance of the array is created. This is demonstrated below:

```
public class MyArray
public static void Main()
{
    int[] intArray_1 = new int[4];

    int[] intArray_2 = new int[4]{10, 20, 30, 40};

    int[] intArray_3 = {10, 20, 30, 40};
}
```

First, we have declared an array to store 4 integers. Note that the specification of the array size has been done within the square brackets. We have also done the same thing in our second statement, but values have also been assigned to the indices within the curly braces {}. In the third statement, we have declared an array and assigned values to it without specifying its size.

### Late Initialization

It is possible for us to initialize an array after it has been declared. This means that it is not a must for us to do both the declaration and the initialization of an array at the same time. Here is an example:

```
using System;
public class MyArray
{
    public static void Main()
    {
        string[] array1, array2;

        array1 = new string[4]{ "Nicholas",
            "Michelle",
            "John",
            "Claire",
```

```
};

array2 = new string[]{"Nicholas",
    "Michelle",
    "John",
    "Claire",
};
Console.WriteLine(array1.Length);
Console.WriteLine(array2.Length);
}
}
```

The above mechanism is known as *late initialization*. The initialization in this case must be done using the *new* keyword. We cannot initialize the array by simply assigning values to it. The following example shows an invalid initialization of an array:

```
string[] myarray;

myrray = {"Nicholas", "Michelle", "John", "Claire"};
```

### Accessing Array Elements

The elements of an array were assigned during the initialization time. However, it is possible for us to assign values to an array using the individual indexes. This is demonstrated below:

```
using System;
public class MyProgram
{
    public static void Main()
    {
        int[] array1 = new int[5];
```

```
        array1[0] = 1;
```

```
        array1[1] = 2;
```

```
array1[2] = 3;  
  
array1[3] = 4;  
array1[4] = 5;  
Console.WriteLine(array1[0]);  
Console.WriteLine(array1[1]);  
Console.WriteLine(array1[2]);  
Console.WriteLine(array1[3]);  
Console.WriteLine(array1[4]);  
}  
}
```

The code will return the following result:



The retrieval or access to the array values is also done using the indexes. The index of the element is passed within brackets. Consider the example given below:

```
array1[0];
array1[1];
array1[2];
array1[3];
array1[4];
```

That is how we can access all the elements of the above array named *array1*.

### Using **for** Loop

We can use a *for* loop to access the elements of an array. The loop will iterate through the elements of the array while accessing the required ones via their indices. For example:

```
using System;
namespace ArrayApp {
    class MyArray {
        static void Main(string[] args) {
            int [] n = new int[10]; /* n is an array of 10 integers */
            int x, y;
            /* initialize the elements of the array n */
            for ( x = 0; x < 10; x++ ) {
                n[ x ] = x + 50;
            }
            /* output each array element's value */
            for (y = 0; y < 10; y++ ) {
                Console.WriteLine("Element at index[{0}] = {1}", y, n[y]);
            }
            Console.ReadKey();
        }
    }
}
```

The *for* loop for the variable *x* helped us fill 10 values into the array, with the first element being 50 and the last one being 59. The *for* loop for variable *y* helped us access all the elements of the array right from index 0 to the last index. The output from the program is given below:

```
Element at index[0] = 50
Element at index[1] = 51
Element at index[2] = 52
Element at index[3] = 53
Element at index[4] = 54
Element at index[5] = 55
Element at index[6] = 56
Element at index[7] = 57
Element at index[8] = 58
Element at index[9] = 59
```

### Using *foreach* Loop

You have known how to use a *for* loop to access all the elements of an array. A *foreach* loop can also help you to iterate through the elements of an array. The following example demonstrates how to use a *foreach* loop to iterate through the elements of an array:

```
using System;
namespace ArrayApp {
    class MyArray {
        static void Main(string[] args) {
            int [] n = new int[10]; /* n is an array of 10 integers */
            //int x, y;
```

```
/* initialize the elements of the array n */
for (int x = 0; x < 10; x++ ) {
    n[ x ] = x + 100;
}
/* output the values of all array elements */
foreach (int y in n ) {
    int x = y-100;
    Console.WriteLine("Element[{0}] = {1}", x, y);
}
Console.ReadKey();
}
}
```

The code gives the following result:

The *for* loop helped us fill 10 values into the array, with the first element being 100 and the last one being 109. The *foreach* loop helped us access all the elements of the array right from index 0 to the last index. Note that the variable *x* has been used for filling the array values while the variable *y* has been used for iterating through the elements of the array. The code returns the output given below:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

## **multidimensional Arrays**

The concept of multidimensional arrays is supported in C#. Such arrays are also known as rectangular arrays. A multidimensional array is a two-dimensional series organized in the form of rows and columns. The following is an example of a multidimensional array:

```
int[,] array1 = new int[3,2]{  
    {1, 5},  
    {2, 5},  
    {5, 7}  
};
```

```
// or  
int[,] array1 = { {1, 5}, {2, 5}, {5, 7} };
```

As you can see in the above example, to initialize a multidimensional array, you must define it in terms of the number of rows and the number of columns. The [3,2] means that the array will have 3 rows and 2 columns.

To access the elements of a multidimensional array, we use two indexes. The first index identifies the row while the second one identifies the column. Note that both indexes begin from 0. Let us demonstrate this using an example:

```
using System;  
public class MyArray  
{  
    public static void Main()  
    {  
        int[,] array1 = new int[3,2]{  
            {1, 5},  
            {2, 5},  
            {5, 7}  
  
        };  
  
        Console.WriteLine(array1[0, 0]);  
  
        Console.WriteLine(array1[0, 1]);  
  
        Console.WriteLine(array1[1, 0]);  
  
        Console.WriteLine(array1[1, 1]);  
  
        Console.WriteLine(array1[2, 0]);  
  
        Console.WriteLine(array1[2, 1]);  
    }  
}
```

The code gives us the following output:



That is how we can access the elements. The `array1[2,1]` returns the element located at row 2 and column 1 of the array.

### Jagged Arrays

A jagged array is simply an array of arrays. These arrays are storing arrays instead of data type values directly. To initialize a jagged array, we use two square brackets `[][]`. In the first bracket, we specify the size of the array while in the second bracket, we specify the dimension of the array which is to be stored as values. Here is an example to show the declaration and initialization of a jagged array:

```
using System;
public class MyArray
{
    public static void Main()
    {
        int[][] jaggedArray = new int[2][];
        jaggedArray[0] = new int[4]{1, 2, 3, 4};
    }
}
```

```
jaggedArray[1] = new int[3]{5, 6, 7};  
  
Console.WriteLine(jaggedArray[0][0]);  
  
Console.WriteLine(jaggedArray[0][2]);  
  
Console.WriteLine(jaggedArray[1][1]);  
}  
}
```

We have declared and initialized our jagged array in the above code. The code will give you the following result:



A jagged array can also store a multidimensional array as a value. We can use [,] in the second bracket to indicate a multi-dimension. Here is an example:

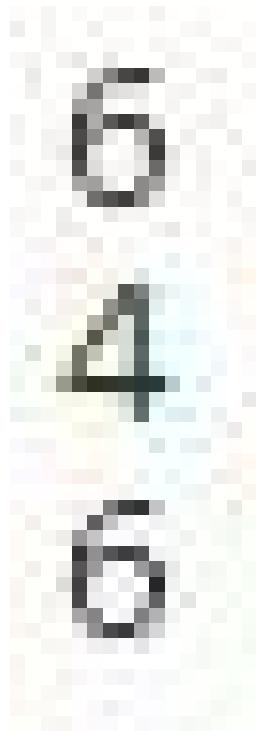
```
using System;  
public class MyArray  
{
```

```
public static void Main()
{
    int[,] jaggedArray = new int[3][,];
    jaggedArray[0] = new int[3, 2] { { 1, 5 }, { 2, 6 }, { 4, 8 } };
    jaggedArray[1] = new int[2, 2] { { 3, 5 }, { 4, 6 } };
    jaggedArray[2] = new int[2, 2];

    Console.WriteLine(jaggedArray[0][1,1]);
    Console.WriteLine(jaggedArray[1][1,0]);

    Console.WriteLine(jaggedArray[1][1,1]);
}
```

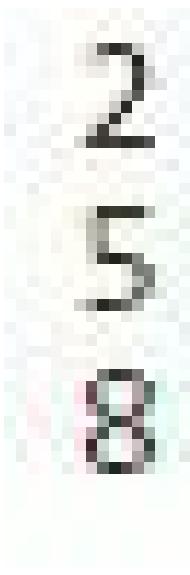
The code will return the following:



In case an additional bracket is added, then this will become an array of array of array. This is demonstrated in the following example:

```
using System;
public class MyArray
{
    public static void Main()
    {
        int[][][] jaggedArray = new int[2][][]
        {
            new int[2][]
            {
                new int[3] { 2, 3, 5},
                new int[2] { 1, 5}
            },
            new int[1][]
            {
                new int[3] { 7, 6, 8}
            }
        };
    }
    Console.WriteLine(jaggedArray[0][0][0]);
    Console.WriteLine(jaggedArray[0][1][1]);
    Console.WriteLine(jaggedArray[1][0][2]);
}
}
```

The code returns the following:



Note that we have used three square brackets `[][][]`, which means an array of array of array. The *jaggedArray* will have 2 elements, which are 2 arrays. Each of these arrays will also have a single dimension array.

## Chapter 11: Classes<sup>[126](#)</sup>

A class can be seen as a blueprint for an object. Objects in the real world have characteristics like shape, color and functionalities. For example, X6 is an object of car type. A car has characteristics like color, speed, interior, shape etc. This means that any company that creates an object with the above characteristics will be of type car. This means that the Car is a class while each object, that is, a physical car, will be an object of type Car.

In object-oriented programming (not forgetting that C# is an object-oriented programming language) a class has fields, properties, methods, events etc. A class should define the types of data and the functionality that the objects should have.

With a class, you can create your own custom types by grouping variables of other types together as well as methods and events.

In C#, we use the *class* keyword to define a class<sup>[127](#)</sup>. Here is a simple example of this:

```
public class TestClass
{
    public string field1 = string.Empty;

    public TestClass()
    {
    }

    public void TestMethod(int param1, string param2)
    {
        Console.WriteLine("The first parameter is {0}, and second
parameter is {1}",
                           param1, param2);
    }

    public int AutoImplementedPropertyTest { get; set; }

    private int propertyVar;
    public int PropertyTest
```

```

{
    get { return propertyVar; }
    set { propertyVar = value; }
}
}

```

The **public** keyword before the class is an Access Specifier, specifying how the class will be accessed. By being public, it means that it will be accessible by all other classes within the same project. We have given the class the name *TestClass*.

We have also defined a field in the class named *field1*. Below this, we have created a constructor for the class. Note that the constructor takes the same name as the class itself, hence the constructor's name is *TestClass()*. Inside this class, we have also defined a method named *TestMethod()*, and this method takes in two parameters, *param1* and *param2*, with the former being an integer and the latter being a string.

Here is another example demonstrating how to declare and use a class:

```

using System;
namespace CubeApplication {
    class Cube {
        public double length; // Length of the cube
        public double breadth; // Breadth of the cube
        public double height; // Height of the cube
    }
    class Cubetester {
        static void Main(string[] args) {
            Cube Cube1 = new Cube(); // Declare Cube1 of type Cube
            Cube Cube2 = new Cube(); // Declare Cube2 of type Cube
            double volume = 0.0; // Store the cube volume here
            // cube 1 specification
            Cube1.height = 4.0;
            Cube1.length = 5.0;
            Cube1.breadth = 8.0;
            // cube 2 specification
            Cube2.height = 8.0;
            Cube2.length = 12.0;
        }
    }
}

```

```

        Cube2.breadth = 14.0;
        // volume of cube 1
        volume = Cube1.height * Cube1.length * Cube1.breadth;
        Console.WriteLine("Volume of Cube1 : {0}", volume);
        // volume of cube 2
        volume = Cube2.height * Cube2.length * Cube2.breadth;
        Console.WriteLine("Volume of Cube2 : {0}", volume);
        Console.ReadKey();
    }
}
}

```

The code will return the following result:

```

Volume of Cube1 : 160
Volume of Cube2 : 1344

```

## Encapsulation and Member Functions

A member function for a class is simply a function with a definition or prototype within the definition of the class in the same way as any other function. Such a function can operate on any object of the class in which it is a member, and it can access all the class members for the object.

Member functions are simply the attributes of the object (from a design perspective) and they are defined as *private* so as to implement the concept of encapsulation. We can only access such variables using public member functions. Let's demonstrate how we can set and access the various members of a class in C#:

```

using System;
namespace CubeApplication {

```

```
class Cube {
    private double length; // Length of a cube
    private double breadth; // Breadth of a cube
    private double height; // Height of a cube
    public void setLength( double len ) {
        length = len;
    }
    public void setBreadth( double brea ) {
        breadth = brea;
    }
    public void setHeight( double heig ) {
        height = heig;
    }
    public double getVolume() {
        return length * breadth * height;
    }
}
class Cubetester {
    static void Main(string[] args) {
        Cube Cube1 = new Cube(); // Declare Cube1 of type Cube
        Cube Cube2 = new Cube();
        double volume;

        // Declare Cube2 of type Cube
        // cube 1 specification
        Cube1.setLength(4.0);
        Cube1.setBreadth(6.0);
        Cube1.setHeight(8.0);

        // cube 2 specification
        Cube2.setLength(10.0);
        Cube2.setBreadth(14.0);
        Cube2.setHeight(12.0);
        // volume of cube 1
        volume = Cube1.getVolume();
        Console.WriteLine("Volume of Cube1 is: {0}" ,volume);
    }
}
```

```
// volume of cube 2
volume = Cube2.getVolume();
Console.WriteLine("Volume of Cube2 is: {0}", volume);
Console.ReadKey();
}
}
}
```

Here is the output from the code:

```
Volume of Cube1 is: 192
Volume of Cube2 is: 1680
```

We used the *setter* methods to set the values of the various attributes of our two cubes. The *getVolume()* function has been called to calculate the volumes of the two cubes.

## Constructors

A constructor is simply a special member function of a class that is run anytime that we create new objects of the class. A constructor assumes the class name and it should not have a return type. The following example demonstrates how to use a constructor in C#:

```
using System;
namespace ConstructorApplication {
    class Person {
        private double height; // height of the person

        public Person() {
            Console.WriteLine("We are creating an object");
```

```
        }
    public void setHeight( double heig ) {
        height = heig;
    }
    public double getHeight() {
        return height;
    }
    static void Main(string[] args) {
        Person p = new Person();
        // set the person's height
        p.setHeight(7.0);
        Console.WriteLine("The height of the person is: {0}",
p.getHeight());
        Console.ReadKey();
    }
}
}
```

The code should return the following:

We are creating an object  
The height of the person is: 7

A default constructor has no parameters, but it is possible to add parameters to a constructor. Such a constructor is known as a *parameterized constructor*. With such a technique, it is possible for one to assign an initial value to an object during the time of its creation. Here is an example:

```
using System;
namespace ConstructorApplication {
    class Person {
        private double height; // Height of the person
        public Person(double heig) { // A parameterized constructor
```

```

        Console.WriteLine("We are creating an object, height = {0}",
heig);
    height = heig;
}
public void setHeight( double heig ) {
    height = heig;
}
public double getHeight() {
    return height;
}
static void Main(string[] args) {
    Person p = new Person(8.0);
    Console.WriteLine("The height of the person is : {0}",
p.getHeight());
    // set the height
    p.setHeight(7.0);
    Console.WriteLine("The height of the person is : {0}",
p.getHeight());
    Console.ReadKey();
}
}
}

```

Here is the output from the code:

```

We are creating an object, height = 8
The height of the person is : 8
The height of the person is : 7

```

## Destructors

A destructor refers to a special member function of a class that is run anytime an object of the class goes out of scope. A destructor takes the same

name as a class but it should be preceded by a tilde (~). A destructor cannot take parameters neither can it return a value.

A destructor is a useful tool for releasing the memory resources before leaving a program. You can overload or inherit a destructor. The following example demonstrates how to use a destructor:

```
using System;
namespace ConstructorApplication {
    class Person {
        private double height; // Height of a person

        public Person() { // A constructor
            Console.WriteLine("We are creating an object");
        }
        ~Person() { //A destructor
            Console.WriteLine("We are deleting an object");
        }
        public void setHeight( double heig ) {
            height = heig;
        }
        public double getHeight() {
            return height;
        }
        static void Main(string[] args) {
            Person p = new Person();
            // set the height of the person
            p.setHeight(7.0);
            Console.WriteLine("The height of the person is : {0}",
                p.getHeight());
        }
    }
}
```

Here is the output from the function:

We are creating an object

The height of the person is : 7

We are deleting an object

### Static Members

To define a class member as static, we use the *static* keyword. When a class member is declared as static, it means that regardless of the number of objects of the class that are created, there exists only one copy of the static member.

The use of the *static* keyword means that there is only one instance of a member existing in the class. We use this keyword when we need to declare constants since their values can be retrieved by invocation of the class without the creation of an instance of the same. We can initialize static variables outside a class definition or a member function. Static variables can also be initialized inside a class definition.

Let us demonstrate the use of static variables using an example:

```
using System;
namespace StaticApp {
    class StaticVariables {
        public static int x;

        public void count() {
            x++;
        }
        public int getX() {
            return x;
        }
    }
}
```

```
class StaticTester {
    static void Main(string[] args) {
        StaticVariables var1 = new StaticVariables();
        StaticVariables var2 = new StaticVariables();
        var1.count();
        var1.count();
        var1.count();
        var2.count();
        var2.count();
        Console.WriteLine("Variable x for var1 is: {0}", var1.getX());
        Console.WriteLine("Variable x for vars is: {0}", var2.getX());
        Console.ReadKey();
    }
}
```

The code returns the following result:

```
Variable x for var1 is: 6
Variable x for vars is: 6
```

A member function can also be declared as static. Such a function will only be able to access static variables. Static functions exist even before the creation of the object. Static functions can be used as demonstrated in the following example:

```
using System;
namespace StaticAppli {
    class StaticVariable {
        public static int x;
        public void count() {
```

```
        x++;
    }
    public static int getX() {
        return x;
    }
}
class StaticTester {
    static void Main(string[] args) {
        StaticVariable var = new StaticVariable();

        var.count();
        var.count();
        var.count();
        Console.WriteLine("Variable x is: {0}", StaticVariable.getX());
        Console.ReadKey();
    }
}
```

The code will return the following:



Variable x is: 3

# Chapter 12: Structure

A structure in C# is a value type data type. With a structure, you can use a single variable to hold related data belonging to different data types. In C#, we use the *struct* keyword to create a structure<sup>[128](#)</sup>.

Structures are used for tracking records. A good example is when you need to keep a record of all class students or all company employees. For the case of storing student records, some of the details that you may need to track including the names, age, course, date of enrollment, date of completion, amount of fee paid, fee balance etc, guardian address and mobile phone number etc. All these details belong to different data types, meaning that you will be required to create a variable for each of them. However, by use of a structure, you can define a single variable and keep these details together as one.

Here is an example of declaring a structure<sup>[129](#)</sup>:

```
struct Employee{  
    public string name;  
    public string department;  
    public string title;  
    public int age;  
    public double salary;  
};
```

To initialize a struct, we can choose the *new* keyword or not. The members of a struct can be assigned values as shown below:

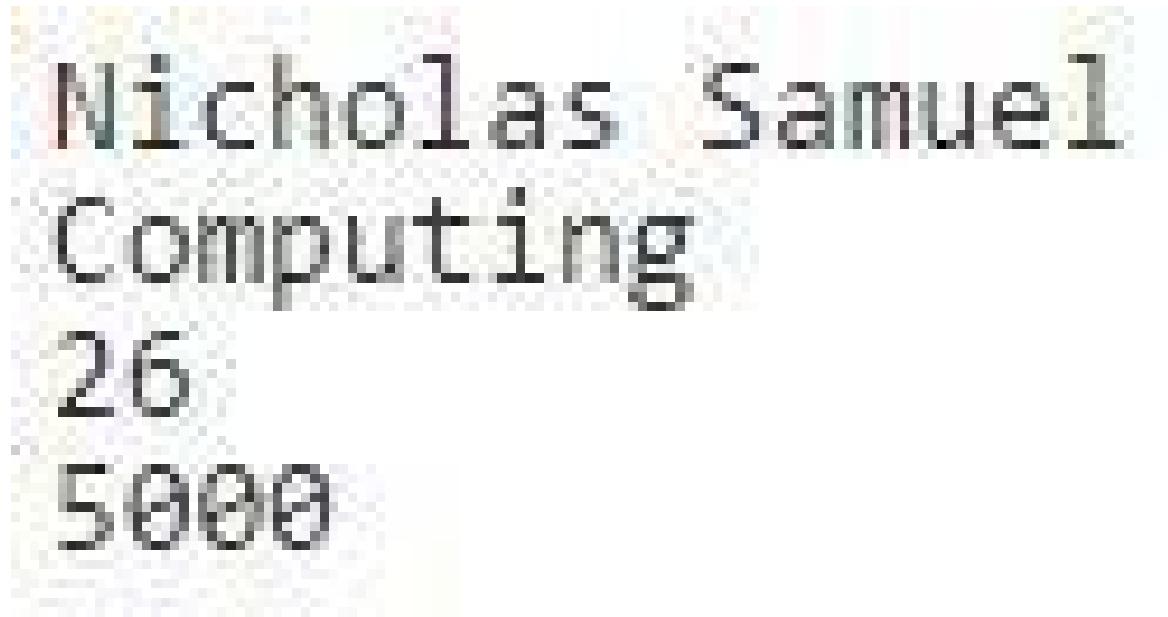
```
using System;
```

```
public class MyStruct  
{  
    public static void Main()  
    {  
        Employee emp = new Employee()  
        emp.name = "Nicholas Samuel";
```

```
emp.department = "Computing";
emp.age = 26;
emp.salary = 5000;
Console.WriteLine(emp.name);
Console.WriteLine(emp.department);
Console.WriteLine(emp.age);
Console.WriteLine(emp.salary);
}
}
```

```
public struct Employee
{
    public string name { get; set; }
    public string department { get; set; }
    public string title { get; set; }
    public int age { get; set; }
    public double salary { get; set; }
}
```

The code gives the following result:



The struct named *Employee* was defined using the *struct* keyword. The various details of this struct have been declared within this. At the top of the class, we created an instance of this struct and we gave it the name *emp*. This

instance has been used to access the various attributes of the struct for display on the screen.

Note that a struct is a value type, and this makes it faster when compared to a class object. This has made them good for use in game programming. However, one can easily transfer a class object than a struct. This means that a struct should not be used when one is need of transferring data to other classes.

Here is another example of a struct in C#:

```
using System;
struct Cars {
    public string model;
    public int cc;
    public int passengers;
    public int year;
};

public class carStructure {
    public static void Main(string[] args) {
        Cars Premio; /* Declare prenio of type Car */
        Cars X6; /* Declare X6 of type Car */
        /* premio specification */
        Premio.model = "Saloon";
        Premio.cc = 1800;
        Premio.passengers = 5;
        Premio.year = 2010;
        /* X6 specification */
        X6.model = "Saloon";
        X6.cc = 3500;
        X6.passengers = 5;
        X6.year = 2012;
        /* print premio info */
        Console.WriteLine("Premio model: {0}", Premio.model);
        Console.WriteLine("Premio CC : {0}", Premio.cc);
        Console.WriteLine("Premio passengers : {0}", Premio.passengers);
        Console.WriteLine("Premio year :{0}", Premio.year);
        /* print X6 info */
    }
}
```

```
Console.WriteLine("X6 model: {0}", X6.model);
Console.WriteLine("X6 CC : {0}", X6.cc);
Console.WriteLine("X6 passengers : {0}", X6.passengers);
Console.WriteLine("X6 year :{0}", X6.year);
Console.ReadKey();
}
}
```

The code will return the following details<sup>130</sup>:

```
Premio model: Saloon
Premio CC : 1800
Premio passengers : 5
Premio year :2010
X6 model: Saloon
X6 CC : 3500
X6 passengers : 5
X6 year :2012
```

```
using System;
namespace StructureExample
{
    public struct Cars
    {
        public int Premio;
        public int X6;
    }
}

class Program
{
    static void Main()
    {
        Cars car = new Cars();
        car.Premio = 1000000;
        car.X6 = 1200000;
        Console.WriteLine("Car details");
        Console.WriteLine("Premio value: " + car.Premio);
        Console.WriteLine("X6 value: " + car.X6);
    }
}
```

We have created a single struct named *Cars*. In this structure, we have stored the two members, Premio and X6. These two instances are instances of the structure. The properties for these had been defined inside the structure. The two instances share properties, but these will take different values. We have then printed the values of these properties as shown in the above output.

## Characteristics of Structures

So far, you have known how to use structures and some of their characteristics. The structures in C# have a great difference from the structures supported in C and C++. C# structures have the features described below:

- Structures may have fields, indexers, methods, properties, operators and events.
- A structure can have defined constructors, not destructors. However, in a structure, we are not allowed to define a default

constructor. The reason is that this is defined automatically and one cannot change it.

- A structure cannot inherit another structure or a class.
- With a structure, we can implement one or more interfaces.
- The members of a structure cannot be specified as virtual abstract or protected.
- After the creation of a Struct object via the *New* keyword, the struct object is created and the necessary constructor is called. Unlike classes, we can instantiate a struct without the use of the *New* keyword.
- If we don't use the *New* operator, the fields will not be assigned and it will not be possible to use the object until an assignment has been done to all the fields.

## **Struct vs. Class**

Here the differences between Structs and Classes:

- A Struct is explained as a value type. On the other side, a class is defined as a reference type.
- Structs don't support inheritance. Classes do.
- A Struct cannot have a default constructor. A class can have.

Let us create an example in relation to the above differences:

```
using System;
struct Cars {
    private string model;
    private int cc;
    private int passengers;
    private int year;
    public void getValues(string m, int c, int p, int yr) {
        model = m;
    }
}
```

```

    cc = c;
    passengers = p;
    year = yr;
}
public void show() {
    Console.WriteLine("Car model : {0}", model);
    Console.WriteLine("CC : {0}", cc);
    Console.WriteLine("Passengers : {0}", passengers);
    Console.WriteLine("Year :{0}", year);
}
};

public class myStructure {
    public static void Main(string[] args) {
        Cars Premio = new Cars(); /* Declare Premio of type Cars */
        Cars X6 = new Cars(); /* Declare X6 of type Cars */
        /* Premio specification */
        Premio.getValues("Saloon",
        1800, 5, 2012);
        /* X6 specification */
        X6.getValues("Saloon",
        3500, 5, 2012);
        /* print Premio info */
        Premio.show();
        /* print X6 info */
        X6.show();
        Console.ReadKey();
    }
}

```

The code will give the following output after execution:

Car model : Saloon

CC : 1800

Passengers : 5

Year : 2012

Car model : Saloon

CC : 3500

Passenger : 5

Year : 2012

# Chapter 13: Encapsulation

Encapsulation is the process of enclosing items within a logical or physical package. In object-oriented programming, encapsulation is used to prevent access to the implementation details.

Encapsulation and Abstraction are closely related features in object-oriented programming. The purpose of abstraction is to make the relevant details visible to users while the purpose of encapsulation is to enable a programmer to implement the required level of abstraction.

To implement encapsulation<sup>[131](#)</sup>, we use *access specifiers*. The role of an access specifier is to state the visibility and scope of a class member. There are different types of access specifiers that are supported in C#. They include the following:

- Public
- Protected
- Private
- Protected internal
- Internal

## Private Access Specifier

With a private access specifier, a class is able to hide its member functions and member variables from other objects and functions. Only functions of a similar class are able to access its private members. An instance of the class is not able to access the private members of the class. Consider the example given below:

```
using System;
namespace AccessApplication {
    class Figure {
        // The class member variables
```

```

private double width;
private double length;
public void GetDetails() {
    Console.WriteLine("Enter the Length: ");
    length = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter the Width: ");
    width = Convert.ToDouble(Console.ReadLine());
}
public double CalculateArea() {
    return length * width;
}
public void Show() {
    Console.WriteLine("The Length of the figure is: {0}", length);
    Console.WriteLine("The Width of the figure is: {0}", width);
    Console.WriteLine("The Area of the figure is: {0}",
CalculateArea());
}
//end the class Figure
class RunFigure {
    static void Main(string[] args) {
        Figure f = new Figure();
        f.GetDetails();
        f.Show();
        Console.ReadLine();
    }
}
}

```

Run the code and enter the measurements of the figure, both the width and the length. You will get the area of the figure.

The member variables, that is, the *width* and the *length* have been defined as *private*. This means that we cannot access them from the *Main()* function. The *GetDetails()* and *Show()* member functions have been declared as *public*, hence they are able to access these variables. We can also access them from the *Main()* function after creating an instance of the *Figure* class. This instance has been given the name *f*.

## Public Access Specifier

With a public access specifier, a class is able to expose its member functions and member variables to the other objects and functions. Any class member declared as public can be accessed from outside of that class. Consider the following example:

```
using System;
namespace FigureApplication {
    class Figure {
        // The class member variables
        public double width;
        public double length;
        public double CalculateArea() {
            return length * width;
        }
        public void Show() {
            Console.WriteLine("The Length of the figure is: {0}", length);
            Console.WriteLine("The Width of the figure is: {0}", width);
            Console.WriteLine("The Area of the figure is: {0}",
                CalculateArea());
        }
    }//end the Figure class
    class RunFigure {
        static void Main(string[] args) {
            Figure f = new Figure();
            f.length = 5.4;
            f.width = 3.2;
            f.Show();
            Console.ReadLine();
        }
    }
}
```

The following is the output from the code:

The Length of the figure is: 5.4

The Width of the figure is: 3.2

The Area of the figure is: 17.28

The class member variables *width* and *length* have been declared as *public*, which means that we are able to access them from the *Main()* function after creating an instance of the class. We have created an instance of the *Figure* class and given it the name *f*.

The *CalculateArea()* and *Show()* member functions are also able to access these member variables without using an instance of the class. The *Show()* member function has also been declared as public, meaning that we are able to access it from the *Main()* function using an instance of the class.

### **Protected Access Specifier**

This type of access specifier makes it possible for a child class to access the member functions and member variables that have been defined in the base class. This way, it is easier to implement inheritance. This will be discussed in inheritance.

### **Internal Access Specifier**

This type of access specifier allows a class to expose its member functions and member variables to the other objects and functions in the current assembly. This means that any member that has been created using the internal access specifier is accessible from any class or method that has been defined within the application where the member has been defined. This is demonstrated in the following example:

```
using System;  
namespace AccesserApplication {  
    class Figure {
```

```
// The member variables
internal double width;
internal double length;
double CalculateArea() {
    return length * width;
}
public void Show() {
    Console.WriteLine("The Length of the figure is: {0}", length);
    Console.WriteLine("The Width of the figure is: {0}", width);
    Console.WriteLine("The Area of the figure is: {0}",
CalculateArea());
}
}//end the class Figure

class RunFigure {
    static void Main(string[] args) {
        Figure f = new Figure();
        f.length = 6.5;
        f.width = 4.8;
        f.Show();
        Console.ReadLine();
    }
}
```

The code should return the following result:

The Length of the figure is: 6.5  
The Width of the figure is: 4.8  
The Area of the figure is: 31.2

# Chapter 14: Inheritance

With inheritance, it is possible for a programmer to define one class in terms of another class. This also makes it possible for us to reuse code and shorten the time taken to implement an application.

During the creation of a class, instead of having to create completely new member functions and data members, the programmer is able to designate that the new class inherits the members of an already existing class. The *base class* is the name for the existing class while the new class is known as the *derived class*.

Inheritance is simply an implementation of IS-A relationship<sup>[132](#)</sup>. For example, Cow is a Mammal.

It is possible for a class to inherit from more than one class or interfaces, meaning that it can inherit data and functions from many base interfaces or classes. The following example demonstrates the concept of the base and derived class:

```
using System;
namespace InheritanceApp {
    class Figure {
        public void setWidth(int wid) {
            width = wid;
        }
        public void setHeight(int heig) {
            height = heig;
        }
        protected int width;
        protected int height;
    }
    // A Derived class
    class Rectangle: Figure {
        public int calculateArea() {
            return (width * height);
        }
    }
}
```

```

    }
}

class TestInheritance {
    static void Main(string[] args) {
        Rectangle R = new Rectangle();
        R.setWidth(6);
        R.setHeight(8);
        // Print the rectangle's area.
        Console.WriteLine("The area is: {0}", R.calculateArea());
        Console.ReadKey();
    }
}

```

The code should return the following output:

We have defined the *Figure* class with two properties namely *width* and *height*. This class also has two methods, *setWidth()* and *SetHeight()*. We have then defined a class named *Rectangle*. Notice the syntax we have used to create this class:

### **class Rectangle: Figure {**

The use of the full colon signals that the *Rectangle* class is inheriting the *Figure* class. The *Rectangle* class only has one method, the *calculateArea()* method. However, since it has inherited the *Figure* class, it means that it has all the properties of the *Figure* class, like the width and the height.

We have then created the *TestInheritance* class. Within the method *Main()*, an instance of the *Rectangle* class has been created and given it the name *R*. We have used this instance to access the properties that have been defined in both the *Rectangle* and the *Figure* classes. That is how powerful inheritance is!

### **Base Class Initialization**

The derived class inherits the member methods and member variables defined in the base class. This means that we should create the super class object before creating the subclass. Instructions for initialization of the superclass can be given in the initialization of the list of members.

This is demonstrated in the program given below:

```
using System;
namespace InheritanceApp {
    class Object {
        // The member variables
        protected double width;
        protected double length;
        public Object(double len, double wid) {
            width = wid;
            length = len;
        }
        public double CalculateArea() {
            return length * width;
        }
        public void Show() {
            Console.WriteLine("The Width of the figure is: {0}", width);
            Console.WriteLine("The Length of the figure is: {0}", length);
            Console.WriteLine("The Area of the figure is: {0}",
CalculateArea());
        }
    }//end class Figure
    class Carpet : Object {
        private double cost;
        public Carpet(double len, double wid) : base(len, wid) { }
        public double CalculateCost() {
            double cost;
            cost = CalculateArea() * 120;
            return cost;
        }
        public void Show() {
            base.Show();
        }
    }
}
```

```

        Console.WriteLine("The total cost for the carpet is: {0}",
CalculateCost());
    }
}
class RunObject {
    static void Main(string[] args) {
        Carpet c = new Carpet(7.5, 9.5);
        c.Show();
        Console.ReadLine();
    }
}
}

```

The code will return the following output:

```

The Width of the figure is: 9.5
The Length of the figure is: 7.5
The Area of the figure is: 71.25
The total cost for the carpet is: 8550

```

## Multiple Inheritance

Multiple inheritance is not supported in C#. However, with interfaces, it is possible for us to implement multiple inheritance. Consider the following example:

```

using System;
namespace InheritanceApp {
    class Object {
        public void setWidth(int wid) {
            width = wid;
        }
        public void setHeight(int heig) {

```

```

        height = heig;
    }
    protected int width;
    protected int height;
}

// PaintCost for the Base class
public interface PaintCost {
    int calculateCost(int area);
}

// The derived class
class Board : Object, PaintCost {
    public int calculateArea() {
        return (width * height);
    }
    public int calculateCost(int area) {
        return area * 120;
    }
}
class InheritanceTester {
    static void Main(string[] args) {
        Board B = new Board();
        int area;
        B.setWidth(5);
        B.setHeight(7);
        area = B.calculateArea();
        // Show the area of the object
        Console.WriteLine("The area of the object is: {0}",
B.calculateArea());
        Console.WriteLine("The total printing cost is: ${0}" ,
B.calculateCost(area));
        Console.ReadKey();
    }
}

```

The code gives the following output:

The area of the object is: 35

The total printing cost is: \$4200

# Chapter 15: Polymorphism

Polymorphism<sup>133</sup> is a term that means taking many forms. In programming, it is expressed as “**one interface, many functions**”. Polymorphism can take two forms, static or dynamic.

For the case of static dynamism, response to the function is determined during compile time. In dynamic polymorphism, response to the function is determined during runtime.

## Static Polymorphism

As stated above, the response to a function in this type of polymorphism is determined during compile time. The process of linking a function to an object during compile time is known as *early binding*. It is also known as *static binding*. In C#, static polymorphism can be implemented in two ways:

- Function overloading
- Operator overloading

## Function Overloading

It is possible for us to have multiple definitions for the same function name within one scope. The differences between these functions are implemented by types or the number of arguments that the functions take. Function declarations cannot be overloaded by differing the return types only.

**The following example demonstrates how to overload a function in C#:**  
**using System;**

```
namespace PolymorphismApp {
    class DisplayData {
        void display(int x) {
            Console.WriteLine("Printing an int value: {0}", x );
        }
        void display(double y) {
            Console.WriteLine("Printing a float value: {0}" , y);
        }
        void display(string z) {
```

```
        Console.WriteLine("Printing a string value: {0}", z);
    }
    static void Main(string[] args) {
        DisplayData dd = new DisplayData();

        // Call display function to return an integer value
        dd.display(10);
        // Call display function to return a float value
        dd.display(11.385);
        // Call display function to return a string value
        dd.display("Hello Sir/Madam");
        Console.ReadKey();
    }
}
```

The code will return the following once executed:

```
Printing an int value: 10
Printing a float value: 11.385
Printing a string value: Hello Sir/Madam
```

We have three definitions of the function *display()*. In this case, overloading has been implementing by varying the types of parameters taken by the function. In one instance, the function is taking an integer value, a float in another instance and a string in the last instance.

## Operator Overloading

Operator overloading refers to the use of a single operator to perform various operations. With operator overloading, additional functionalities can be added to the C# operators during their application on user-defined types. This is the case when either one or both the operands belong to a user-defined class.

Here is an example that demonstrates this:

```
using System;
namespace PolymorphismApp {
    class Object {
        private double breadth; // The breadth of the box
        private double length; // The Length of the box
        private double height; // The Height of the box
        public double calculateVolume() {
            return breadth * length * height;
        }
        public void setLength( double l ) {
            length = l;
        }
        public void setBreadth( double b ) {
            breadth = b;
        }
        public void setHeight( double h ) {
            height = h;
        }
        // Overload the + operator to add 2 objects.
        public static Object operator+ (Object obj1, Object obj2) {
            Object ob = new Object();
            ob.length = obj1.length + obj2.length;
            ob.breadth = obj1.breadth + obj2.breadth;
            ob.height = obj1.height + obj2.height;
            return ob;
        }
    }
    class BoxTester {
        static void Main(string[] args) {
            Object Object1 = new Object(); // Declare Object1 of type Object
            Object Object2 = new Object(); // Declare Object2 of type Object
            Object Object3 = new Object(); // Declare Object3 of type Object
            double volume = 0.0; // Store volume of the object here
            // object 1 dimensions
            Object1.setLength(7.0);
            Object1.setBreadth(8.0);
```

```
Object1.setHeight(10.0);
// Object 2 dimensions
Object2.setLength(6.0);
Object2.setBreadth(9.0);
Object2.setHeight(11.0);

// Calculate the volume of object 1
volume = Object1.calculateVolume();
Console.WriteLine("The volume of Object1 : {0}", volume);
// Calculate the volume of object 2
volume = Object2.calculateVolume();
Console.WriteLine("The volume of Object2 : {0}", volume);
// Add the two objects
Object3 = Object1 + Object2;
// Calculate the volume of object 3
volume = Object3.calculateVolume();
Console.WriteLine("The volume of Object3 : {0}", volume);
Console.ReadKey();
}

}
}
```

The code returns the following output:

```
The volume of Object1 : 560
The volume of Object2 : 594
The volume of Object3 : 4641
```

We have overloaded the + operator. The operator is being utilized for the addition of two numeric types. However, in the above example, we have used the operator to add the two objects together, that is, Object1 and Object2. This has given us Object3. The Object1 and Object2 are user-defined types, meaning that we have used the + operator to add user-defined types.

Note that it is true that we can overload operators in C#, but not all operators can be overloaded. For example, you cannot overload the comparison operators like ==, <, >, !=, >= and <=. For the case of the conditional logic operators like && and ||, we can overload them, but not directly.

## **Dynamic Polymorphism**

With C#, we can create abstract classes that are good for a partial class implementation of an interface. The completion of this is implemented once after a derived class has inherited from it. An abstract class has abstract methods, and the derived class implements these. However, the derived class has a more specialized functionality.

Note that you cannot create an instance of any class that is abstract. An abstract method cannot also be declared outside an abstract class. If a C# class is declared as *sealed*, it means that the class cannot be inherited. However, you are not allowed to declare an abstract class *sealed*. Here is an example of an abstract class:

```
using System;
namespace PolymorphismApp {
    abstract class Object {
        public abstract int area();
    }
    class Object1: Object {
        private int width;
        private int length;
        public Object1( int x = 0, int y = 0 ) {
            length = x;
            width = y;
        }
        public override int area () {

```

```

        Console.WriteLine("The Object1 area is:");
        return (width * length);
    }
}

class Object1Tester {
    static void Main(string[] args) {
        Object1 obj = new Object1(8, 6);
        double x = obj.area();
        Console.WriteLine("The area is: {0}",x);
        Console.ReadKey();
    }
}

```

The code will print the following result:

After defining a function in a class that you want to implement in an inherited class, you should use a *virtual* function.

To implement dynamic polymorphism, we use *abstract classes* and *virtual functions*. Let us demonstrate this using an example:

```

using System;
namespace PolymorphismApp {
    class Object {
        protected int width, height;
        public Object( int x = 0, int y = 0 ) {
            width = x;
            height = y;
        }
    }
}
```

```
public virtual int area() {
    Console.WriteLine("The area of the parent class is :");
    return 0;
}
class Object1: Object {
    public Object1( int x = 0, int y = 0): base(x, y) {

    }
    public override int area () {
        Console.WriteLine("The area of Object1 class is:");
        return (width * height);
    }
}
class Object2: Object {
    public Object2(int x = 0, int y = 0): base(x, y) {
    }
    public override int area() {
        Console.WriteLine("The area of Object2 class is:");
        return (width * height / 2);
    }
}
class Caller {
    public void CallArea(Object obj) {
        int x;
        x = obj.area();
        Console.WriteLine("The Area is: {0}", x);
    }
}
class TesterClass {
    static void Main(string[] args) {
        Caller c = new Caller();
        Object1 ob1 = new Object1(10, 7);
        Object2 ob2 = new Object2(10, 5);
        c.CallArea(ob1);
        c.CallArea(ob2);
        Console.ReadKey();
    }
}
```

```
    }  
}  
}
```

The code returns the following output:

The area of Object1 class is:

The Area is: 70

The area of Object2 class is:

The Area is: 25

# Chapter 16: Regular Expressions [134](#)

A regular expression is explained as a pattern that we can match against an input text. In the .Net framework, there is a regular expression engine that we can use to perform that matching. A pattern is made up of one or even more operators, character literals or constructs.

## Regex Class

This is a class used for representation of regular expressions. This class comes with a number of inbuilt functions such as the following:

1. public bool isMatch(string input)- the method specifies whether the specified regular expression in the Regex constructor gets a match in the specified input string.
2. public bool isMatch(string input, int startPoint)- the method specifies whether the specified regular expression in the Regex constructor gets a match in the specified input string from the specified location in the input string.
3. public static bool isMatch(string input, int pattern)- the method specifies whether the specified regular expression gets a match in the specified input string.
4. public MatchCollection Matches(string input, string replacement)- this replaces a match in the input string with the specified replacement string.
5. public string[] Split(string input)- this function helps in splitting an input string into an array of substrings at positions that have been defined by the regular expression pattern that is specified in a Regex constructor.

Consider the example given below:

```
using System.Text.RegularExpressions;
```

```
using System;

namespace RegularExpApp {
    class MyProgram {
        private static void findMatch(string input, string expre) {
            Console.WriteLine("The Expression is: " + expre);
            MatchCollection mc = Regex.Matches(input, expre);
            foreach (Match x in mc) {
                Console.WriteLine(x);
            }
        }
        static void Main(string[] args) {
            string s = "Splitting an Input String into Substrings";

            Console.WriteLine("Match words that begin with 'S': ");
            findMatch(s, @"\bS\S*");
            Console.ReadKey();
        }
    }
}
```

After the match, the code will return the following:

```
Match words that begin with 'S':
The Expression is: \bS\S*
Splitting
String
Substrings
```

The output shows that three words have been matched from the input string. We were matching any words that begin with S and three of them were found.

Consider the next example given below:

```
using System.Text.RegularExpressions;
using System;
namespace RegularExpApp {
    class MyProgram {
        private static void findMatch(string input, string expre) {
            Console.WriteLine("The input expression is: " + expre);
            MatchCollection mc = Regex.Matches(input, expre);
            foreach (Match x in mc) {
                Console.WriteLine(x);
            }
        }
        static void Main(string[] args) {
            string s = "so she was the same";

            Console.WriteLine("Match words beginning with 's' and ending
with 'e':");
            findMatch(s, @"\bs\S*e\b");
            Console.ReadKey();
        }
    }
}
```

The code will give the following result:

```
Match words beginning with 's' and ending with 'e':
The input expression is: \bs\S*e\b
she
same
```

In the above example, we are matching the words that begin with s and end with e. We have successfully matched two words in the input string. Here is another example:

```
using System;
using System.Text.RegularExpressions;
namespace RegularExpApp {
    class MyProgram {
        static void Main(string[] args) {
            string text = "Hi  learner  ";
            string pattern = @"\s+";
            string substitute = " ";
            Regex r = new Regex(pattern);
            string output = r.Replace(text, substitute);
            Console.WriteLine("The input string is: {0}", text);
            Console.WriteLine("The string after replacement is: {0}",
output);
            Console.ReadKey();
        }
    }
}
```

The code returns the following:

```
The input string is: Hi  learner
The string after replacement is: Hi learner
```

What we are doing in the code is that we are replacing the extra white space. There is a big space between “Hi” and “learner”. The extra one has been replaced or removed.

# **Chapter 17: The Process of Handling Exceptions**

Applications usually encounter errors during execution. After the occurrence of an error, the program throws an exception with more information regarding the error. Exceptions should be handled to prevent a program from crashing.

In C#, exceptions are handled using 4 main keywords:

1. try- this keyword identifies the block of code in which particular exceptions have been activated. It is then followed by either one or more *catch* blocks.
2. catch- a program should catch an exception with an exception handler at a place within the program where you need to handle the problem. The *catch* is a keyword that indicates a place where the exception will be caught.
3. finally- the finally block is used for executing a set of statements regardless of whether an exception has been thrown or not. For example, a file must be closed after being opened, whether an exception is thrown or not.
4. throw- an exception is normally thrown after the occurrence of a problem. This is done via the *throw* keyword.

A combination of *try* and *catch* is used for catching exceptions. The try/catch block has to be placed around a code that may raise an exception. Such code is said to be *protected*, and here is the syntax for using these keywords:

```
try {  
    // statements raising the exception  
} catch( ExceptionName exception1 ) {
```

```

// The error handling code
} catch( ExceptionName exception2 ) {
    // The error handling code
} catch( ExceptionName eexceptionN ) {
    // The error handling code
} finally {
    // statements to execute
}

```

One can use many catch statements with the goal of catching many different exceptions if many exceptions are raised by the *try* block.

In C#, exceptions are represented using classes. The *System.Exception* acts as the base class for all exception classes in C# since all other classes are derived from it, either directly or indirectly.

## Handling Exceptions

In C#, exceptions can be handled using the *try* and *catch* blocks. With these blocks, we can separate the core program statements from statements for handling errors. We can handle errors using the *try*, *catch* and *finally* keywords.

A division by zero should, for example, raise an exception as it not mathematically supported. Let us create some code to handle this:

```

using System;
namespace ExceptionHandlingApp {
    class DivisionClass {
        int answer;
        DivisionClass() {
            answer = 0;
        }
        public void division(int x, int y) {
            try {
                answer = x / y;
            } catch (DivideByZeroException exception) {
                Console.WriteLine("Exception caught: {0}", exception);
            } finally {
                Console.WriteLine("Answer: {0}", answer);
            }
        }
    }
}

```

```

        }
    }

    static void Main(string[] args) {
        DivisionClass dc = new DivisionClass();
        dc.division(12, 0);
        Console.ReadKey();
    }
}
}

```

The code should return the exception given below:

```

Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ExceptionHandlingApp.DivisionClass.division (System.Int32 x, System.Int32 y) [0x00000
Answer: 0

```

We created the *division()* function that takes two arguments, x and y, of type integer. We have then passed 12 and 0 to the function, meaning that we will be dividing 12 by 0. However, this has generated an exception since division by zero is not allowed.

## User-Defined Exceptions

C# allows programmers to define their own exceptions. We derive such user-defined exception classes from the *Exception* class. Consider the example given below:

```

using System;
namespace ExceptionHandlingApp {
    class TemperatureTest {
        static void Main(string[] args) {
            TemperatureClass tc = new TemperatureClass();
            try {
                tc.displayTemperature();
            } catch(TempIsZeroException exception) {
                Console.WriteLine("TempIsZeroException: {0}",
exception.Message);
            }
            Console.ReadKey();
        }
    }
}

```

```

        }
    }
}

public class TempIsZeroException: Exception {
    public TempIsZeroException(string msg): base(msg) {
    }
}

public class TemperatureClass {
    int temp = 0;

    public void displayTemperature() {

        if(temp == 0) {
            throw (new TempIsZeroException("We found Zero
Temperature"));
        } else {
            Console.WriteLine("Temperature is: {0}", temp);
        }
    }
}

```

The code will return the following:

**TempIsZeroException: We found Zero Temperature**

### Nested try-catch

C# allows us to create a block of nested try-catch. In such a case, the exception will be caught in the catch block following the try block in which the exception occurred. Consider the following example:

```

using System;
public class NestedtryCatch
{
    public static void Main()
{
    Employee emp = null;

```

```
try
{
try
{
emp.EmployeeName = "";
}
catch
{
Console.WriteLine("The inner catch");
}
}
catch
{
Console.WriteLine("The outer catch");
}
}
```

**public class Employee{**

```
    public string EmployeeName { get; set; }
}
```

The code prints the following:



The inner catch

In case there is no inner catch block with the necessary exception type, the exception will flow to outer catch block until an appropriate exception filter is found. Here is an example:

**using System;**

```
public class NestedtryCatch
{
    public static void Main()
    {
        Employee emp = null;

        try
        {
            try
            {
                // This will throw a NullReferenceException
                emp.EmployeeName = "";
            }
            catch (InvalidOperationException innerException)
            {
                Console.WriteLine("The inner catch");
            }
        }
        catch
        {
            Console.WriteLine("The outer catch");
        }
    }
}
```

```
public class Employee{

    public string EmployeeName { get; set; }
}
```

In the example given above, the statement `emp.EmployeeName` will generate a `NullReferenceException`, but we don't have a catch block that handles a `NullReferenceException` or an `Exception` type. The outer block will handle this. The code returns the following:

# The outer catch

## Chapter 18: File I/O<sup>[136](#)</sup>

A file is characterized by a name and a directory path. Once you open a file, it becomes a *stream*. Files are normally opened for reading or writing purposes.

The stream is simply the sequence of bytes that pass through the communication path. Streams are of two types:

- Input stream
- Output stream

The input stream helps us to read data from a file, that is, a read operation, while the output stream helps us to write data into a file, that is, a write operation.

### **Input/ Output Classes**

C# provides us with a number of classes that we can use for working with files. These classes can be used for accessing directories, files, creating new files, opening existing files and moving files from one directory to another. These classes are defined in the System.IO class. Let us discuss some of these classes:

- **BinaryReader**- this class helps us in reading primitive data from a binary stream.
- **BinaryWriter**- this class helps us in writing primitive data in a binary format.
- **BufferedStream**- this acts as a temporary storage for bytes of streams.
- **Directory**- this class helps us to manipulate the structure of a directory.

- **DriveInfo**- this class provides us with information about the drives.
- **File**- this class is used for manipulation of files.
- **FileInfo**- helps in performing operations on files.
- **FileStream**- used for writing and reading from any location in a file.
- **MemoryStream**- helps in randomly accessing data kept in a memory.
- **Path**- for performing operations on path information.
- **StreamReader**- reads characters from a stream of bytes.
- **StreamWriter**- writes characters to a stream.
- **StringReader**- reads from a string buffer.
- **StringWriter**- writes into a string buffer.

### **FileStream Class**

This class is defined in the `System.IO` namespace and it helps us to read from and write to files. We can also use it to close files<sup>[137](#)</sup>.

For you to be able to use this class, you have to create its object. This instance can then be used for creating new files and opening existing files. Here is how you can create an instance of the `FileStream` class:

```
FileStream <object> = new FileStream(<file>, < FileMode Enumerator>,
< FileAccess Enumerator>, < FileShare Enumerator>);
```

For example, suppose we need to read a file named `names.txt`. We can create a `FileStream` object named `FS` and use it for this purpose. This is demonstrated below:

```
FileStream FS = new FileStream("names.txt",  FileMode.Open,  
 FileAccess.Read, FileShare.Read);
```

The FileMode is an enumerator that defines a number of methods that can be used for opening files. These methods include the following:

- Append – this method opens an existing file then puts the cursor at the end of the file, or it creates a new file if the specified file doesn't exist.
- Create – for creating a new file.
- CreateNew – It instructs the operating system to create a new file.
- Open – for opening an existing file.
- OpenOrCreate – It instructs the operating system to open a file if it is in existence or create a new file if it doesn't exist.
- Truncate – for opening an existing file and truncating its size to zero bytes.

The FileAccess is an enumerator that comes with a number of methods including Read, Write and ReadWrite.

The FileShare enumerator comes with the following members:

- Inheritable – helps a file handle in passing inheritance to child processes.
- None – It disables sharing of the current file.
- Read – It opens a file for reading.
- ReadWrite – opens a file for reading and writing.
- Write – opens a file for writing.

The FileStream class can be used as demonstrated below:

```

using System.IO;
using System;
namespace FileApp {
    class MyProgram {
        static void Main(string[] args) {
            FileStream FS = new FileStream("names.dat",
FileMode.OpenOrCreate,
            FileAccess.ReadWrite);
            for (int x = 1; x <= 10; x++) {
                FS.WriteByte((byte)x);
            }
            FS.Position = 0;
            for (int x = 0; x <= 10; x++) {
                Console.Write(FS.ReadByte() + " ");
            }
            FS.Close();
            Console.ReadKey();
        }
    }
}

```

The code will return the following output after execution:

1 2 3 4 5 6 7 8 9 10 -1

## Appending Text Lines<sup>[138](#)</sup>

Sometimes, you may need to append a number of text lines to a file. This can be done by calling the *AppendAllLines()* method. Here is an example:

```

string multipleLines = "The first line." + Environment.NewLine +
    "The second line." + Environment.NewLine +
    "The third line.";

```

**// To open the file named Myfile.txt then append the above lines. If the file does not exist, a new one will be created.**

```
File.AppendAllLines(@"C:\MyFile.txt",
multipleLines.Split(Environment.NewLine.ToCharArray()).ToList<string>());
```

The file will be opened and the specified lines will be appended to the file.

### Appending a String

The *File.AppendAllText()* method can allow you to append a string to a file in only a single line of code. This is demonstrated below:

```
//Open the file named MyFile.txt then append text to it. If the file does
not exist, create a new one and open it for writing.
```

```
File.AppendAllText(@"C:\ MyFile.txt", "This string will be written into
the file");
```

### Overwriting Text

If you need to overwrite a file, use the *File.WriteAllText()* method. The method will delete the text in the file and replace it with the one that you specify. This method can be used as demonstrated below:

```
// To open the file MyFile.txt and write the text into it. If the file does not
exist, a new one will be created and opened for writing.
```

```
File.WriteAllText(@"C:\MyFile.txt", "This text will be used for
replacing the current text in the file.");
```

With the Static File Class, one can perform various operations. This is demonstrated below:

```
//Check whether the file exists at the specified location or not
```

```
bool isFileExists = File.Exists(@"C:\ MyFile.txt"); // it will return false
```

```
//Copy MyFile.txt as the new file MyNewFile.txt
```

```
File.Copy(@"C:\MyFile.txt", @"D:\MyNewFile.txt");
```

```
// Check when the file was lastly accessed
```

```
DateTime lastAccessTime = 
File.GetLastAccessTime(@"C:\MyFile.txt");
```

```
//get the last time the file was written
```

```
DateTime lastWriteTime = File.GetLastWriteTime(@"C:\MyFile.txt");
```

```

// Transfer the file to a new location
File.Move(@"C:\MyFile.txt", @"D:\MyFile.txt");

//Open file and returns FileStream for reading bytes from the file
FileStream fs = File.Open(@"D:\MyFile.txt",
 FileMode.OpenOrCreate);

//Open the file and return a StreamReader for reading a string from the
file
StreamReader sr = File.OpenText(@"D:\MyFile.txt");

```

#### **//Delete the file**

```
File.Delete(@"C:\MyFile.txt");
```

The above shows that with the Static File class, it becomes easy for us to work with physical files. However, for more flexibility, one can use the *FileInfo* class.

### **StreamReader Class**

This class helps us to read data from a text file. This class inherits the Stream base class. It also inherits the TextReader class, an abstract base class for reading a series of characters. The following are the popular methods provided by this class:

- public override void Close()- the method closes the StreamReader object and the stream, then any system resources that were being used by the reader are released.
- public override int Peek()- this method will return the next character that is available without consuming it.
- public override int Read()- this method will read the next character in an input stream then advance the position of the character by 1.

Let us demonstrate how we can use this class to read from a file named *names.txt*:

```
using System.IO;
```

```

using System;

namespace IOApplication {
    class MyProgram {
        static void Main(string[] args) {
            try {
                // Create a StreamReader instance for reading from a file.
                // The using statement will close the StreamReader.
                using (StreamReader sr = new StreamReader("c:/names.txt")) {
                    string line;

                    // Read the show lines from the file until
                    // you reach the end of the file.
                    while ((line = sr.ReadLine()) != null) {
                        Console.WriteLine(line);
                    }
                }
            } catch (Exception exception) {
                // Information the user about what went wrong
                Console.WriteLine("File couldn't be read:");
                Console.WriteLine(exception.Message);
            }
            Console.ReadKey();
        }
    }
}

```

The code will read the text written in the file *names.txt* and print it.

## **StreamWriter Class**<sup>[139](#)</sup>

This class inherits the `TextWriter` abstract class that represents a write, capable of writing a series of characters. Here are the popular methods for this class:

- public override void `Close()`- this method will close the current object of the `StreamWriter` as well as the underlying stream.
- public override void `Flush()`- this method will clear buffers for current writer and makes any buffered data to be written to the

stream.

- public virtual void Write(bool value)- this method writes a textual representation of a Boolean value into the text stream or string.
- public override void Write(char value)- for writing a character to a stream.
- public virtual void Write(decimal value)- this method writes a textual representation of a decimal value to a text stream or string.
- public virtual void Write(double value)- writes a text representation of 8-byte floating point value into a text stream or string.
- public virtual void Write(int value)- writes a text representation of 4-byte signed integer into a text stream or string.
- public override void Write(string value)- for writing a string to a stream.
- public virtual void WriteLine()- for writing a line terminator to a text stream or string.

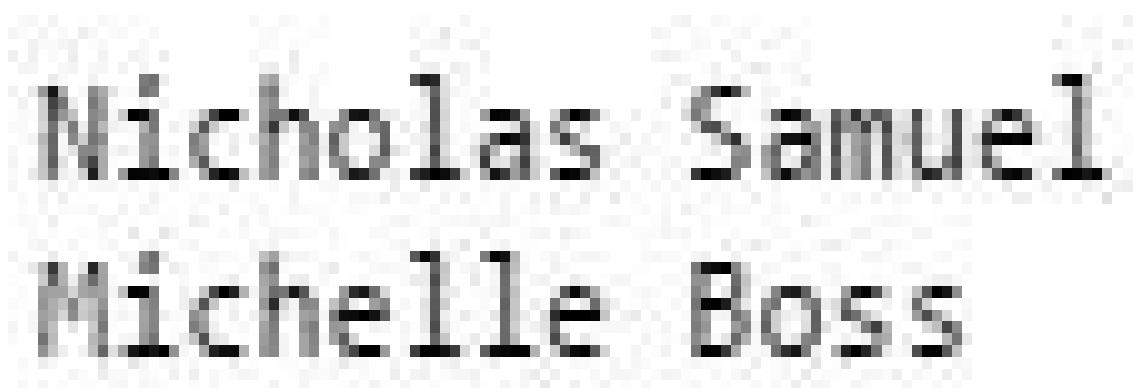
Let us demonstrate how we can use the StreamWriter class to write text data:

```
using System.IO;
using System;
namespace IOApp {
    class MyProgram {
        static void Main(string[] args) {
            string[] students = new string[] {"Nicholas Samuel", "Michelle
Boss"};
            using (StreamWriter sw = new StreamWriter("students.txt")) {

                foreach (string x in students) {
                    sw.WriteLine(x);
                }
            }
        }
    }
}
```

```
// Read and display every line from the file.  
string text = "";  
using (StreamReader sr = new StreamReader("students.txt")) {  
    while ((text = sr.ReadLine()) != null) {  
        Console.WriteLine(text);  
    }  
}  
Console.ReadKey();  
}  
}
```

The code will print the following result after execution:



# BinaryReader Class [140](#)

This class can be used for reading binary data from a file. We should first create a `BinaryReader` object by passing an object of `FileStream` to its constructor. This class comes with a number of methods including the following:

- public override void Close()- for closing the BinaryReader object as well as the underlying stream.
  - public virtual int Read()- for reading characters from the underlying stream and advancing the stream's current position.

- public virtual bool ReadBoolean()- for reading a Boolean value from the current stream and advancing the stream's current position by a byte.
- public virtual byte ReadByte()- for reading the next byte from current stream into byte array and advancing current position by a similar number of bytes.

## **BinaryWriter Class**<sup>[141](#)</sup>

This class helps in writing binary data into a stream. To create a BinaryWriter object, we pass a FileStream object to the constructor. This class comes with a number of methods including the following:

- public override void Close()- for closing the BinaryWriter object as well as the underlying stream.
- public virtual void Flush()-this method will clear buffers for current writer and makes any buffered data to be written to the device.
- public virtual void Write(bool value)- for writing a one-byte Boolean value into the current stream. 1 represents true while 0 represents false.

Let us create an example that demonstrates how to read and write binary data:

```
using System;
using System.IO;
namespace BinaryFileApplication {
    class MyProgram {
        static void Main(string[] args) {
            BinaryWriter bw;
            BinaryReader br;

            int x = 12;
            double db = 1.24658;
            bool bl = false;
```

```
string st = "Hello world";

//create a file
try {
    bw = new BinaryWriter(new FileStream("testdata",
 FileMode.Create));
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to create the
file.");
    return;
}
//write to the file
try {
    bw.Write(x);
    bw.Write(db);
    bw.Write(bl);
    bw.Write(st);
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to write to
the file.");
    return;
}
bw.Close();

//read from the file
try {
    br = new BinaryReader(new FileStream("testdata",
 FileMode.Open));
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to open the
file.");
    return;
}
try {
    x = br.ReadInt32();
    Console.WriteLine("Integer data: {0}", x);
    db = br.ReadDouble();
```

```
        Console.WriteLine("Double data: {0}", db);
        bl = br.ReadBoolean();
        Console.WriteLine("Boolean data: {0}", bl);
        st = br.ReadString();
        Console.WriteLine("String data: {0}", st);
    } catch (IOException exception) {
        Console.WriteLine(exception.Message + "\n Unable to read from
the file.");
        return;
    }
    br.Close();
    Console.ReadKey();
}
}
```

The code will give the following result after execution:

```
Integer data: 12
Double data: 1.24658
Boolean data: False
String data: Hello world
```

# Chapter 19: Delegates

We can have a function with more than one parameters from different data types. However, we may sometimes need to pass a certain function as a parameter. How can C# handle the event handler or callback functions? This is done via *delegates*<sup>[142](#)</sup>.

A delegate is taken as a pointer to a function. It is a reference data type that holds reference of a method. All delegates are derived from *System.Delegate* class implicitly. We use delegates to implement events and callback methods<sup>[143](#)</sup>.

## Declaring a Delegate

The way we declare a delegate determine the methods that the delegate can reference. A delegate may refer to a method, which has the same signature as the delegate<sup>[144](#)</sup>. We use the *delegate* keyword to declare a delegate. The following syntax is used for the declaration:

```
<access modifier> delegate <return type><delegate_name>
(<parameters>)
```

Here is an example:

```
public delegate void Display(int value);
```

Above, we have declared a *Display* delegate. We can use this delegate to point to a method with same return type and parameters that have been declared with the *Display* delegate.

Consider the example given below:

```
using System;
public class MyProgram
{
    public delegate void Display(int num);
```

```
public static void Main()
{
    // Display delegate points to the DisplayNumber
```

```

Display displayDel = DisplayNumber;

displayDel(100000);
displayDel(200);

// Display delegate points to DisplayMoney
displayDel = DisplayMoney;

displayDel(50000);
displayDel(20);
}

```

```

public static void DisplayNumber(int x)
{
    Console.WriteLine("Number: {0,-12:N0}",x);
}
public static void DisplayMoney(int amount)
{
    Console.WriteLine("Money: {0:C}", amount);
}
}

```

The code will print out the following:

```

Number: 100,000
Number: 200
Money: $50,000.00
Money: $20.00

```

We have created a delegate named *Display* that accepts a parameter of type integer and returns void. With the Main() method, we have created a variable

of type Display and a method named *DisplayNumber* has been assigned to it. After invoking the Display delegate, the *DisplayNumber* method will be called also, if the Display delegate variable is assigned to *DisplayMoney* method, the *DisplayMoney* method will be invoked.

Also, it is possible for us to create a delegate object using the *new* keyword. In this case, we have to specify the name of the method as shown below:

**Display displayDel = new Display (DisplayNumber);**

### **Invoking a Delegate**

A delegate is a reference to a method; hence we can invoke it simply like a method. When a delegate is invoked, the method that is referred to will, in turn, be invoked.

There are two ways through which we can invoke a delegate: using the () operator or using *Invoke()* method of delegate. Let us demonstrate this using an example:

**Display displayDel = DisplayNumber;**

**displayDel.Invoke(5000);**

**//or**

**printDel(10000);**

The two ways that we can use to invoke a delegate have been shown above.

### **Passing a Delegate as a Parameter**

A method may have a parameter of delegate type and it can invoke the delegate parameter. Here is an example of a delegate parameter:

```
public static void DisplayHelper(Display delegateFunc, int numToDisplay)
{
    delegateFunc(numToDisplay);
}
```

In the example given above, the *DisplayHelper* function has a delegate parameter of *Display* type and it has been invoked as a function using the statement given below:

**delegateFunc(numToDisplay);**

Let us give another example demonstrating how to use the *DisplayHelper* method and a delegate type parameter:

```
using System;
```

```
public class MyProgram
{
    public delegate void Display(int num);

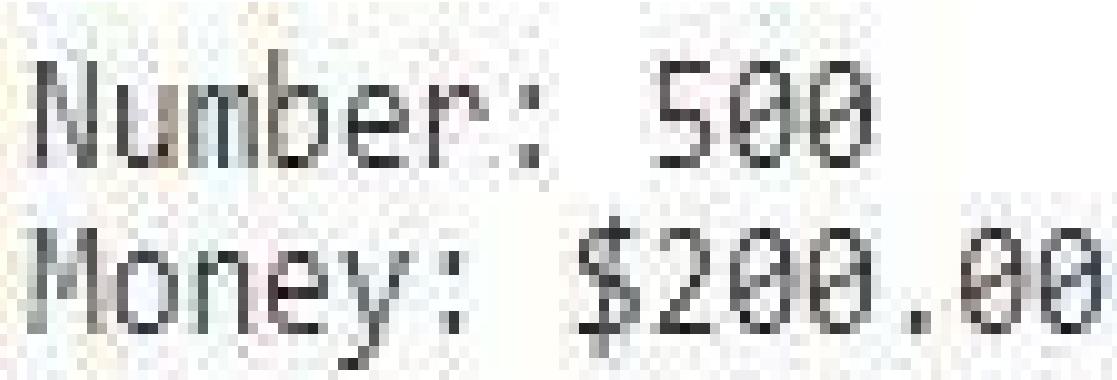
    public static void Main()
    {
        DisplayHelper(DisplayNumber, 500);
        DisplayHelper(DisplayMoney, 200);
    }

    public static void DisplayHelper(Display delegateFunc, int
numToDisplay)
    {
        delegateFunc(numToDisplay);
    }

    public static void DisplayNumber(int val)
    {
        Console.WriteLine("Number: {0,-12:N0}",val);
    }

    public static void DisplayMoney(int amount)
    {
        Console.WriteLine("Money: {0:C}", amount);
    }
}
```

The code will return the following output:



Number: 500  
Money: \$200.00

### Multicast Delegate

It is possible for a delegate to point to multiple functions. Any delegate that points to many functions is referred to as a *multicast delegate*. Use the + operator to add a function to a delegate object and the – operator to remove an existing function from a delegate object.

Consider the example given below:

```
using System;
public class MyProgram
{
    public delegate void Display(int value);

    public static void Main()
    {
        Display displayDel = DisplayNumber;
        displayDel += DisplayHexadecimal;
        displayDel += DisplayMoney;

        displayDel(1000);
        displayDel -= DisplayHexadecimal;
        displayDel(2000);
    }

    public static void DisplayNumber(int x)
    {
```

```
Console.WriteLine("Number: {0,-12:N0}",x);
}

public static void DisplayMoney(int amount)
{
Console.WriteLine("Money: {0:C}", amount);
}
public static void DisplayHexadecimal(int dec)
{
Console.WriteLine("Hexadecimal: {0:X}", dec);
}
}

The code will print the following as the result:
```

```
Number: 1,000
Money: $1,000.00
Number: 2,000
Money: $2,000.00
```

In the above example, we have the `Display` pointing to three methods namely `DisplayNumber`, `DisplayMoney` and `DisplayHexadecimal`. This makes it a multicast delegate. This means that when the `displayDel` is invoked, it will, in turn, invoke all the three methods sequentially.



# Chapter 20: Multithreading in C#

A thread defines a control flow. Thread is a basic unit that the operating system assigns a thread. The execution of a thread is independent within a program.

A single process is executed using one thread. Such process is known as single – threaded process. Only one task can be performed at a time. The user has to wait for the task to complete before executing new task.

For executing more than one thread at a time, multiple threads are created. The process creating two or more threads is known as multithreading<sup>145</sup>.

## Life cycle of a thread

The life cycle begins when the object of **System.Threading.Thread** class is created. The life ends as soon as the task is completed. There are various states in the life cycle of a thread.

- **Unstarted State:** When the instance of the **Thread** class is created, the thread enters in unstarted state.
- **Ready State:** The thread is in this state until the program calls the **Start()** method.
- **Not Runnable State:** Some of the elements that make a thread not to be in a runnable state include:
  1. **Waiting:** The **Wait()** method is called to make the thread for a specified condition
  2. **Blocked:** The thread is blocked by an I/O operation
  3. **Sleeping:** The **Sleep()** method is called to put the thread in sleeping mode.
- **Dead State:** Once the thread completes its execution or aborted, it is placed in a dead state

## Main thread

When working with threads, we make use of the **System.Threading.Thread** class. The main thread is created as soon as the program starts execution. The **Thread** class is used for creating threads.

They are known as child threads. The user can access the main thread by using the **CurrentThread** property of the Thread class.

**Example:**

```
using System;
namespace thread
{
    class MainThread
    {
        static void Main(string[] args)
        {
            Thread t1 = new Thread();
            t1.Name="Thread1";
            Console.WriteLine("Thread is:{0}",t1.Name);
            Console.Read();
        }
    }
}
```

**On compiling and executing the code, the output is:**

Thread is: Thread1

## Properties and methods of the Thread class

### Properties <sup>[146](#)</sup>:

- **IsAlive:** The value showing the execution status of the current thread
- **CurrentThread:** The current running thread is retrieved
- **CurrentContext:** The current context in which the thread is executing is retrieved
- **ExecutionContext:** The ExecutionContext object contains information about different contexts
- **Name:** Gets or sets the name of the thread
- **ThreadState:** The value containing states of the current thread

- **Priority:** It gets or sets the value showing the scheduling priority of a thread

### Methods:

- **public static void BeginThreadAffinity():** The host is to about to execute instructions depending on the current physical operating system thread.
- **public void Abort():** The ThreadAbortException is raised in the thread on which it is invoked.
- **public void interrupt():** The thread present in the WaitSleepJoin state is interrupted
- **public static AppDomain GetDomain():** A unique domain identifier is returned
- **public static void MemoryBarrier():** The processor executes the current thread. The instructions cannot be reordered.
- **public void Start():** It starts the thread
- **public static bool Yield():** The calling thread to yield execution to another thread which is ready to run on the processor

### Creating and managing threads<sup>[147](#)</sup>

The extended thread class creates a thread. The extended thread class calls the **Start()** method to start the child thread execution.

### **Example:**

```
using System;
using System.Threading;

namespace MultipleThread
{
    class ThreadProgram
    {
        public static void CallChild()
        {
            Console.WriteLine("Start child thread");
        }

        static void Main(string[] args)
        {
            ThreadStart child1 = new ThreadStart(CallChild);
            Console.WriteLine("Creating child thread");
            Thread child2 = new Thread(child1);
            child2.Start();
            Console.Read();
        }
    }
}
```

**On compiling and executing the code, the output is:**

Start child thread

Creating child thread

### **Managing Threads**

When there is a need to pause a thread for a period of time so that another thread can execute, the **Thread.Sleep()** method is used. The method takes a single argument stating time in milliseconds.

### **Example:**

```
using System;
using System.Threading;
```

```
namespace Multithreaded
{
    class Program
    {
        public static void ChildThread()
        {
            Console.WriteLine("Start child thread");
            int sleeptime = 4000;

            Console.WriteLine("Thread sleeping for {0} seconds",sleeptime / 1000);
            Thread.Sleep(sleeptime);
            Console.WriteLine("Resume child thread");

        }
        public static void Main()
        {
            ThreadStart t1 = new ThreadStart(ChildThread);
            Console.WriteLine("child thread created");
            Thread child1 = new Thread(t1);
            child1.Start();
            Console.Read();
        }
    }
}
```

**On compiling and executing the code, the output is:**

Start child thread

Thread sleeping for 4 seconds

Resume child thread

child thread created

## Destroying threads

The **Thread.Abort()** method is used to destroy the thread. The **ThreadAbortException** is thrown when the thread is destroyed. The exception is not caught and is sent to the **finally** block.

### Example:

```
using System;
using System.Threading;

namespace ThreadDemo
{
    class Program
    {
        public static void ChildThread()
        {
            try
            {
                Console.WriteLine("Child Thread started");
                for(int j = 0; j <= 10; j++)
                {
                    Thread.Sleep(1000);
                    Console.WriteLine("Child thread finished");
                }
            }
            catch(ThreadAbortException e)
            {
                Console.WriteLine("Exception caught");
            }
            finally
            {
                Console.WriteLine("Exception is not handled");
            }
        }

        public static void Main()
    }
}
```

```
{  
    ThreadStart t1 = new ThreadStart(ChildThread);  
    Console.WriteLine("Creating child thread");  
    Thread t2 = new Thread(t1);  
    t1.Start();  
  
    //main thread is stopped  
    Thread.Sleep(2000);  
  
    //child thread aborted  
    Console.WriteLine("Aborting child thread");  
    t2.Abort();  
    Console.Read();  
}  
}  
}
```

c, the output is:

Creating child thread

Child Thread started

0

1

Aborting child thread

Exception caught

Exception is not handled

# Chapter 21: Event

An event is something that is expected to happen. In C#, events are user actions like click, key press, mouse movements or occurrences such as system-generated notifications<sup>[148](#)</sup>. An application should respond to events once they occur. A good example is the occurrence of an interrupt. Events are used to facilitate inter-process communications.

Events are declared in a class and associated with an event handler via delegates in the same class or in another class. The class with the event is used for publishing the event. Such is known as the *publisher class*. The class accepting the event is known as the *subscriber class*. This means that events rely on a publisher-subscriber model.

The publisher is the object with the definition of both the event and the delegate. The association between the event and the delegate is defined in this object. An object of the publisher class invokes the event which is in turn notified to the other objects.

The subscriber is the object that accepts the event and offers the event handler. The publisher class has a delegate that invokes the method (event handler) of the subscriber class.

## Event Declaration

For an event to be declared inside a class, you should first declare a delegate type for the event. It is after this that you can declare the event using the *event* keyword<sup>[149](#)</sup>. Here is an example:

```
public delegate void myEvent();
public event myEvent myEvent;
```

Thus, we have used the *event* keyword to make it an event. Here is a complete example:

```
using System;
public class DisplayHelper
{
    // declare the delegate
```

```
public delegate void BeforeDisplay();

//declare an event of type delegate
public event BeforeDisplay beforeDisplayEvent;

public DisplayHelper()
{
}

public void DisplayNumber(int x)
{
    //call the delegate method before moving to display
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Number: {0,-12:N0}", x);
}

public void PrintDecimal(int dec)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Decimal: {0:G}", dec);
}

public void DisplayMoney(int amount)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Money: {0:C}", amount);
}

public void DisplayTemperature(int x)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();
```

```

        Console.WriteLine("Temperature: {0,4:N1} F", x);
    }
    public void DisplayHexadecimal(int dec)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Hexadecimal: {0:X}", dec);
    }
}

```

The *DisplayHelper* is a publisher class responsible for publishing the *beforeDisplay* event. After every *display* method, it first checks to determine whether the *beforeDisplayEvent* is not null then it calls the *beforeDisplayEvent()* method.

We now need to create a subscriber. Consider the class given below:

```
using System;
```

```

public class MyProgram
{
    public static void Main()
    {
        NumberClass nc = new NumberClass(200);
        nc.DisplayMoney();
        nc.DisplayNumber();
    }
}

class NumberClass
{
    private DisplayHelper _displayHelper;

    public NumberClass(int x)

```

```
{  
    _value = x;  
  
    _displayHelper = new DisplayHelper();  
    //subscribe to the beforeDisplayEvent event  
    _displayHelper.beforeDisplayEvent +=  
displayHelper_beforeDisplayEvent;  
}  
//beforeDisplayevent handler  
void displayHelper_beforeDisplayEvent()  
{  
    Console.WriteLine("BeforeDisplayEventHandler: DisplayHelper  
will print a value");  
}  
  
private int _value;  
  
public int Value  
{  
    get { return _value; }  
    set { _value = value; }  
}  
  
public void DisplayMoney()  
{  
    _displayHelper.DisplayMoney(_value);  
}  
  
public void DisplayNumber()  
{  
    _displayHelper.DisplayNumber(_value);  
}  
}  
  
public class DisplayHelper  
{  
    // declare a delegate
```

```
public delegate void BeforeDisplay();

//declare an event of type delegate
public event BeforeDisplay beforeDisplayEvent;

public DisplayHelper()
{
}

public void DisplayNumber(int y)
{
    //call a delegate method before printing
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Number: {0,-12:N0}", y);
}

public void DisplayDecimal(int dec)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Decimal: {0:G}", dec);
}

public void DisplayMoney(int amount)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Money: {0:C}", amount);
}

public void DisplayTemperature(int y)
{
```

```
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Temperature: {0,4:N1} F", y);
    }

    public void DisplayHexadecimal(int dc)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Hexadecimal: {0:X}", dc);
    }
}
```

The code returns the following result:

```
BeforeDisplayEventHandler: DisplayHelper will print a value
Money: $200.00
BeforeDisplayEventHandler: DisplayHelper will print a value
Number: 200
```

All subscribers must have a handler function which will be called after the publisher has raised an event.

## Chapter 22: Hints and Important Resources

Although I believe that if you follow the steps of this book you will have a very thorough and firm grasp on C# in a short period, there are undoubtedly still questions you may have that haven't been addressed.

The wonderful thing about the tech-savvy world that we live in today is that this book is not your only resource to learning and mastering the programming and coding of C#.

As I believe you now feel ready and able to take on many challenges you wish to address with your coding in the C# format, you may come to a crossroads where you are unsure of what the next potential step in problem-solving is. This chapter should serve as your guide to finding any answers to questions that you have not received within the pages of this book.

First and foremost, I encourage you to look through your code for simple errors in spacing or capitalization.

Sometimes if you are used to coding in a different language, you may have accidentally put a period where a semicolon should be in C# or even put brackets where they should not be.

These are the types of errors that I find to be the most frustrating and typically are the mistakes that I have made. Essentially, always double check your work when you get to a frustrating point that seems to keep giving you an error message.

Also, if you try this tactic and it doesn't work and you are ready to give up —take a break! Come back to your project after you have let yourself relax and take your mind off your project. When you come back and look at it with refreshed eyes you may find the error of your ways.

Another resource that I utilize as well are online forums for C# in particular. Although some programming forums offer great general advice, I encourage you to Google things like "C# help forum" in order to find others with potentially sage advice on how to solve an issue.

There is a likelihood that you may not be the only individual to ever come across a certain speed hump that may be preventing you from completing a portion of your project. These can be great and accessible ways to quickly find an answer by using the power of the internet.

Although I already mentioned these next resources, I want to reiterate just how important code editors and libraries can potentially be.

Just as within any programming language, you can find seemingly endless answers in the libraries that will allow you to code for many different situations. As I suggested before, however, please have a firm grasp on the process of coding and programming before you venture into the world of using code.

If something changes within the code and you can't find the mistake, you could potentially have irreparable damage to your program or project. Code editors can help you find these mistakes before making your code live, so I highly suggest that you consistently utilize this resource until you are a C# master—maybe even then just in case you fall victim to human error.

If you find that you need a better visual for any of the content within this book, or outside of the scope—I encourage you to visit my favorite video site—YouTube!

Just search for a key phrase or word that you are attempting to figure out during your coding and programming and I guarantee that you will find a helpful video. Sometimes, it may not be the very first video that you come across but it certainly will be there.

You may already have a preferred YouTuber that you go to for examples in a different language—maybe go look at the other users he or she subscribes to and see if their content can be helpful to you.

Never think that just because you are struggling it is the end of your project as a whole.

I can tell you from personal experience that allowing yourself to fully explore and problem-solve when issues like this come up gives you an

opportunity to create new pathways in your brain that will serve you well with your struggles to come.

Think back on the ways that you have solved other issues in your life, or programming experiences, and allow yourself to strategize how and why you can apply it to the situation at hand.

You are your own best ally when it comes to knowing how you think and what you want to accomplish. Use your resources to the best of your ability and become as knowledgeable as you can—the power of knowledge and learning is quite a remarkable skill.

Programming and coding can be seen as a problem-solving matter that you use to create programs that perform a specific purpose that can make things easier for you and the lives of others. Always work towards an achievable goal for your skill set, while continuing to push yourself to learn more and get to the next level of your functioning.

You can get more detailed information about the C# language from the following reference links.

- [Visual Studio Application](#) – The IDE for creating C# applications.
- [C# \( Programming\\_guide \)](#) – An overview of C# programming language
- [C# Programming](#) – The information about the C# features using .NET framework is explained
- [Mono](#) – Cross-platform applications can be easily created using the software.

# Conclusion

Now that this book has come to a close, you have all the information and knowledge of skills in order to be an excellent C# programmer and coder. It is important to me that you use this book as the foundation for your skills. I have worked hard to provide what I think is a guide that is comprehensive and easy to understand. Without languages like C#, we could not have the amazing technology both in our personal and business lives.

When you think about the opportunities that technological skills such as programming create in your life, I believe it is an invaluable skill. Knowing about the very basics of the computer system setup is something that can open doors for you in a professional sense, with more complex and high paying jobs.

However, in your personal life, you can use the critical thinking and problem-solving skills to help enhance your own gaming experience on your computer or other fields outside of technology. With the quickness you learned C#, you should feel very confident and fulfilled in your accomplishment!

Not everyone can learn an entirely new language in a month's time, which you have just simply and easily done through exploring a new virtual world. It is an accomplishment that you should be happy about, and really start to create the projects that you desire. As an object-oriented language, I am sure you have some excellent new ideas for contributions within the software community. Or perhaps you just want to be able to manipulate the Windows computer you have!

The descriptions and sections that I have broken down in this book should have allowed you to grasp and connect different pieces and parts of C# in a way that, when you reached this final conclusion, you are able to feel as though you can pass along knowledge to someone else.

I hope this language showed you that it encourages gameplay programming, through the many different avenues used in this language to define and

separate different aspects that may touch (classes and statements for example) but are not interchangeable.

It may appear to be an overwhelming confusion as you get used to the nuances—the more you test out your skills and put examples of code to the test, the more you will come to realize the full impact and advantages these offer you.

Thank you for picking up my book and giving it a chance to teach you something new! I would like to make the basics of every topic something that any person can learn quickly and easily. I hope you use this as a reference guide for your future projects and that you have been able to absorb the material and concepts with the help of the examples within the book. I want to end the book with a few more pearls of wisdom I find important to reiterate from my own experiences.

Always test your projects in a test zone before putting them to life, especially if irrevocable damage could be done! Remember that you should be enjoying yourself when you are programming and coding because you are creating something complex by putting the pieces together yourself. Lastly, have fun and enjoy yourself while doing it!

# Bibliography

AG, A. (n.d.). *Delegates and Events in C# / .NET*. [online] Akadia.com. Available at: [https://www.akadia.com/services/dotnet\\_delegates\\_and\\_events.html](https://www.akadia.com/services/dotnet_delegates_and_events.html) [Accessed 15 Feb. 2019].

Bodnar, J. (2019). *Methods in C# - working with CSharp methods*. [online] Zetcode.com. Available at: <http://zetcode.com/lang/csharp/methods/> [Accessed 15 Feb. 2019].

Bodnar, J. (2019). *Operators in C# - describing CSharp operators and expressions*. [online] Zetcode.com. Available at: <http://zetcode.com/lang/csharp/operators/> [Accessed 15 Feb. 2019].

Codescracker.com. (n.d.). *C# Program Structure*. [online] Available at: <https://codescracker.com/c-sharp/c-sharp-program-structure.htm> [Accessed 15 Feb. 2019].

Csharp.net-informations.com. (n.d.). *C# File Operations Tutorial*. [online] Available at: <http://csharp.net-informations.com/file/csharp-file-tutorial.htm> [Accessed 15 Feb. 2019].

Csharp.net-tutorials.com. (n.d.). *Variables - The complete C# tutorial*. [online] Available at: <https://csharp.net-tutorials.com/basics/variables/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Arrays - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Casting and type conversions - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Delegates - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *goto statement - C# Reference*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Passing Parameters - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Classes - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Default values table - C# Reference*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Using threads and threading*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *BinaryReader Class (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *BinaryWriter Class (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *File.AppendText(String) Method (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.file.appendtext?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

us/dotnet/api/system.io.file.appendtext?view=netframework-4.7.2 [Accessed 15 Feb. 2019].

Download.microsoft.com. (n.d.). *Microsoft Download Center: Windows, Office, Xbox & More.* [online] Available at: <http://download.microsoft.com/> [Accessed 15 Feb. 2019].

En.wikipedia.org. (2019). *C Sharp (programming language)*. [online] Available at: [https://en.wikipedia.org/wiki/C\\_Sharp\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)) [Accessed 15 Feb. 2019].

GeeksforGeeks. (n.d.). *C# | Arrays - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-arrays/> [Accessed 15 Feb. 2019].

GeeksforGeeks. (n.d.). *C# | Methods - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-methods/> [Accessed 15 Feb. 2019].

Guru99.com. (n.d.). [online] Available at: <https://www.guru99.com/c-sharp-data-types.html> [Accessed 15 Feb. 2019].

Heydrick, C. (2014). *C# Struct Sizes*. [online] Chris Heydrick: Serial Hobbyist. Available at: <https://chrisheydrick.com/2014/12/17/c-struct-sizes/> [Accessed 15 Feb. 2019].

Jonskeet.uk. (n.d.). *Parameter passing in C#*. [online] Available at: <http://jonskeet.uk/csharp/parameters.html> [Accessed 15 Feb. 2019].

Khorshidnia, S. (2013). *Recursive methods using C#*. [online] Codeproject.com. Available at: <https://www.codeproject.com/Articles/142292/Recursive-methods-in-Csharp> [Accessed 15 Feb. 2019].

Mayo, J. (n.d.). *Lesson 5: Methods - C# Station*. [online] C# Station. Available at: <https://csharp-station.com/Tutorial/CSharp/Lesson05> [Accessed 15 Feb. 2019].

Mkhitaryan, A. (2017). *Why Is C# Among The Most Popular Programming Languages in The World?*. [online] Medium. Available at: <https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb> [Accessed 15 Feb. 2019].

Mono-project.com. (2019). *Home | Mono*. [online] Available at: <http://www.mono-project.com/> [Accessed 15 Feb. 2019].

Msdn.microsoft.com. (2017). *C# Programming Guide*. [online] Available at: <https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx> [Accessed 15 Feb. 2019].

o7planning.org. (n.d.). *C# Multithreading Programming Tutorial*. [online] Available at: <https://o7planning.org/en/10553/csharp-multithreading-programming-tutorial> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# foreach loop (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/foreach-loop> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# if, if...else, if...else if and Nested if Statement (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/if-else-statement> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# Operators: Arithmetic, Comparison, Logical and more..* [online] Available at: <https://www.programiz.com/csharp-programming/operators> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# switch Statement (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/switch-statement> [Accessed 15 Feb. 2019].

Spasojevic, M. (2018). *C# Basics - C# Type Conversions (Implicit and Explicit Conversion)*. [online] Code Maze. Available at: <https://code-maze.com/csharp-basics-type-conversion/> [Accessed 15 Feb. 2019].

Tutorialspoint.com. (n.d.). *Online Csharp Compiler - Online Csharp Editor - Online Csharp IDE - Csharp Coding Online - Practice Csharp Online - Execute Csharp Online - Compile Csharp Online - Run Csharp Online*. [online] Available at: [https://www.tutorialspoint.com/compile\\_csharp\\_online.php](https://www.tutorialspoint.com/compile_csharp_online.php) [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# Files & Directories*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-file> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# for loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-for-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# keywords*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-keywords> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# while loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-while-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Do-While loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-do-while-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Event in C#*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-event> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Struct in C#*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-struct> [Accessed 15 Feb. 2019].

Visual Studio. (n.d.). *Downloads | IDE, Code, & Team Foundation Server | Visual Studio*. [online] Available at:

<https://www.visualstudio.com/downloads/download-visual-studio-vs> [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Arrays*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_arrays.htm](https://www.tutorialspoint.com/csharp/csharp_arrays.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Classes*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_classes.htm](https://www.tutorialspoint.com/csharp/csharp_classes.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Delegates*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_delegates.htm](https://www.tutorialspoint.com/csharp/csharp_delegates.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Encapsulation*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_encapsulation.htm](https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Events*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_events.htm](https://www.tutorialspoint.com/csharp/csharp_events.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Exception Handling*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_exception\\_handling.htm](https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# File I/O*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_file\\_io.htm](https://www.tutorialspoint.com/csharp/csharp_file_io.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Inheritance*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_inheritance.htm](https://www.tutorialspoint.com/csharp/csharp_inheritance.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Multithreading*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_multithreading.htm](https://www.tutorialspoint.com/csharp/csharp_multithreading.htm)

[Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Polymorphism*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_polymorphism.htm](https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Regular Expressions*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_regular\\_expressions.htm](https://www.tutorialspoint.com/csharp/csharp_regular_expressions.htm) [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Structures*. [online] Available at: [https://www.tutorialspoint.com/csharp/csharp\\_struct.htm](https://www.tutorialspoint.com/csharp/csharp_struct.htm) [Accessed 15 Feb. 2019].

# JAVASCRIPT

*The Ultimate Beginner's Guide  
to Learn JavaScript Programming step by  
step*

**Ryan Turner**

**© Copyright 2019 – Ryan Turner**

**All rights reserved.**

The content contained within this book may not be reproduced, duplicated, or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book, either directly or indirectly.

**Legal Notice:**

This book is copyright protected. It is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author do not engage in the rendering of legal, financial, medical, or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, that are incurred as a result of the use of the information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

# Table of Contents

- [Introduction](#)
- [Chapter 1: What is JavaScript](#)
  - [Client-Side JavaScript](#)
  - [Advantages of JavaScript](#)
  - [Limitations of JavaScript](#)
  - [Javascript Placement](#)
  - [JavaScript in External Files](#)
- [Chapter 2: Values, Types, and Operators](#)
  - [Strings](#)
  - [Unary operators](#)
- [Chapter 3: Program Structure](#)
  - [Bindings](#)
  - [Functions](#)
  - [for loops](#)
- [Chapter 4: Functions](#)
  - [Bindings and scopes](#)
  - [Growing functions](#)
- [Chapter 5: Data Structures: Objects and Arrays](#)
  - [Methods](#)
  - [Array loops](#)
- [Chapter 6: Higher-Order Functions](#)
  - [Transforming with map](#)
  - [Strings and character codes](#)
- [Chapter 7: The Secret Life of Objects](#)
  - [Prototypes](#)
  - [Polymorphism](#)
- [Chapter 8: A Robot](#)
  - [The task](#)
  - [Simulation](#)
- [Chapter 9: Bugs and Errors](#)
  - [Testing](#)
  - [Debugging](#)
- [Chapter 10: Regular Expressions](#)
  - [Sets of characters](#)
  - [String and Word boundaries](#)
- [Chapter 11: Modules](#)

[Packages](#)

[Evaluating data as code](#)

[Chapter 12: Asynchronous Programming](#)

[Callbacks](#)

[Collections of promises](#)

[Chapter 13: Parsing](#)

[Special forms](#)

[Functions](#)

[Conclusion](#)

[References](#)

# **Introduction**

JavaScript is a distinguished and deciphered programming language that adheres to the ECMAScript specification. The programming language consists of high-powered typing, the curly bracket syntax, top-quality functions, and prototype-based object-orientation. Accompanied by HTML and CSS, JavaScript is one of the significant technologies of the World Wide Web. JavaScript facilitates interchangeable web pages and is a vital part of web applications. Almost every website utilizes JavaScript, and all the significant web browsers consist of the JavaScript engine to function correctly. Brendan Eich developed JavaScript in 1995 when he was with Netscape Communications. JavaScript aids functional, event-driven, prototype-based and object-oriented programming styles. JavaScript upgrades web user interface by affirming activities taken on the client-side by the user. You can insert JavaScript engines into different types of host software, which includes databases and web servers, and non-web related programs like word processors.

The concepts of JavaScript are explained further in this book with the goal to help you learn and understand JavaScript language without pressure. The knowledge of HTML, text editor, web browser, and CSS are all that's needed to learn JavaScript. To get the learning process started, one of the essential tools is a text editor, and it is required to write codes. You'll also need a browser to unveil your developed web pages. There are different types of text editors such as the Sublime Text, Notepad++ and browsers such as Firefox, Google Chrome, and so on.

# **Chapter 1: What is JavaScript**

JavaScript, abbreviated JS, is a high-level programming language introduced to add specific programs to web pages, and it has been adopted by all major web browsers. JavaScript can be used to build interactive web applications to function appropriately without reloading every page per-action. JavaScript is used to create different forms of activities within web pages and is one of the essential components of the World Wide Web (www), which are HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets). JavaScript and HTML are used to develop web pages. JavaScript brings a page to life by adding special effects such as sliders, pop-ups, form validations, etc. CSS determines the color intensity, image size, background colors, typeface, font size, etc.

## **Client-Side JavaScript**

Client-Side JavaScript is the usual form of JavaScript language. The script must be inserted or referenced by an HTML document so that the browser can interpret the code. It enables web pages to include interactive programs with the user, develop HTML content dynamically, and command the browser. Users use JavaScript code when they want to submit forms and also determine if all entries are valid before it transfers them to the server.

## **Advantages of JavaScript**

- With JavaScript, users can organize input before getting the page sent to the server, which automatically reduces loads on the server.
- JavaScript enables swift response to page visitors. The page does not have to reload before visitors can see if there was an error in typing.
- JavaScript is used to build a reactive interface that gives a reaction when the mouse hovers over them.

## **Limitations of JavaScript**

JavaScript programming language lacks the following essential features. They are:

- For security reasons, client-side JavaScript does not read or write files
- JavaScript for networking applications
- JavaScript does not contain multiprocessor capabilities

JavaScript Development Tools

You do not need an expensive development tool to write JavaScript codes. You can write with a simple text editor such as Microsoft FrontPage, Macromedia Dreamweaver MX, Macromedia HomeSite 5, etc.

## **Javascript Placement**

JavaScript code is inserted anywhere in an HTML document. However, you want to insert JavaScript code in an HTML file like this:

The Script within <head> </head> part.

The Script within <body> </body> part.

The Script within <body> </body> and <head> </head> part.

Include external file Script in the <head>...</head> part.

JavaScript in <head>...</head> Section

In some cases, when you want the script to run on a special event, such as an action when a button clicked, place the script in the head section like this:

```
<html>
<head>
<script type="text/JavaScript">
<!--
function sayHi() {
alert("Hello World")
}
//-->
</script>
</head>
<body>
Tap here for result
<input type="button" onclick="sayHi()" value="Say Hi" />
</body>
</html>
```

JavaScript in <body>...</body> Section

Sometimes you want a script to run immediately and to create script on the content page, insert the script within the <body> section. This is what the code should look like:

```
<html>
<head>
</head>
<body>
```

```
<script type="text/JavaScript">
<!--
document.write("Hello World")
//-->
</script>
<p>This is web page body </p>
</body>
</html>
```

JavaScript in `<body>` and `<head>` Sections

Insert your JavaScript code in `<head>` and `<body>` section like this:

```
<html>
<head>
<script type="text/JavaScript">
<!--
function sayHi() {
alert("Hello World")
}
//-->
</script>
</head>
<body>
<script type="text/JavaScript">
<!--
document.write("Hello World")
//-->
</script>
<input type="button" onclick="sayHi()" value="Say Hi" />
</body>
</html>
```

## JavaScript in External Files

To avoid the use of identical JavaScript code repetition, the language enables you to create an external file and store JavaScript codes and then integrate the external file into the HTML files. The sample below displays how to integrate an external JavaScript file within the HTML code utilizing the `script` tag and `src` attribute.

```
<html>
<head>
```

```
<script type="text/javascript" src="filename.js" ></script>
</head>
<body>
.....
</body>
</html>
```

The external file source file should be saved with an extension .js.

### Summary

In this first chapter, we explained the origination of JavaScript and its placement into internal and external files. We discussed the use of JavaScript to build interactive web applications to perform appropriately without having to reload every page per-action.

### Exercise

How do you insert JavaScript code into the head, body and include external file script in an HTML document?

### Solution

The Script for the head part is <head> </head>.

The Script for the body part is <body> </body>.

The Script within the body is <body> </body> and <head> </head>.

The Script to Include external file Script in the <head>...</head>.

## **Chapter 2: Values, Types, and Operators**

In the world of computers, there is data. You can create new data, read data and change data, which are all stored as a look-alike long succession of bits. Define bits as zeros and ones that take a strong or weak signal, and high or low electrical charge from inside the computer. All data and pieces of information are described as a succession of zeros and ones and represented in bits.

### **Values**

Take a deep breath and think of an ocean of bits. The latest PCs contains more than 30 billion bits in its data storage. We use bits to create values. The computer can function correctly because every bit of information is split into values. Every value consists of a type that influences its role, and values can be numbers, text or functions, etc. To generate value, you need to invoke its name, and it appears from where it was stored.

### **Arithmetic**

Arithmetic is the major thing to do with numbers. The multiplication, addition, and subtraction of more than one number to produce another number is an arithmetic operation. This is an example of what they look like in JavaScript:

`100 + 4 * 11`

The + and \* symbols are called operators, the first means addition while the other means multiplication. An operator inserted between two values will produce another value. The - operator is for subtraction and the / operator is for the division. If operators show together without parentheses, the precedence of the operators decides the way they are applied. If several operators with the same precedence show right next to each other like `1 - 2 + 1`, apply them left to right: `(1 - 2) + 1`.

### **Special numbers**

JavaScript consists of three unique values that do not act like numbers but are regarded as numbers. Infinity and -Infinity are the first two, which mean the positive and negative infinities, and the last value is the NaN. NaN says “not a number,” although it is a value of the number type.

### **Strings**

A string is the succession of numbers. The string is the next data type, and they represent text. Strings confine their content in quotes.

`'Down the walkway path.'`

"On top of the roof."

'I am at home.'

Quotes are used in different types like the double quotes, single quotes, or the backticks to mark strings. It is essential that the strings match. The elements within the quotes create a string value by JavaScript. JavaScript uses the Unicode standard to assign a number to every character needed, including Arabic, Armenian, Japanese, etc. You cannot subtract, multiply, or divide strings but you can use the + operator, which will not add but concatenates two strings together. Concatenation means to glue strings together.

## Unary operators

Symbols do not represent all the operators. You can write some operators in words. A clear example is a type of operator, and this operator creates a string value with the name of the typeof its attached value.

```
console.log(typeof 4.5)  
// → number  
console.log(typeof "x")  
// → string
```

The second displayed operator is called the binary operator because they use two values, while operators that use one value are the unary operator.

```
console.log(- (10 - 2))  
// → -8
```

### Boolean values

Boolean is a value that differs only between two possibilities such as “on” and “off” and so on. The Boolean consists of only true and false.

### Comparison

Comparison is a way to create Boolean values:

```
console.log(3 > 2)  
// → true  
console.log(3 < 2)  
// → false
```

The > and < characters are the signs that represent “is greater than” and “is less than,” accordingly. You can use binary operators in a Boolean value that determines if the contained value is true or false.

You can compare strings in the same manner.

```
console.log("Aardvark" < "Zoroaster")  
// → true
```

You can instruct strings in alphabetical order, uppercase letters are often “less” than lowercase, so "Z" < "a," and non-alphabetic characters (! -, and so on) are also present in the ordering. When JavaScript want to compare strings, it recognizes characters from left to right, differentiating the Unicode codes individually.

Here are other related operators <= (less than or equal to), >= (greater than or equal to), == (equal to), and != (not equal to).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

In JavaScript, there is only one value that is not equal to itself, which is the NaN (“not a number”).

```
console.log(NaN == NaN)
// → false
```

### Logical operator

JavaScript endorses only three operators that you can apply to Boolean values, they are and, or, and not. The && operator signifies logical or and its results depend on where the values imputed are true or false.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

The || operator indicates logical or. This operator outputs true if the given value is true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

An exclamation mark (!) indicates Not. It is an operator that overturns the set value, and it changes from true to false and false to true.

The || consists of the lowest precedence of all operators, then &&, the comparison operators (>, ==, etc.), and so on. The example below states that parentheses are necessary:

```
1 + 1 == 2 && 10 * 10 > 50
```

Empty values

Null and undefined are the only two types of special values that are used to indicate the non-appearance of an important value. They contain zero data.

#### Automatic type conversion

Earlier I said that JavaScript accepts practically all given programs, including programs with odd behaviors. Automatic type conversion illustrates in the following expressions:

```
console.log(8 * null)  
// → 0  
console.log("5" - 1)  
// → 4  
console.log("5" + 1)  
// → 51  
console.log("five" * 2)  
// → NaN  
console.log(false == 0)  
// → true
```

When you assign an operator the wrong value, JavaScript silently returns that value to the exact type it requires using the type coercion rule. In the first expression, the null turns to 0, and the 5 in the second remains 5 (from string to number). In the third expression, there was a string concatenation before the numeric addition, which converts the 1 to 1 (from number to a string). When odd numbers such as "five" or undefined changes to the number, it gets the value of NaN. If you want to differentiate between values of the same type using ==, the output should be true if the values are similar except NaN. If you want to test if a value contains a real value, use the == (or! =) operator to compare it. To avert unexpected type conversions, use the three-character comparison operators.

#### Short-circuiting of logical operators

The && and || are called the logical operators. They are used to hold several types of values in a specific way. They change the values contained in the left side to Boolean type to decide, although it depends on the operators and the type of generated result, but will always reinstate the left or right-hand value. The || operator sends back value to the left when it can be changed to true and will reinstate the value to the right.

```
console.log(null || "user")  
// → user  
console.log("Kate" || "user")
```

// → Kate

This function is used to return values to its default value and placed within an empty value as a replacement. Strings and numbers to Boolean value conversion rules indicate that 0, NaN, and empty string ("") count as false while other values are true. Therefore `0 || -1` outputs `-1`, and `"" || "!"` yields `"!"`. The `&&` operator operates identically but the other way around. When values to the left can be changed to false, return the value, or it sends the value to the right. Both operators evaluate the value to the right only when it is required. For example, we have the following values set as `true || X`, the value of `X` will be true and will not consider it. The same rule applies to the `false && X`, which the `x` is false and will overlook it. You can call this process the short-circuit evaluation.

## Summary

This chapter looks at the four types of JavaScript values, which are strings, numbers, undefined values and Booleans. These values are developed by inserting their names as true, null or value (13, "ABC"). Operators can integrate and change values. We looked at binary operators for arithmetic (+, -, \*, /, and %), string concatenation (+), comparison (==, !=, ===, !==, <, >, <=, >=), and logic (&&, ||), and also various unary operators (- to nullify a number, ! to nullify logically, and type of to search for a value's type) and a ternary operator (?:) to choose one of two values depending on a third value. You will get sufficient information to use JavaScript like a small calculator, and you will improve in the following chapters.

## Exercise

Write a JavaScript practice to build a variable through a user-defined name.

## Solution

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-width">
6   <title>Create a variable using a user-defined name</title>
7 </head>
8 <body>
9
10 </body>
11 </html>
```

## JavaScript Code:

```
1 var var_name = 'abcd';
2 var n = 120;
3 this[var_name] = n;
4 console.log(this[var_name])
```



# Chapter 3: Program Structure

We will begin the expansion of the JavaScript programming language commands beyond sentence fragments and nouns.

## Expressions and statements

A block of code that manufactures a value is called an expression. A single value is written precisely as 22, or psychoanalysis is called an expression. An expression within a binary operator set to two expressions or within parentheses is an expression. An expression that can accommodate other expressions comparably to how the human language translates. It enables us to develop expressions that narrate the arbitrarily complex computations. When an expression is comparable to a sentence fragment, the JavaScript statement will complete the sentence. A list of statements is called a program, and the most accessible type of statement is a line of code with a semicolon ending it.

For instance:

```
1;  
!false;
```

With this function, a statement can stand independently, and it can add a feature that changes colors occasionally to a screen or modify the inner state of the machine influencing the following statements after it.

## Bindings

JavaScript uses binding of variables to hold values, for instance:

```
Get money = 5 * 5;
```

Get is the keyword in that statement, and it means that the sentence will specify a binding together with the binding name and value can be attached using the = operator and an expression. Use a defined binding as an expression, an expression containing the binding value. Here is an example below:

```
let ten = 10;  
console.log(ten * ten);  
// → 100
```

The = operator is used on existing bindings to remove the binding from a set value and point them to a new one.

```
let mood = "light";  
console.log(mood);  
// → light
```

```
mood = "dark";  
console.log(mood);  
// → dark
```

Defining a binding without assigning a value will result in nothing to hold. Therefore, asking for the value of a binding produce the undefined value. You can define multiple binding by a single statement divided by commas.

```
let one = 1, two = 2;  
console.log(one + two);  
// → 3
```

Use the var and const words to develop bindings familiarly:

```
var name = "Ayad";  
const greeting = "Hello ";  
console.log(greeting + name);  
// → Hello Ayad
```

The word const means constant. It describes a resolute binding, which points to its set value for its lifetime. These words are often used to declare a name to a value to enable easy reference later on.

### Binding names

Although there are some reserved words like const, class, default, break, continue, delete, do, else yet any word can be called a binding name, it can also consist of digits but do not begin the statement with a figure. A binding name can also contain dollar signs (\$) or underscores (\_) but do not entertain any other special characters or punctuation.

### The environment

The group of binding and values existing at a stipulated time is known as the environment. When you launch a program, the environment holds the language standard binding and often contains the binding that creates interaction with system surroundings. For instance, a browser carries functions that communicate with the launched website as well as read the keyboard and mouse input.

### **Functions**

A bit of program enclosed in value is called a function, and these values are added to launch the wrapped program. For instance, a function that displays a small dialog box for user input.

```
Prompt ("Enter passcode");
```

The process of effecting a function is described as a calling, invoking, or applying. Call a function by inserting parentheses following an expression that provides a function value. Set the parenthesis value to the program within the function. Define values set to functions as arguments.

### The console.log function

In the above samples, we used the console.log to output values. All major browsers utilize the console.log function to state out its arguments to the device that outputs the text. In modern browsers, the output is often in the JavaScript console that is invisible by default; you can tap the F12 command on your keyboard or the command-option-I on Mac. Binding names do not accommodate period characters, but console.log does because it is an expression that reclaims the log property from the console binding value.

### Return values

Functions are used to produce side effects. They can also provide values that do not need side effects. For instance, the function Calculate.max will take a sum of number arguments and returns the greatest.

```
console.log(Calculate.max(2, 4));
// → 4
```

JavaScript regards anything that provides value as an expression, which enables function calls to attach into the substantial expression. Let's call the Calculate.min, which is a direct opposite to Calculate.max:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

### Control flow

When more than one statement is within a program, they execute in a story form from the beginning of the code to the end. This type of program consists of two types of statement; the first demand numbers from the user while the second displays the square of the number and executes immediately after the first.

```
let theNumber = Number(prompt("Select a digit "));
console.log("Your digit is the square root of " + theNumber * theNumber);
A value is changed to a number by the function number and outputs a string value.
```

### Conditional execution

The keyboard is used to develop conditional executions in JavaScript. You may want some code to execute if, and only if, a specific condition is positive. Let us display the square of the input if only it is a number.

```
let theNumber = Number(prompt("Select a digit "));  
if (! Number.isNaN(theNumber)) {  
    console.log("Your digit is the square root of " +  
    theNumber * theNumber);  
}
```

The if keyword performs or evades a statement determined by the Boolean expression value. Type the determining expression after the keyboard within the parentheses accompanied by the statement to accomplish.

The Number.isNaN function is a high-level JavaScript function that outputs only true if the statement is declared as NaN. The number function returns NaN when an assigned string is not a valid number. Statements after the if statement is enclosed in braces ({and}). Braces are used to categorize different numbers of statement within an individual statement.

28

```
if (1 + 1 == 2) console.log("It's true");  
// → It's true
```

The else keyword can be utilized together with the if statement to provide two different execution paths.

```
let theNumber = Number (prompt("Select a digit"));  
if (! Number.isNaN(theNumber)) {  
    console.log("Your digit is the square root of " +  
    theNumber * theNumber);  
} else {  
    console.log("Hey. Why didn't you give me a number?");  
}
```

If there are more than two paths to choose from, you can “chain” multiple if/else pairs together. Below is an example:

```
let num = Number (prompt("Pick a number"));  
if (num < 10) {  
    console.log("Small");  
} else if (num < 100) {  
    console.log("Medium");  
} else {  
    console.log("Large");  
}
```

The program confirms if the num is less than 10, and if it is, it selects that branch, and then displays "Small". But if the num is greater than 10, it

selects the else branch which consists of a second if statement of its own.

## **while and do loops**

This is the way to write a program that displays all the even numbers from 0 to 12.

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

A way to run a block of code multiple times is described as a loop.

The looping control flow enables the user to return to certain points in written programs and redo it with the current state of the program. Integrate this with a binding that enumerates as follows:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
// ... etcetera
```

A statement that begins with the keyword and at the same time producing a loop. The keyword while accompanied by an expression in parentheses followed by a statement like the if statement. The loop becomes continuous until the expression provides a value that states true when translated to Boolean. Each time the loop repeats, numbers adopt a number twice of their previous value. Let us write a program that evaluates and display the value of 2<sup>10</sup> (2 raised to the power of 10th). We will utilize two bindings. One to keep an eye on our result and the other to calculate how many times the value of two multiplied by the result. It multiplies until the second binding reaches 10:

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
```

```
counter = counter + 1;  
}  
console.log(result);  
// → 1024
```

A do loop is a command structure related to a while loop. A do loop always take place at least once in a statement, and then it begins to check if it ends after the first execution only. Follow the following steps:

```
let yourName;  
do {  
  yourName = prompt("Who are you?");  
} while (! yourName);  
console.log(yourName);
```

## for loops

A lot of loops follow the while loop pattern, creating a counter binding to track the loop's progress. Then a while loop with a test expression to see if the counter reaches the set value.

```
for (let number = 0; number <= 10; number = number + 4) {  
  console.log(number);  
}  
// → 0  
// → 4  
// etc
```

The parentheses must consist of two semicolons after the keyword, the first part that is before the first semicolon separates the loop by describing a binding. The other part is the statement that determines if the loop should continue. Here is the code that evaluate 210 using for instead of while:

```
let result = 1;  
for (let counter = 0; counter < 10; counter = counter + 1) {  
  result = result * 2;  
}  
console.log(result);  
// → 1024
```

Breaking Out of a Loop

There are other ways to end a loop other than making a looping condition give a false. Break can influence jumping out of the confined loop. The break statement evaluates this program if the first number, which is both greater than or equal to 20, and can be divided by 7.

```
for (let current = 20; current = current + 1) {  
  if (current % 7 == 0) {  
    console.log(current);  
    break;  
  }  
}  
// → 21
```

The (%) operator is used to check if a number can be divided by another number. If it can be divided, then the remaining division is zero. The for in the example is not checked at the end of the loop, which means until the break statement inside is implemented the loop becomes infinite and never stops.

Updating bindings succinctly

When you are in a loop, programs will update a binding regularly to store a value based on its previous value.

```
counter = counter + 1;
```

JavaScript offers a shortcut.

```
counter += 1;
```

Related shortcuts function for several other operators, like result \*= 2 to multiply result or counter -= 1 to count in descending order.

This enables us to minimize our counting example.

```
for (let number = 0; number <= 12; number += 2) {  
  console.log(number);  
}
```

For counter += 1 and counter -= 1, here are shorter equivalents:

counter++ and counter--.

Dispatching on a value with switch

Dispatching on a value with switch Codes can look like this:

```
if (x == "value1") action1();  
else if (x == "value2") action2();  
else if (x == "value3") action3();  
else defaultAction();
```

The switch is used to express the above form of dispatch straightforwardly. Below is a good example:

```
switch (prompt("What is the atmosphere like?")) {  
    case "rainy":  
        console.log("Do not forget to come with an umbrella.");  
        break;  
    case "sunny":  
        console.log("Be light with your dressing.");  
    case "cloudy":  
        console.log("Move out.");  
        break;  
    default:  
        console.log("Weather type Unknown!");  
        break;  
}
```

### Capitalization

Names binding do not allow spaces, but it supports the usage of several words to define the binding values. Here are a few types that can be utilized when binding names with different words:

fuzzylittleturtle  
fuzzy\_little\_turtle  
FuzzyLittleTurtle  
fuzzyLittleTurtle

### Comments

Sometimes raw codes do not transmit every single piece of information that you want the program to send to readers or spreads the message in a way people will find it hard to decipher. Other times you feel like you should attach some similar thoughts to your program, the comment performs this function. A comment is a bit chunk of the text contained in a program, but the computer ignores it. To code a single line comment, use the two slash characters (//), followed by the comment text.

```
let accountBalance = calculateBalance(account);  
// It's a yellow-orange on the mango tree accountBalance.adjust();  
// Catching green tatters in our home. let report = new Report();  
// Where the sun and the moon meets: addToReport(accountBalance, report);  
// It's a large hallway, and the lights are quite amazing.
```

A // comment appears only at the end of the line. Any text within /\* and \*/ ignore it, and it does not matter if it contains line breaks. It is used to attach blocks of information about a program or file.

### Summary

Now you understand that you can develop a program through the use of statements, which statement itself can contain several statements. Statements consist of expressions, and you can create expressions through smaller expressions. Setting statements after each another provides an executed program from top to bottom. Disturbances can also come into the flow of influence through conditional (if, else, and switch) and looping (while, do, and for) statements. And we touched bindings, they categorize bits of data under a name, and they can also track state within the program. The defined bindings enjoy the environment better. We also touched on the functions section being unique values that summarize a piece of program. You can call them by typing functionName(argument1, argument2). This type of function call is an expression and can provide value.

### Exercise

Write a JavaScript program that displays the larger and accepts double integers.

### Solution

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset=utf-8 />
5 <title>Write a JavaScript program that accept two integers and display the larger</title>
6 </head>
7 <body>
8
9 </body>
10 </html>
```

## JavaScript Code:

```
1 var num1, num2;
2 num1 = window.prompt("Input the first integer", "0");
3 num2 = window.prompt("Input the second integer", "0");
4
5 if(parseInt(num1, 10) > parseInt(num2, 10))
6 {
7   console.log("The larger of " + num1 + " and " + num2 + " is " + num1 + ".");
8 }
```

```
1 if (num1 > num2) {
2   console.log(`The larger of ${num1} and ${num2} is ${num1}.`)
3 } else {
4   console.log(`The value ${num1} is equal to ${num2}.`)
5 }
```

It is important to know that you can write .length at the end of a string to find its length.

```
let abc = "abc";
console.log(abc.length);
// → 3
```

# Chapter 4: Functions

Functions are the alpha and omega of JavaScript programming language. The concept of enclosing a bit of program contained in value has a lot of advantages. It provides a way to organize more prominent programs, to minimize repetition, relate names with subprograms, and separate programs from themselves. Functions are used to describe new words.

Defining a function

A function is a systematic binding whereby a function defines the binding value. The code below indicates a function that provides the square of a stated number:

```
const square = function(x) {  
    return x * x;  
};  
console.log(square(12));  
// → 144
```

Create a function with an expression that begins with the keyboard function. Functions consists of a body and a set of parameters that accommodates the yet to be executed statements when you call a function. The function body should always be enclosed in braces, even when it is just an individual statement. A function can contain several parameters or zero parameter. In the below example, makeNoise contains no parameter names and power consists of two:

```
const makeNoise = function() {  
    console.log("Pling!");  
};  
makeNoise();  
// → Pling!  
const power = function(base, exponent) {  
    let result = 1;  
    for (let count = 0; count < exponent; count++) {  
        result *= base;  
    }  
    return result;  
};  
console.log(power(2, 10));  
// → 1024
```

Some functions can create values such as square and power while some results only into a side effect. A return keyword that does not contain an expression at the end will be returned as undefined.

## Bindings and scopes

Every binding consists of a scope that enables the visibility of the binding. Whenever you call a function, new illustrations of binding are developed. This creates separation between functions, every function call behaves in its own world and is easy to understand. Functions that were declared with the var keyword in the pre-2015 JavaScript and can be seen throughout the global scope. They are absent in a function.

```
let x = 10;
if (true) {
let y = 20;
var z = 30;
console.log(x + y + z);
// → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

### Nested scope

JavaScript differentiate between the global and local bindings. You can create block and functions within the other functions and blocks manufacturing several degrees of locality. For instance, this function produces the needed components to make a group hummus contains another function within it:

```
const hummus = function(factor) {
const ingredient = function(amount, unit, name) {
let ingredientAmount = amount * factor;
if (ingredientAmount > 1) {
unit += "s";
}
console.log(` ${ingredientAmount} ${unit} ${name}`);
};
ingredient(1, "can", "chickpeas");
ingredient(0.25, "cup", "tahini");
ingredient(0.25, "cup", "lemon juice");
```

```
ingredient(1, "clove", "garlic");
ingredient(2, "tablespoon", "olive oil");
ingredient(0.5, "teaspoon", "cumin");
};
```

The code within the component function can access the factor binding from the outer function. The set of bindings accessible within a block is decided by the block position in the program text. Every local scope can access the contained local scope and every scope can as well access the global scope. This concept is named lexical scoping.

### Functions as values

A function binding often behaves as a name for a particular block of program. This type of binding does not change because it is defined. Functions can be used in arbitrary expressions as well as used to save a function value into a new binding. A binding that stores a function is a systematic binding and a new value can be assigned like:

```
let launchMissiles = function() {
missileSystem.launch("now");
};

if (safeMode) {
launchMissiles = function() /* do nothing */;
}
```

### Declaration notation

There is a little way to generate a function binding. When you use the function keyword at the beginning of a statement, it will function differently.

```
function square(x) {
return x * x;
}
```

This statement describes the binding square and directs it at a stated function. This type of function definition consists of one precision.

```
console.log("My mind tells me every time:", future());
function future() {
return "I'll surpass Bill Gates";
}
```

### Arrow functions

Functions contains a third notation that does not look identical to the others. You can use the arrow (`=>`) as an alternative to the function keyword. The arrow (`=>`) contains an equal sign and the greater than character.

```
const power = (base, exponent) => {
let result = 1;
for (let count = 0; count < exponent; count++) {
result *= base;
}
return result;
};
```

If you have just one parameter name, exclude the parentheses surrounding the parameter list. Sometimes the body is a single expression instead of a block of braces, taking back the expression from the function. Therefore, square definitions perform the same task.

```
const square1 = (x) => {return x * x;};
const square2 = x => x * x;
```

When there is no parameter on an arrow function, its parameter list becomes a set of unoccupied and ineffective parentheses.

```
const horn = () => {
console.log("Toot");
};
```

You don't have to use both the function expressions and arrow functions in the language, they offer and perform the same operations.

#### The call stacks

Let us take some time to see how control goes through functions. Below is a simple program making few function calls:

```
function greet(who) {
console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

The greetings call influences to control to jump to the beginning of the function. The function call console.log gets its job done and seize control, and then send control back to the function. The function cycle ends there and transfers back to the place that calls it. The flow of control is below:

```
not in function
in greet
in console.log
in greet
not in function
```

in console.log

not in function

The computer has to recollect the context in which the call sequences occurred because functions go back to where it's called position. When Console.log completes, it must return to the end of the program. A call stack is a place where the computer stores this context. Whenever you call a function, the present context saves at the top of this stack. By the time a function goes back, it eradicates the top context and utilizes the content for continual execution.

The stack needs space to be saved into within the computer memory. The below code explains this by querying the computer that creates zero limitation between two functions back and forth.

```
function chicken() {  
    return egg();  
}  
function egg() {  
    return chicken();  
}  
console.log(chicken() + " came first.");  
// → ??
```

### Optional Arguments

The below code functions properly without interference:

```
function square(x) {return x * x;}  
console.log(square(4, true, "hedgehog"));  
// → 16
```

The square is described with just one parameter and we call it three. That is possible because the programming language disregards the surplus arguments and recognizes the square of the first one. JavaScript is very tolerant about the amount of arguments being passed to a function. The good side of this character is that it enables calling functions with separate numbers of arguments:

```
function minus(a, b) {  
    if (b === undefined) return -a;  
    else return a - b;  
}  
console.log(minus(10));  
// → -10
```

```
console.log(minus(10, 5));
// → 5
```

When an = operator is written after a parameter and an expression, the expression value will restore the non-specified argument. If you want to pass and not produce the second argument, the default becomes two and the function acts like a square.

```
function power(base, exponent = 2) {
let result = 1;
for (let count = 0; count < exponent; count++) {
result *= base;
}
return result;
}
console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

### Closure

The capability to use functions as values is coupled with the fact that local bindings are created again whenever you call a function. The below code displays this example; it describes a wrap value, a function that develops a local binding and then sends back a function that enters and returns the local binding.

```
function wrapValue(n) {
let local = n;
return () => local;
}
let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

This concept is called closure. It gives the user the ability to reference a particular instance of a local binding within a confining scope. Adding a few changes, our previous example can turn into a way to develop functions that accumulates by an arbitrary amount.

```
function multiplier(factor) {  
    return number => number * factor;  
}  
let twice = multiplier(2);  
console.log(twice(5));  
// → 10
```

## Recursion

A function can call itself but should not call itself regularly to avoid stack overflow. Recursive function is a function that calls itself. Recursion enables few functions written in separate styles. For instance, the below code is the execution of power.

```
function power(base, exponent) {  
    if (exponent == 0) {  
        return 1;  
    } else {  
        return base * power(base, exponent - 1);  
    }  
}  
console.log(power(2, 3));  
// → 8
```

There is one problem with this execution. It is slower compared to other looping versions. Utilizing a single loop is considered low cost than the multiple calling of functions. Although that does not make recursion an ineffective option of looping, few problems are solved with recursion easier than with the use of loops. Problems that need inspecting or processing several branches. Check this out: we begin from number 1 and continuously add 5 or multiply by 3. For instance, the number 13 can be obtained by multiplying by 3 and the addition of 5 twice, thereby we cannot attain the 15. The code is the recursive solution:

```
function findSolution(target) {  
    function find(current, history) {  
        if (current == target) {  
            return history;  
        } else if (current > target) {  
            return null;  
        } else {  
            return find(current + 5, `(${history} + 5)`)  
        }  
    }  
}
```

```

    find(current * 3, `(${history} * 3));
}
}
return find(1, "1");
}
console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)

```

The inner function `find` takes two arguments, the current number and a string that documents the process of attaining this number. If a solution is found, it sends back a string that displays the route to the target and if no solution is found, the returned value will be null. To achieve this, the function executes one of three actions. If your target is the current number, you can attain that target by using the current history so it is sent back. Sometimes the number is larger than the target, but you do not need to explore this option because addition and subtraction can only enlarge the number so it is sent back as null. If you remain beneath the target number, both paths that begin the current number is tried by calling itself twice, one to add and the other to multiply. If the first call sends something valid back in return, then good, if not, return the second call, it does not matter whether a string or null is provided. Below is an illustration of how functions provide effects. This example searches for a remedy for the number 13.

```

find(1, "1")
find(6, "(1 + 5)")
find(11, "((1 + 5) + 5)")
find(16, "(((1 + 5) + 5) + 5)")
too big
find(33, "(((1 + 5) + 5) * 3)")
too big
find(18, "((1 + 5) * 3)")
too big
find(3, "(1 * 3)")
find(8, "((1 * 3) + 5)")
find(13, "(((1 * 3) + 5) + 5)")
found!

```

The indentation specifies profound of the call stack.

## ***Growing functions***

Functions can be introduced into programs in two ways. The first is by writing similar codes a lot of times, which enables more mistakes while the second is to search for a few functionalities that have not been written and deserves its own function. You can begin by naming the function and write the body. Below is an example. We will write a program that reproduce two numbers—the amount of chicken and cows available on a farm.

```
007 Cows
011 Chickens
function printFarmInventory(cows, chickens) {
let cowString = String(cows);
while (cowString.length < 3) {
cowString = "0" + cowString;
}
console.log(` ${cowString} Cows`);
let chickenString = String(chickens);
while (chickenString.length < 3) {
chickenString = "0" + chickenString;
}
console.log(` ${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

Writing length after a string expression determines the length of the particular string. The loop continues to add zeros at the beginning of the number strings until they comprise of three characters.

Here is a better attempt:

```
function printZeroPaddedWithLabel(number, label) {
let numberString = String(number);
while (numberString.length < 3) {
numberString = "0" + numberString;
}
console.log(` ${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, pigs) {
printZeroPaddedWithLabel(cows, "Cows");
printZeroPaddedWithLabel(chickens, "Chickens");
printZeroPaddedWithLabel(pigs, "Pigs");
```

```

}

printFarmInventory(7, 11, 3);
Instead of taking out the replicated part of the program, try to pick out an
individual concept with the following steps:
function zeroPad(number, width) {
let string = String(number);
while (string.length < width) {
string = "0" + string;
}
return string;
}

function printFarmInventory(cows, chickens, pigs) {
console.log(` ${zeroPad(cows, 3)} Cows`);
console.log(` ${zeroPad(chickens, 3)} Chickens`);
console.log(` ${zeroPad(pigs, 3)} Pigs`);
}
printFarmInventory(7, 16, 3);

```

A function can also be used to print aligned tables of numbers.

#### Functions and side effects

Functions can consist of side effects and return a value. They develop values that integrate easily in new ways much more than functions that produce side effects. A pure function is a unique value manufacturing function that does not depend on either code side effects. For instance, it will not recognize a global binding with a changeable value. It also consists of a callable property together with the same arguments and produces equal value.

#### Summary

In this chapter, you now understand how to write your functions. Knowing how to write the function keyword and when to use an expression, you can now build a function value. When you use a function as a statement, it can set a binding and name a function to be its value. You can also use arrow functions to build functions.

```

// Define f to hold a function value
const f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {

```

```
return a * b * 3.5;  
}  
  
// A less verbose function value  
let h = a => a % 3;
```

The major perspective of understanding functions is knowing the scopes. Every block builds a new scope. Parameters and bindings set in a specific scope are local and invisible from the outside. Bindings set with var act differently, and they result in the global scope. Differentiating the tasks that the program executes into various functions is important. You do not need to repeat yourself often, and functions can arrange a program by categorizing code into bits that perform specific actions.

### Exercise

Write a JavaScript program to see if a number is even or not.

### Solution



```
21 console.log(is_even_recursion(234)); //true  
22 console.log(is_even_recursion(-45)); // false  
23 console.log(is_even_recursion(-45)); // false
```

Output:

```
true  
false  
false
```

JavaScript Code:

```
1 function is_even_recursion(number)  
2 {  
3     if (number < 0)
```

```
4  {
5      number = Math.abs(number);
6  }
7  if (number==0)
8  {
9      return true;
10 }
11 if (number==1)
12 {
13     return false;
14 }
15 else
16 {
17     number = number - 2;
18     return is_even_recursion(number);
19 }
20 }
```

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Use recursion to determine if a number is or not.</title>
6 </head>
7 <body>
8 </body>
9 </html>
```

# Chapter 5: Data Structures: Objects and Arrays

Data layouts develop from Booleans, strings, and numbers fragments. Several types of information use more than one chunk. Objects enable the user to collate values together with other objects to develop more compound layouts. In this chapter, you will understand the concepts of solving a real problem at hand.

## Data sets

If you want to work with a lot of digital data, you must represent the digital data in the machine memory. For instance, you need to represent a group of numbers like 2, 3, 5, 7, and 11. Now let's get prolific with our use of strings as strings can contain quite a lot of data. We will use 2 3 5 7 11 as the representation. JavaScript programming language offers a unique data type for saving sequences of values, which is called an array. Write an array as a list of values within the between square brackets, differentiated by commas.

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

The notation used to get elements within an array also utilize the square brackets. A complete square brackets pair right after an expression that contains another expression within. When you start counting elements in an array, you begin from zero and not one. Therefore the first element is redeemed with `listOfNumbers[0]`.

## Properties

The previous chapters have contained expressions like the `myString.length` (to calculate the length of a string) and `Math.max` (the maximum function). These are properties that can approach the property values. Virtually all JavaScript values contain properties. The few values that do not contain properties are `null` and `undefined`. If you access a property with a nonvalue, the result will be an error.

```
null.length;
// → TypeError: null has no properties
```

In JavaScript, the two major ways to evaluate properties are with a dot and square brackets. The `value.x` and `value[x]` can evaluate the property on

value, not particularly the same property. How you use x is different from how you use a dot.

If you want to use the dot, the text you input after the dot is the name of the property. And if you're going to use the square brackets, the expression within the brackets is accessed to redeem the property name. Value.x delivers the property of value called "x," while value[x] accesses the expression x and utilize the result, transformed into a string as the name of the property. If the property you want is named color, you should type value.color. You can withdraw the property that the value called within the binding i, type value[i]. The name of properties are strings, but the dot notation performs with only valid binding names. Therefore if you want to evaluate a property named 2 or John Doe, use the square brackets: value[2] or value["John Doe"].

The elements within an array save as the properties of an array that utilize numbers as property names. An array's length property determines how many items it contains. The name of that property is a valid, binding name. Type array.length to determine the length of an array, it is much easier to type than the array["length"].

## Methods

The string and array objects accommodate several properties that store function values.

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

Each string consists of a toUpperCase property. When you call this property, it sends back a copy of the string whereby all the containing letters transform into uppercase. Although the toUpperCase do not go through an argument, the function can evaluate the string "Doh," which is the property of the value we called above. Properties that consist of functions are known as methods. The below example illustrates two methods that can be used to influence arrays.

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
```

```
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

The push method is used to attach a value to the end of an array while the pop method does the direct opposite. It eradicates the last value within an array and sends it back. These are generational terms of operations on a stack. A stack is a data layout that enables users to push values in and pop them out in the opposite direction so that the added value will remove first.

## Objects

You can represent a group of log entries as an array although each string entry is required to save a list of activities as well as a Boolean value that specifies if Jacques transformed to a squirrel or not. We will group this into a single value and place the grouped values within an array of log entries. Arbitrary collections of properties are the value of the type object. We will develop an object utilizing braces as an expression.

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

The braces contain a list of properties divided by commas. Every property contains a name accompanied by a value and a colon. When you write an object over several lines, make sure your indenting is good because it enables better readability. Quoted invalid binding names properties or valid numbers should.

```
let descriptions = {
  work: "Went to work,"
  "touched tree": "Touched a tree."
};
```

You can use braces in two different ways in JavaScript. It is used at the beginning of a statement and can also be utilized to begin a block of statements. You can specify a value to a property expression using the = operator. It takes the place of the property value, and if it is in existence, it develops a new one. The delete operator is a unary operator that eradicates the property name of an object. Below is an illustration:

```
let anObject = {left: 1, right: 2};  
console.log(anObject.left);  
// → 1  
delete anObject.left;  
console.log(anObject.left);  
// → undefined  
console.log("left" in anObject);  
// → false  
console.log("right" in anObject);  
// → true
```

The binary in operation determines if an object or string contains a named property. The major differentiation between specifying a property to undefined and deleting it is that the object consists of a property and the deleting does not possess the property, so the value returns as false. To determine what properties contained in an object and sends back an array of strings with property names of the object.

```
console.log(Object.keys({x: 0, y: 0, c: 2}));  
// → ["x", "y", "c"]
```

The Object.assign function is used to copy properties from one object to another.

```
let objectA = {a: 1, b: 2};  
Object.assign(objectA, {b: 3, c: 4});  
console.log(objectA);  
// → {a: 1, b: 3, c: 4}
```

An array is a unique object used for saving succession of things. When you access the typeof [], it provides "object." The below illustration stands for the journal that Jacques stores as an array of objects.

```
let journal = [  
  {events: ["work", "touched tree", "pizza",  
    "running", "television"],  
   squirrel: false},
```

```
{events: ["work", "ice cream", "cauliflower",
"lasagna", "touched tree", "brushed teeth"],
squirrel: false},
{events: ["weekend", "cycling", "break", "peanuts",
"beer"],
squirrel: true},
/* and so on...
];

```

## Mutability

We have discussed different values such as Booleans, strings, and numbers whose values are difficult to change, although they can be combined to acquire new values from them. Objects are different from values, and it enables property changing; they can create different content for a single object value at different times. When there are two numbers, 120 and 120, they are considered the same, and there is a similarity between setting two references to one object and possessing two separate objects containing the same properties. Check the below code:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};
63
console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false
object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

The object1 and object2 bindings hold the same object, and that is why altering object1 one influences the value of object2. They are identical by nature. The binding object3 points to a separate object that consists of similar properties as the object1 but has a different life. Bindings can be constant or changeable, but it does not influence their values. A const binding an object cannot be transformed and relentlessly pointing to the same object. It is the object contents that can be changed.

```
const score = {visitors: 0, home: 0};  
// This is okay  
score.visitors = 1;  
// This isn't allowed  
score = {visitors: 1, home: 1};
```

When the JavaScript's == operator is used to differentiate objects, it uses the identity to perform that task. It will translate true if the objects contain the same value. Differentiating several objects will return false, even if their properties are similar.

The lycanthrope's log

So, you begin your JavaScript interpreter and creates the environment you need to keep your journal.

```
let journal = [];  
function addEntry(events, squirrel) {  
journal.push({events, squirrel});  
}
```

Instead of proclaiming properties such as events, it sets a property name.

So then, at 10 p.m. every evening or occasionally in the following morning, after

getting down from the top shelf of your bookcase, your daily records.

```
addEntry(["work", "touched tree", "pizza", "running",  
"television"], false);  
addEntry(["work", "ice cream", "cauliflower", "lasagna",  
"touched tree", "brushed teeth"], false);  
addEntry(["weekend", "cycling", "break", "peanuts",  
"beer"], true);
```

Correlation is an evaluation of vulnerability between analytical variables. They are known as values that span from -1 to 1. An Unrelated variable is called a zero correlation. A correlation of 1 signifies that the two are related. Negative means that the variables are related perfectly but opposites to each other; one is true, and the other is false. To calculate the evaluation of the correlation between two Boolean values, utilize the phi coefficient ( $\phi$ ). This formula input is a frequency table holding the number of times it notices several mixtures of the variables. The formula's output will be a number between -1 and 1. That is the best definition of a correlation.

Let us use the event if eating pizza as an example and insert it into a frequency table whereby every number signifies the total of times, and we

used the combination in our measurements. We name the table  $n$ , and we calculate  $\varphi$  with the below formula:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_1 \cdot n_0 \cdot n_1 \cdot n_0}}$$

The notation  $n_{01}$  signifies the number of measurements where the first variable turns false (0), and the second variable becomes true (1). The pizza table,  $n_{01}$  is 9.

The value  $n_1 \cdot$  indicates the sum of all measurements where the first variable turns true, that is 5 in the table example. As well as  $n \cdot 0$  suggests the quantity of the measurements where the second variable becomes false.

The pizza table, the top of the division line would be  $1 \times 76 - 4 \times 9 = 40$ , and the below part below would be the square root of  $5 \times 85 \times 10 \times 80$ , or  $\sqrt{340000}$ . The result will be  $\varphi \approx 0.069$ , which is tiny. Therefore, eating pizza had no impact on transformations.

### Computing correlation

JavaScript enables the representation of a two-by-two table with four element arrays (`[76, 9, 4, 1]`). Other representations such as an array holding two two-element (`[[76, 9], [4, 1]]`) or an object containing property names such as "11" and "01", and the flat array, which produces the expression that access the chair short. We will explain the indices to the array as two-bit binary numbers, where the most significant digit refers to the squirrel variable while the least significant refers to the event table. For instance, the binary number 10 refers to the "Jacques did turn into a squirrel" case but the events didn't appear. It occurred four ties and binary is 2 in decimal, the number is stored as an index 2 of the array. This function calculates the  $\varphi$  coefficient from an array:

```
function phi(chair) {
    return (chair[3] * chair[0] - chair[2] * chair[1]) /
        Math.sqrt((chair[2] + chair[3]) *
            (chair[0] + chair[1]) *
            (chair[1] + chair[3]) *
            (chair[0] + chair[2]));
}
console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

The table requires two fields to acquire fields like n1• because the total number of rows and columns are stored indirectly into our data layout. If you want to extract a two-by-two table for a particular event, you must loop across all the entries and calculate the number of times the event happened to the squirrel transformations.

```
function tableFor(event, journal) {  
let table = [0, 0, 0, 0];  
for (let i = 0; i < journal.length; i++) {  
let entry = journal[i], index = 0;  
if (entry.events.includes(event)) index += 1;  
if (entry.squirrel) index += 2;  
table[index] += 1;  
}  
return table;  
}  
console.log(tableFor("pizza", JOURNAL));  
// → [76, 9, 4, 1]
```

Arrays consists of an include method, which is used to check if a set value exists in the array. This method is used by the function to decide if the event name is interested the stated event list for a given day.

## **Array loops**

The tableFor function contains a loop like this:

```
for (let I = 0; I < JOURNAL.length; I++) {  
let entry = JOURNAL[i];  
// Do something with entry  
}
```

This type of loop is used in high-level JavaScript

This is how to write that type of loop in modern JavaScript.

```
for (entrance of JOURNAL) {  
console.log(`#${entrance.events.length} events.`);  
}
```

With a loop like this, the loop continues across the elements of the value specified after of. This loop is used for strings and other data layouts.

The final analysis

A correlation is calculated for every type of event that happens in the data set. If you want to achieve that, search every type of event.

```
function journalEvents(journal) {let events = [];
```

```

for (let entry of journal) {
  for (let event of entry.events) {
    if (!events.includes(event)) {
      events.push(event);
    }
  }
}
return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", ...]

```

By moving across every event and making additions to those that are not in the events array, this function gathers all the type of event.

A lot of correlations appear to lie close to zero. Eating bread, carrots or pudding does not activate the squirrel-lycanthropy. It happens often during the weekends. Using that, below are all the correlations.

```

for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}

// → carrot: 0.0140970969
// → exercise: 0.0685994341
// → weekend: 0.1371988681
// → bread: -0.0757554019
// → pudding: -0.0648203724
// and so on...

```

let us filter results to display only correlations that are greater than 0.1 or less than -0.1.

```

for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}

// → weekend: 0.1371988681
// → brushed teeth: -0.3805211953
// → candy: 0.1296407447
// → work: -0.1371988681

```

```
// → spaghetti: 0.2425356250
// → reading: 0.1106828054
// → peanuts: 0.5902679812
```

In a correlation factors, one is stronger than the other. Eating peanuts has a powerful chance of transforming into a squirrel, thereby brushing the teeth has a notable negative effect. Here's something:

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
    !entry.events.includes("brushed teeth")) {
    entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

### Further arrayology

Here are a few more object-related concepts you should know. We begin with the use of some useful methods of an array.

The adding and removing of methods things at the beginning of an array are called unshift and shift.

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

The above program organizes a chain of tasks. You can add tasks to the end of the queue by calling `remember("groceries")`, and when you want to get something done, call `getTask()` to get (and remove) item from the queue. If you want to search for a particular value, arrays supply an `indexOf` method. The method finds its way through the array from the beginning to the end and sends back the index the value requested, if it was found it returns —or `-1` if it wasn't found. To search from the beginning to the end, use the `lastIndexOf` method.

```
console.log([1, 2, 3, 2, 4].indexOf(3));
// → 1
console.log([1, 2, 3, 2, 4].lastIndexOf(3));
// → 3
```

The `indexOf` and `lastIndexOf` takes a voluntary argument that specifies where the search begins. Another great method is the `slice`, this method begins and ends the indices and send back an array containing three elements. The beginning is inclusive while the end is exclusive.

```
console.log([0, 1, 2, 3, 5].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 5].slice(2));
// → [2, 3, 5]
```

If the end index is not stated, `slice` take all the elements after the start index. You can use the `concat` method to glue arrays together to produce a new array. The below example displays `concat` and `slice` in action.

```
function remove(arrays, index) {
  return arrays.slice(0, index)
    .concat(arrays.slice(index + 1));
}
console.log(remove(["a", "k", "c", "f", "e"], 2));
// → ["a", "k", "f", "e"]
```

### Strings and their properties

During the string value chapter, we discussed `length` and `toUpperCase` but if you want to add a new property, it does not stick.

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

All strings value contains a few methods. Some are the `slice` and `indexOf` which have a similar appearance of array methods as well as the name.

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```

A string's `indexOf` can look for a string holding more than one character, while the corresponding array method search for a single element.

```
console.log("one two three".indexOf("ee"));
```

```
// → 11
```

The trim method eradicates whitespace from the beginning to the end of a string.

```
console.log(" okay \n ".trim());
```

```
// → okay
```

The zeroPad function can also be called a padStart, it takes the preferred padding and length character as arguments.

```
console.log(String(6).padStart(3, "0"));
```

```
// → 006
```

You can break a string on every development of another string with split and then join them together with join.

```
let sentence = "Secretarybirds specialize in stomping";
```

```
let words = sentence.split(" ");
```

```
console.log(words);
```

```
// → ["Secretarybirds", "specialize", "in", "stomping"]
```

```
console.log(words.join(". "));
```

```
// → Secretarybirds. specialize. in. stomping
```

You can use the repeat method to repeat a string, which develops a new string holding several copies of the original string attached together.

```
console.log("LA".repeat(3));
```

```
// → LALALA
```

If you want to access individual characters in a string. look below:

```
let string = "ABC";
```

```
console.log(string.length);
```

```
// → 3
```

```
console.log(string[1]);
```

```
// → b
```

Rest parameters

It is important for a function to receive any number of arguments. If you want to write such a function, add three dots before the last parameter of the function.

```
function max (...numbers) {
```

```
let result = -Infinity;
```

```
for (let number of numbers) {
```

```
if (number > result) result = number;
```

```
}
```

```
return result;
```

```
}
```

```
console.log(max(4, 1, 9, -2));
```

```
// → 9
```

When you call a function, the rest of the parameters belongs to an array holding the rest of the arguments. You can also utilize the three dot notation to call functions within an array of arguments.

```
let numbers = [5, 1, 7];
```

```
console.log(max(...numbers));
```

```
// → 7
```

Spread the array out into a function call, pass the element different arguments, like `max (9, ...numbers, 2)`. The Square bracket array notation enables the triple-dot operator bring another array into a new array.

```
let words = ["never", "fully"];
```

```
console.log(["will", ...words, "understand"]);
```

```
// → ["will", "never", "fully", "understand"]
```

### The Math objects

The Math object is utilized as vessel to collate a lot of similar functionality. It produces a namespace to enable values a functions and functions without global bindings. This is the old way to write the constant value name in all caps.

```
function randomPointOnCircle(radius) {
```

```
let angle = Math.random() * 2 * Math.PI;
```

```
return {x: radius * Math.cos(angle),
```

```
y: radius * Math.sin(angle)};
```

```
}
```

```
console.log(randomPointOnCircle(2));
```

```
// → {x: 0.3667, y: 1.966}
```

The math function sends back a new pseudo random number between zero and one whenever you call it.

```
console.log(Math.random());
```

```
// → 0.36993729369714856
```

75

```
console.log(Math.random());
```

```
// → 0.727367032552138
```

```
console.log(Math.random());
```

```
// → 0.40180766698904335
```

When you need a whole random number and not the fractional one, you can utilize the Math.floor on the outcome of the Math.random.

```
console.log(Math.floor(Math.random() * 10));  
// → 2
```

The random number is multiplied by 10 and outputs a number greater than or equal to 0 and below 10. Remember the Math.floor rounds down, the output will be any number from 0 through 9. The math object contains several functions like the Math.ceil, which rounds up to a whole number, and the Math.round, which rounds up to the nearest whole number, and Math.abs, this function takes the entire value of a number.

### Destructuring

Let's use the phi function a little bit.

```
function phi(chair) {  
    return (table[3] * chair[0] - table[2] * chair[1]) /  
        Math.sqrt((chair [2] + chair [3]) *  
        (table[0] + chair [1]) *  
        (table[1] + chair [3]) *  
        (table[0] + chair [2]));  
}
```

One of the few reasons this function is hard to read is because there is a binding directed to the array, but we want bindings for the elements of the array, that is, let n00 = table[0] and so on. Although, there is a better way to get this done in JavaScript.

```
function phi([n00, n01, n10, n11]) {  
    return (n11 * n00 - n10 * n01) /  
        Math.sqrt((n10 + n11) * (n00 + n01) *  
        (n01 + n11) * (n00 + n10));  
}
```

This function also performs brilliantly with the bindings developed by the left, var, or const. If your preferred binding value is an array, utilize the square brackets to check for the value of its binding contents. Use the same concept for objects, utilize the braces and not the square brackets.

```
let {name} = {name: "Faraji", age: 23};  
console.log(name);  
// → Faraji  
JSON
```

JSON represents JavaScript Object Notation and it is utilized as a communication format and data storage for the web. JSON is very similar to JavaScript in its writing style of arrays and object, although there are restrictions. Its property names should be enclosed within double quotes, it allows only simple data expressions, binding etc. JSON do not allow comments.

Here is a sample of a journal entry represented as JSON data:

```
{  
  "squirrel": false,  
  "events": ["work", "touched tree", "pizza", "running"]  
}
```

JavaScript consists of the functions `JSON.stringify` and `JSON.parse` to transform data to and from this format. `JSON.stringify` takes a JavaScript value and sends back a JSON-encoded string while `JSON.parse` takes a string and transforms it to its encoded value.

```
let string = JSON.stringify({squirrel: false,  
  events: ["weekend"]});  
console.log(string);  
// → {"squirrel":false,"events":["weekend"]}  
console.log(JSON.parse(string).events);  
// → ["weekend"]
```

Reversing an array

Arrays consist of the `reverse` method, which is used to change an array by reversing the order that its elements display.

Summary

We covered the objects and arrays in this chapter, and they create ways to categorize various values in a single value. Fancifully, this enables you to insert several associated items in a bag and move about with the bag, rather than enclosing your arms around every singular thing and trying to keep them differently. A lot of values in JavaScript contain properties, the inconsistency being `undefined` and `null`. You can access properties using the `value.prop` or `value["prop"]`. You can use names for objects properties and save a defined set of them. Arrays, as well contain different amounts of similar fanciful values and utilize the numbers (beginning from 0) as the property's names. Arrays contain few sets of properties, which are the `length` and several methods. Methods are functions that reside within properties and

behave on the value they are its property. You can loop across arrays through a unique type of for loop—for (the let element of an array).

### Exercise

Write a JavaScript function to obtain the first item in an array. Setting a parameter 'n' will send back the first 'n' elements of the array.

Data:

```
console.log(array_Clone([1, 2, 4, 0]));
console.log(array_Clone([1, 2, [4, 0]]));
```

[1, 2, 4, 0]

[1, 2, [4, 0]]

Solutions

```
9     });
10
11    console.log(first([7, 9, 0, -2]));
12    console.log(first([],3));
13    console.log(first([7, 9, 0, -2],3));
14    console.log(first([7, 9, 0, -2],6));
15    console.log(first([7, 9, 0, -2],-3));
```

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Get the first element of an array</title>
6 </head>
7 <body>
8 </body>
```

9 | </html>

### JavaScript Code:

```
1 var first = function(array, n) {  
2     if (array == null)  
3         return void 0;  
4     if (n == null)  
5         return array[0];  
6     if (n < 0)  
7         return [];  
8     return array.slice(0, n);
```

# Chapter 6: Higher-Order Functions

A large program takes time to develop and produces a lot of space for bugs to hide, which in turn makes them hard to find. Here are examples of two large programs:

The first is self-contained, and the statement is six lines long.

```
let total = 0, count = 1;
while (count <= 10) {
    total += count;
    count += 1;
}
console.log(total);
```

The second depends on two external functions, and the statement is one line long.

```
console.log(sum(range(1, 10)));
```

Abstraction

When it comes to JavaScript programming language, vocabulary's use the term abstraction. Abstraction is used to hide details and gives the user the capability to talk about problems on a higher level. Let us differentiate two recipes for pea soup. First:

Per person, place 1 cup of dried peas within a container. Put enough water to cover the peas, let it soak for about 12 hours. Remove it from the water and place it in a pot used for cooking with the addition of water four cups of water (4 cups). Let it boil for about two hours, per-person hold half of an onion, slice it using a knife, and pour it into the peas, let it prepare for 10 minutes.

Then the second recipe

Per person: Half an onion, a stalk of celery, 1 cup dried split peas, and a carrot.

Dip the peas into the water for about 12 hours. Boil it for about 2 hours in water (4 cups), attach and slice vegetables. Cook it for 10 minutes.

The computer performs tasks one by one, from blind to high-level concepts.

Abstracting repetition

Plain functions are used to create abstractions. It is habitual for a program to repeat a particular function a specified amount of time. you can achieve that with a loop like this:

83

```
for (let i = 0; i < 10; i++) {
```

```
console.log(i);
}
```

Here we write a function that calls console.log N times.

```
function repeatLog(n) {
for (let I = 0; I < n; I++) {
console.log(I);
}
}
```

If you want to perform a task different from logging in numbers, you can roll action as a function value.

```
function repeat(n, action) {
for (let I = 0; I < n; I++) {
action(i);
}
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

Sometimes you do not need to roll in a predefined function to repeat in a loop, create a function value as an alternative:

```
let labels = [];
repeat(5, i => {
labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

This method layout looks similar to a loop; it defines the type of loop and then produce a body. Although the body is written in function enclosed in parentheses of the call to repeat.

### Higher-order functions

Higher-order functions are functions that work on other functions whether by arguments or by sending them back. This type of function enables user to abstract over actions. For instance, you have a function that can produce functions

```
function greaterThan(n) {
return m => m > n;
```

```
}
```

```
let greaterThan10 = greaterThan(10);
```

```
console.log(greaterThan10(11));
```

```
// → true
```

You can also have functions that can transform other functions.

```
function noisy(f) {
```

```
return (...args) => {
```

```
console.log("calling with", args);
```

```
let result = f(...args);
```

```
console.log("called with", args, ", returned", result);
```

```
return result;
```

```
};
```

```
}
```

```
noisy(Math.min)(3, 2, 1);
```

```
// → calling with [3, 2, 1]
```

```
// → called with [3, 2, 1], returned 1
```

You can as well write functions that produce new set of control flow.

```
function unless(test, then) {
```

```
if (!test) then();
```

```
}
```

85

```
repeat(3, n => {
```

```
unless(n % 2 == 1, () => {
```

```
console.log(n, "is even");
```

```
});
```

```
});
```

```
// → 0 is even
```

```
// → 2 is even
```

This method is a built-in array method, `forEach`, that produce a loop similar to the `for/of` loop as a higher-order function.

```
["A", "B"].forEach(l => console.log(l));
```

```
// → A
```

```
// → B
```

### Filtering arrays

To search for the script in a data set, use the following function. This function helps to filter out element that failed a test in an array.

```
function filter(array, test) {
```

```

let passed = [];
for (let element of array) {
if (test(element)) {
passed.push(element);
}
}
return passed;
}
console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", ...}, ...]

```

This function utilizes the argument called test, a function value, to cover a gap during the calculation. It determines which element to receive.

## Transforming with map

The map method changes an array by setting a function to each and every of its elements and creating a new array from the returned values. The new array consists of the same length with the input array but will map out its content to a renewed for by the function.

```

function map(array, transform) {
let mapped = [];
for (let element of array) {
mapped.push(transform(element));
}
return mapped;
}
let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", ...]

```

Like forEach and filter.

## Summarizing with reduce

When you add up numbers, begin with zero and add the foreach element to the sum. The parameter to reduce are map function combination and begin a value. This function is much easier to understand than the filter and map. It is below:

```

function reduce(array, combine, begin) {
let current = begin;
for (let items of arrays) {
existing = combine(existing, element);
}
}
```

```
}
```

```
return existing;
```

```
}
```

```
console.log(reduce([0,1, 2, 3, 4], (a, b) => a + b, 0));
```

```
// → 10
```

This function is similar to the standard array method reduce and attaches more comfort.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
```

```
// → 10
```

If you want to utilize reduce to search for the script with the more characters, write it like this:

```
function characterCount(script) {
```

```
return script.ranges.reduce((count, [from, to]) => {
```

```
return count + (to - from);
```

```
}, 0);
```

```
}
```

```
console.log(SCRIPTS.reduce((a, b) => {
```

```
return characterCount(a) < characterCount(b)? b: a;
```

```
}));
```

```
// → {name: "Han", ...}
```

### Composability

Take some time to think about how long our list of code would have been without the higher-order functions.

```
let biggest = null;
```

```
for (let script of SCRIPTS) {
```

```
if (biggest == null ||
```

```
characterCount(biggest) < characterCount(script)) {
```

```
biggest = script;
```

```
}
```

```
}
```

```
console.log(biggest);
```

```
// → {name: "Han", ...}
```

You can use the high-order functions when you want to compose operations. For instance, here is a line of code that searches for the average year for living and dead in humans' scripts in the data set.

```
function average(array) {
```

```
return array.reduce((a, b) => a + b) / array.length;
```

```

}

console.log(Math.round(average(
SCRIPTS.filter(s => s.living).map(s => s.year))));  

// → 1165
console.log(Math.round(average(
SCRIPTS.filter(s => !s.living).map(s => s.year))));  

// → 204

```

The block of code displays that the death script is older than the living ones. Let's begin with the entire script, filter out the dead (or living), eradicate their years, average them and complete the result. You can as well write these calculations as a big loop:

```

let total = 0, count = 0;
for (let script of SCRIPTS) {
if (script.living) {
total += script.year;
count += 1;
}
}
console.log(Math.round(total / count));
// → 1165

```

Our two examples are not identical, the first creates new arrays when administering the map and filter while the second calculate numbers only.

### ***Strings and character codes***

One of the significant use of data set is to find out what script a block of text is using. Below is a program that performs that task.

Every script contains an array of character code ranges related with it. So, if you are given a character code, utilize this function to search for the equivalent script (if available):

```

function characterScript(code) {
for (let script of SCRIPTS) {
if (script.ranges.some(([from, to]) => {
return code >= from && code < to;
})) {
return script;
}
}
return null;

```

```
}
```

```
console.log(characterScript(121));
```

```
// → {name: "Latin", ...}
```

Another high-order function is the same method. This method gets a test function and determine if that function sends back true for any component in the array.

There are some operations on JavaScript strings like securing their length using the length property and evaluating their contents through the square brackets.

```
// Two emoji characters, horse and shoe
```

```
let horseShoe = "🐴��";
```

```
console.log(horseShoe.length);
```

```
// → 4
```

```
console.log(horseShoe[0]);
```

```
// → (Invalid half-character)
```

```
console.log(horseShoe.charCodeAt(0));
```

```
// → 55357 (Code of the half-character)
```

```
console.log(horseShoe.codePointAt(0));
```

```
// → 128052 (Actual code for horse emoji)
```

JavaScript's charCodeAt method offers a full Unicode character. This method is used to take characters from a string. Although the argument sent to codePointAt remains an index into the succession of code units. If you use the codePointAt to loop across a string, it produces real characters and not code units.

```
let roseDragon = "🐉🐲";
```

```
for (let char of roseDragon) {
```

```
console.log(char);
```

```
}
```

```
// → 🐉
```

```
// → 🐐
```

### Recognizing text

You have the character Script function and a way to loop across characters correctly, now calculate the characters each script owns. The below counting abstraction is used here:

```
function countBy(items, groupName) {
```

```
let counts = [];
```

```
for (let item of items) {
```

```

let name = groupName(item);
let known = counts.findIndex(c => c.number == name);
if (known == -1) {
  counts.push({number, count: 0});
} else {
  counts[known].count++;
}
}
return counts;
}
console.log(countBy([0,1, 2, 3, 4, 5], n => n > 2));
// → [{number: false, count: 2}, {number: true, count: 3}]

```

The `countBy` function anticipates a collection (a group of numbers, element or anything that you can loop over with `for/of`) and a function that calculates a group name for a specified element. It sends back a list of objects. Each of the objects names a group and determines the number of elements contained in the group.

The method `findIndex` is similar to `indexOf`. This method is used to search for the first value for which the stated function sends back true and returns -1 when zero elements is found. You can use the `countBy` to determine which script is utilized in a block of text.

```

function textScripts(text) {
let scripts = countBy(text, char => {
let script = characterScript(char.codePointAt(0));
return script? script.name : "none";
}).filter(({name}) => name != "none");
let total = scripts.reduce((n, {count}) => n + count, 0);
if (total == 0) return "No scripts found";
return scripts.map(({name, count}) => {
return `${Math.round(count * 100 / total)}% ${name}`;
}).join(", ");
}

```

```

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тьв"'));
// → 61% Han, 22% Latin, 17% Cyrillic

```

The function calculates the characters by name through the use of `characterScript` to attach a name and returning back to the string "none" for characters without any script. If you want to calculate percentages, you need

to determine the total amount of characters belonging to a script that the reduce method can reduce. If it finds zero characters, the function sends back the "none" string.

### Summary

Having the ability to send function values to several functions is an essential aspect of JavaScript. It enables writing functions that imitate computations with “gaps” between them. The code that declares these functions can contain the gaps by giving function values. Arrays provide several important higher-order methods. Use forEach to loop across elements within an array. Use the filter method to send back a new array that has elements that convey the predicate function. Changing an array by setting each item through a function using the map. Use reduce to merge every component of an array to an individual value.

### Exercises

Illustrate a JavaScript function that takes an array of numbers saved and search for the second-lowest and second highest numbers.

### Solution

## JavaScript Code:

```
1 function Second_Greatest_Lowest(arr_num)
2 {
3     arr_num.sort(function(x,y)
4     {
5         return x-y;
6     });
7     var uniqa = [arr_num[0]];
8     var result = [];
9
10    for(var j=1; j < arr_num.length; j++)
11    {
12        if(arr_num[j-1] !== arr_num[j])
13        {
14            uniqa.push(arr_num[j]);
15        }
16    }
17    result.push(uniqa[1],uniqa[uniqa.length-2]);
18    return result.join(',');
19}
20
21 console.log(Second_Greatest_Lowest([1,2,3,4,5]));
```

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <title>Find the second lowest and second greatest numbers from an array</title>
6   </head>
7   <body>
8
9 </body>
```

# Chapter 7: The Secret Life of Objects

When it comes to Programming languages, there is a set technique that utilizes objects as a fundamental idea of program organization, and they are called the object-oriented programming. Its concept will be explained further in this chapter.

## Encapsulation

The primary importance of object-oriented programming is to break programs into little pieces and makes sure each piece manages its section. Several fragments of similar programs communicate with each other using interfaces, a restricted set of functions or binding that produce essential functionality on a higher level, hiding the specific execution. Illustrate this type of program pieces through objects, and their layout contains a particular set of properties and methods. Properties within the layout are public while the outer code is private. Define the accessible layout in comments or documentation. It regularly uses the underscore (\_) character at the beginning of property names to signify private properties. Meanwhile, diving layouts from execution is called encapsulation.

## Methods

Methods are properties that grasp function values.

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`The rabbit says '${line}'`);
};
rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

When you call a function as a method, the binding calls it in its body and direct the object it was called on.

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};
whiteRabbit.speak("Oh my whiskers," + "how late it's getting!");
// → The white rabbit says 'Oh my whiskers, how
// late it's getting!'
```

```
HungryRabbit.speak("I could use a carrot right now.");
// → The hungry rabbit says 'I could use a carrot right now.'
Arrow functions are do not bind their own. You can write the following code:
function normal() {
  console.log(this.coords.map(n => n / this.length));
}
normal.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

## Prototypes

Keep an eye out on this.

```
let empty = {};
console.log(empty.toString());
// → function toString()...{}
console.log(empty.toString());
// → [object Object]
```

A property is pulled out of an empty object. A lot of JavaScript objects consists of a prototype. This is an object used as an alternative source of properties. Therefore, the prototype of that empty object is the inherited prototype, the `Object.prototype`.

```
console.log(Object.getPrototypeOf({}) ==
Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

`Object.getPrototypeOf` sends back the prototype of an object. This prototype produces some method that displays in every object like `toString` that transforms an object to a string execution.

A lot of objects do not contain `Object.prototype` as their prototype but use another object that produces a separate set of default properties;

Functions acquires from `Function.prototype`, and arrays acquires from `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==
Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
Array.prototype);
```

```
// → true
```

This type of prototype object contains a prototype, the `Object.prototype`, so that it can create methods such as `toString` indirectly. Use the `Object.create` to build an object with a particular prototype.

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};

let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!'
```

They `speak(line)` property in an object expression produces a property called `speak` and set a function as its value. It is a shorthand way of describing a method.

The “proto” rabbit is a container for all properties.

### Classes

JavaScript’s prototype system can be described as subsidiary on an object-oriented method called classes. A class describes an object type’s shape, its properties and methods. Prototypes are used to describe properties whereby every instance share similar values such as methods. If you want to build an instance of a set class, create an object that receives from the normal Prototype. Make sure it contains properties that should be included in the instance of this class. Below is the constructor function.

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

JavaScript offers a way to make describing this function effortlessly. Set the keyword at the front of a function call. The prototype object utilized during the construction of objects is attained by fetching the `prototype` property of the constructor function.

```
function Rabbit(type) {
  this.type = type;
}
```

```
Rabbit.prototype.speak = function(line) {  
  console.log(`The ${this.type} rabbit says '${line}'`);  
};  
let weirdRabbit = new Rabbit("weird");  
Function.prototype is the prototype of a constructor. Its prototype property  
grasps the prototype utilized for instances built through it.  
console.log(Object.getPrototypeOf(Rabbit) ==  
Function.prototype);  
// → true  
console.log(Object.getPrototypeOf(weirdRabbit) ==  
Rabbit.prototype);  
// → true
```

### Class notation

JavaScript classes are constructor functions with a prototype property. Below  
is a sample:

```
class Rabbit {  
  constructor(type) {  
    this.type = type;  
  }  
  speak(line) {  
    console.log(`The ${this.type} rabbit says '${line}'`);  
  }  
}
```

```
let killerRabbit = new Rabbit("killer");
```

```
let blackRabbit = new Rabbit("black");
```

The class keyword begins with a class declaration, which enables you to  
describe a constructor and a set of methods in a particular place. Class  
declarations enables methods, properties that grasp functions to be included  
in the prototype. When you use an expression, it does not describe a binding  
but provides the constructor as a value. You can exclude the class name in an  
expression.

```
let object = new class {getWord() { return "hello"; } };  
console.log(object.getWord());  
// → hello
```

### Overriding derived properties

When a property is added to an object, either available in the prototype or  
not, the property adds to the object. If there was an assigned property with

similar names in the prototype, it does not influence the object.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

Overriding properties is used to bestow a standard function and array prototypes a separate `toString` method.

```
console.log(Array.prototype.toString ==
Object.prototype.toString);
// → false
console.log([1, 3].toString());
// → 1,3
```

Calling `toString` on an array bestows a result that is likened to calling `join(",")` on it. It places a comma between the values contained in the array. If you call `Object.prototype.toString` to an array, it provides a separate string.

```
console.log(Object.prototype.toString.call([1, 3]));
// → [object Array]
```

## Maps

A map is a data layout that connect values with other values. If you want to map names to ages, write your code like this:

```
let ages = {
  Boris: 39,
  Liang: 22,
  Júlia: 62
};
console.log(`Julia is ${ages["Julia"]}`);
// → Julia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

Object property names should be in strings. If you want a map that the keys cannot be transformed to strings, do not use object as a map. JavaScript contains a class called map. It saves mapping and enables different types of keys.

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);
console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

The methods get, set, and has are bodies of the layout of the map object. The Object.keys sends back an object's own keys only. You can use the hasOwnProperty method as an option.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

## **Polymorphism**

When a string function is called on an object, the toString method is called on that object in order to build a significant string from it. You can define your own version of toString so that they can build a string that consists of more important data more than the "[object Object]". Below is an illustration:

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
}
console.log(String(blackRabbit));
// → a black rabbit
```

When a line of code is written to function with objects that have a specific layout is called polymorphism. Polymorphic code can function properly with values of several shapes.

## Symbols

Symbols are unique values built with the Symbol function which cannot be built twice.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

The string passed to the symbol is attached when you transform it to a string. They are distinctive and functions as property names.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
  return `${this.length} cm of blue yarn`;
}
console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 cm of blue yarn
```

You can attach the symbol properties in object classes and expressions through the use of square brackets surrounding the property name. Below we refer to a binding storing the symbol.

```
let stringObject = {
  [toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute ropes
```

### The iterator interfaces

The stated object to a for/of loop is iterable. This means that it consists of a named method with the Symbol.iterator symbol when the property is called. The method sends back an object that produces a second layout. It contains a next method that sends back the following result. The result is an object with a value property that produces the next value, Symbol.iterator can be added to several objects. Checkout this layout:

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
```

```

// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}

Let's implement an iterable data structure. We'll build a matrix class, acting
as a two-dimensional array class Matrix {
constructor(width, height, element = (x, y) => undefined) {
this.width = width;
this.height = height;
this.content = [];
for (let y = 0; y < height; y++) {
for (let x = 0; x < width; x++) {
this.content[y * width + x] = element(x, y);
}
}
}
get(x, y) {
return this.content[y * this.width + x];
}
set(x, y, value) {
this.content[y * this.width + x] = value;
}
}

```

The content of the class is saved in an individual array of  $\text{width} \times \text{height}$  elements. These elements are saved row after row. The constructor function gets a width, height, and an additional element function, which is used to set the initial values. Use the get method and set method to recover and update elements within the matrix. Below is the layout of objects with x, y, and value properties.

```

class MatrixIterator {
constructor(matrix) {
this.x = 0;
this.y = 0;
this.matrix = matrix;
}
next() {
if (this.y == this.matrix.height) return {done: true};
let value = {x: this.x,

```

```

y: this.y,
value: this.matrix.get(this.x, this.y)};
this.x++;
if (this.x == this.matrix.width) {
this.x = 0;
this.y++;
}
return {value, done: false};
}
}

```

Below is an illustration of an iterable Matrix class:

```

Matrix.prototype[Symbol.iterator] = function() {
return new MatrixIterator(this);
};

```

We can now loop over a matrix with for/of.

```

let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
console.log(x, y, value);
}
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1

```

## Statics, getters, and setters

Layouts consists of methods, but you can also attach properties that stores values with no function like a Map object containing a size property that determines how many keys are saved in them. An assessed property can hide a method call. Those methods are called getters, and you can describe them by writing "get" at the beginning of the method name in a class declaration or an object expression.

```

let varyingSize = {
get size() {
return Math.floor(Math.random() * 100);
}
};
console.log(varyingSize.size);

```

```

// → 73
console.log(varyingSize.size);
// → 49

```

When you want to use an object's size property, you call the associated method. You can perform a similar task when you a property using a setter.

```

class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }
  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}
let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30

```

The Temperature class enable you to read and write the temperature in degrees Celsius or degrees Fahrenheit, but it saves only Celsius and transform to and from Celsius in the Fahrenheit getter and setter. Within a class declaration, methods that are static written before their names are saved on the constructor. So, you can write Temperature.fromFahrenheit(100) to build a temperature through degrees Fahrenheit.

### Inheritance

Some matrices are symmetric. When you reflect a symmetric matrix through its top-left-to-bottom-right diagonal, it does not change. The value set at x,y remains the same as the y,x. Use the JavaScript's prototype system to build a new class, the prototype for the new class is acquired from the old prototype

but sets a new definition for the method. This is called inheritance in object-oriented programming terms. Below is an illustration:

```
class SymmetricMatrix extends Matrix {  
    constructor(size, element = (x, y) => undefined) {  
        super(size, size, (x, y) => {  
            if (x < y) return element(y, x);  
            else return element(x, y);  
        });  
    }  
    set(x, y, value) {  
        super.set(x, y, value);  
        if (x != y) {  
            super.set(y, x, value);  
        }  
    }  
}  
  
let matrix = new SymmetricMatrix(5, (x, y) => `#${x},${y}`);  
console.log(matrix.get(2, 3));  
// → 3,2
```

### The instanceof operator

It is important to know if an object was gotten from a particular instanceof.

```
console.log(  
    new SymmetricMatrix(2) instanceof SymmetricMatrix);  
// → true  
console.log(new SymmetricMatrix(2) instanceof Matrix);  
// → true  
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);  
// → false  
console.log([1] instanceof Array);  
// → true
```

The operator will scan through the types that were inherited. Therefore, a SymmetricMatrix is an instance of Matrix. You can also apply this method to standard constructors like Array.

### Summary

In this chapter, we realize that objects can perform more than holding their properties. They consist of prototypes that are objects as well. They behave like they contain properties that they do not provide as long as the prototype

has that property. Simple objects use the Object. Prototype to represent their prototype. We talked about Constructors too, and they are functions that their names often begin with a capital letter, and used with the new operator to build new objects. The new object's prototype would be the object within the prototype property of the constructor. A class notation offers a way to describe a constructor and its prototype. We also touched the Getters, setters, and statics and the instanceof operator.

### Exercise

Write a JavaScript program to create an array, through an iterator function and a primary seed value.

### Solution

## HTML Code:

```
1 <html>
2   <head>
3     <title>
4       <meta charset="utf-8">
5       <title>Build an array, using an iterator function and an initial seed value!</title>
6     </head>
7     <body>
8       <h1>
9         </h1>
10    </body>
11  </html>
```

## JavaScript Code:

```
1 //Source: https://bit.ly/2mfFQ
2 const unfold = (fn, seed) => {
3   let result = [],
4     val = [null, seed];
5   while ((val = fn(val[1]))) result.push(val[0]);
6   return result;
7 }
8 var f = n => (n > 50 ? false : [-n, n + 10]);
9 console.log(unfold(f, 12));
```

# Chapter 8: A Robot

In this chapter, there will be work on a robot program, a short program that executes a task in a virtual world. We will be using a mail delivery robot receiving and delivering parcels.

## Meadow field

The Meadow field village is a small one consisting of 14 roads and 11 places. Below is an illustration with an array of roads:

```
const roads = [
  "Alice's House-Bob's House", "Alice's House-Cabin",
  "Alice's House-Post Office", "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop", "Marketplace-Farm",
  "Marketplace-Post Office", "Marketplace-Shop",
  "Marketplace-Town Hall", "Shop-Town Hall"
]
```

The village roads create a graph. A graph is a group of points (village places) with lines dividing them (roads). The graph is the world where our robot walks through. Now let us transform our list to a data layout that each place decides where you can.

```
function buildGraph(borders) {
  let graph = Object.create(null);
  function addBorders(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of borders.map(r => r.split("-"))) {
    addBorders(from, to);
    addBorders(to, from);
  }
  return graph;
}
const roadGraph = buildGraph(roads);
```

Given an array of edges, buildGraph creates a map object that, for each node, saves a connected node of an array. It utilizes the split method to go through the road strings, that contains the form "Start-End", to two-element arrays within the start and end as different strings.

## The task

Our robot will walk across the village. There are different parcels in several places. The robot receives a parcel when it arrives and delivers them when it gets to their destination. When all tasks have all been executed, all parcels must be delivered. If you want to replicate this process, describe a virtual world that can define it. This method reveals the robot location as well as the parcels. Let us bring down the village's state to a specific set of values that describes it.

```
class VillageState {  
    constructor(place, parcels) {  
        this.place = place;  
        this.parcels = parcels;  
    }  
    move(destination) {  
        if (!roadGraph[this.place].includes(destination)) {  
            return this;  
        } else {  
            let parcels = this.parcels.map(p => {  
                if (p.place != this.place) return p;  
                return {place: destination, address: p.address};  
            }).filter(p => p.place != p.address);  
            return new VillageState(destination, parcels);  
        }  
    }  
}
```

The move method is the action center. It examines if a path leads from one place to another, if not the old state is returned because the move is not valid. Then it builds a new state with the destination as the robot's new place as well as building a new set of parcels. The call to map handles the movement and the call to filter handles delivering.

```
let first = new VillageState(  
    "Post Office",  
    [{place: "Post Office", address: "Alice's House"}]
```

```
};

let next = first.move("Alice's House");
console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
Persistent data
```

Unchangeable data layouts are called immutable or persistent. They act like strings and numbers. In JavaScript, you can change almost everything. There is a function named `Object.freeze` that transforms an object so that it disregards writing to its properties.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

## ***Simulation***

A delivery robot takes a good glance at the world and determines what direction it moves to. Therefore, a robot is a function which takes a `VillageState` object and send back the name of a nearby place. The robot sends back an object consisting of its intended direction and a memory value returned the next time you call it.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

The robot can walk through in different directions at every turn. It can run into all parcels and then get to its delivery point. Below is an example:

```

function randomPick(array) {
let choice = Math.floor(Math.random() * array.length);
return array[choice];
}
function randomRobot(state) {
return {direction: randomPick(roadGraph[state.place])};
}

```

If you want to test this unique robot, create a new state with few parcels.

```

VillageState.random = function(parcelCount = 5) {
let parcels = [];
for (let i = 0; i < parcelCount; i++) {
let address = randomPick(Object.keys(roadGraph));
let place;
do {
place = randomPick(Object.keys(roadGraph));
} while (place == address);
parcels.push({place, address});
}
return new VillageState("Post Office", parcels);
};

```

The do loop continues to select new places when it gets a correct address.

Let's start up a virtual world.

```

runRobot(VillageState.random(), randomRobot);
// → Advances to Marketplace
// → Advances to Town Hall
// → ...
// → Run in 63 turns

```

It takes the robot too many turns to transport the parcels because there was no plan ahead.

The mail truck's route

If you search a route that goes through all the places in the village, that could run twice by the robot, but there is a guarantee it will run it. Below is an example:

(starting from the post office):

```

const mailRoute = [
"Alice's House," "Cabin," "Alice's House," "Bob's House,"
"Town Hall," "Daria's House," "Ernie's House,"

```

```
"Grete's House", "Shop", "Grete's House", "Farm",
"Marketplace," "Post Office"
];
```

To use the route, you need to utilize the robot memory. The rest of the robot route is stored in its memory and dispatches the first element at every turn.

```
function routeRobot(state, memory) {
if (memory.length == 0) {
memory = mailRoutes;
}
return {direction: memory[0], memory: memory.slice(01)};
}
```

The robot is faster now. It takes a maximum of 26 turns (twice the 13-step route) but more often less.

## Pathfinding

An interesting approach is to develop routes from the starting point, and inspect every available place that has not been visited until it attains its goal. The below illustration explains that:

```
function findRoute(graph, from, to) {
let work = [{at: from, route: []}];
for (let i = 0; i < work.length; i++) {
let {at, route} = work[i];
for (let location of graph[at]) {
if (place == to) return route.concat(location);
if (!work.some(w => w.at == location)) {
work.push({at: place, route: route.concat(location)}));
}
}
}
}
}
```

The function saves a work list. This is an array of places that will be inspected next, together with the route. It begins with an empty route and the start position. Every location can be touched from every location, a route can also be found between two points therefore the route will not fail.

```
function goalOrientedRobot({location, parcel}, routes) {
if (route.length == 0) {
```

```
let parcel = parcel[0];
if (parcel.location != place) {
  route = findRoute(roadGraph, location, parcel.place);
} else {
  route = findRoute(roadGraph, location, parcel.address);
}
}

return {direction: routes[0], memory: route.slice(1)};
}
```

### Summary

In this chapter we touched on robots using JavaScript. We got to the Meadow field village where we talked about a village with 11 places and 14 roads, then we moved to the Persistent data, whereby Data structures behave like strings and do not change. Simulation enables you to pass memory to robots and let them to send back a new memory.

### Exercise

Write a straightforward JavaScript program to connect every element in the below array into a string.

```
"Red, Green, White, Black"
"Red, Green, White, Black"
"Red+Green+White+Black"
```

## Solution

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset=utf-8 />
5 <title>JavaScript Array Join</title>
6 </head>
7 <body>
8 </body>
9 </html>
```

### JavaScript Code:

```
1 myColor = ["Red", "Green", "White", "Black"];
2 console.log(myColor.toString());
3 console.log(myColor.join());
4 console.log(myColor.join('+'));
```

### Sample Output:

```
Red,Green,White,Black
Red,Green,White,Black
Red+Green+White+Black
```



# Chapter 9: Bugs and Errors

Computer program flaws are called bugs, the mistakes made by a computer program.

## Strict mode

You can utilize JavaScript in a strict mode. This can be achieved by adding the string "use strict" right at the top of a file. Below is an illustration:

```
function canYouSpotTheProblem() {  
    "use strict";  
    for (counter = 0; counter < 10; counter++) {  
        console.log("Happy happy");  
    }  
}  
  
canYouSpotTheProblem();  
// → ReferenceError: counter is not defined
```

Use the below code when you want to call a constructor function omitting the new keyword so it doesn't introduce a newly created object:

```
function Person(name) {this.name = name;}  
let Ferdinand = Person("Ferdinand"); // oops  
console.log(name);  
// → Ferdinand
```

The fake call to Person flourished but sends back an undefined value and built the global binding name. In strict mode, use "use strict";

```
function Person(name) {this.name = name;}  
let Ferdinand = Person("Ferdinand"); // forgot new  
// → TypeError: Cannot set property 'name' of undefined
```

This quickly indicates that we have a problem. That's good. Constructors built with the class notation regularly complains when you call them without the new. Strict mode does not allow setting function multiple characters with similar name and eradicates some language problems.

## Testing

You can use this function to search for mistakes in a program. It is achieved by running over and over again. The computer is excellent in repetitive tasks. This process is called Automated testing. It entails writing a program that tests another program. Tests regularly go through small labeled programs that determine some part of your code. For instance, let us create a set of tests for the toUpperCase method. Below is an illustration:

```

function test(label, body) {
if (!body()) console.log(`Failed: ${label}`);
}
test("convert Latin text to uppercase", () => {
return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
return "Χαίρετε".toUpperCase() == "XAIPETE";
});
test("don't convert case-less characters", () => {
return "ΛΟΥΛΟΥ".toUpperCase() == "ΛΟΥΛΟΥ";
131
});

```

## **Debugging**

When a program displays an error, the next thing is to determine the problem and get it fixed. The error message will point directly at a particular line in your block of code. Take a look at the error description, and you will find the problematic line. This example program tries to transform a whole number into a string within set base (decimal, binary, and so on) by continuously selecting out the last digit and dividing the number to eradicate the digit.

```

function numberToString(n, base = 10) {
let result = "", sign = "";
if (n < 0) {
sign = "-";
n = -n;
}
do {
result = String(n % base) + result;
n /= base;
} while (n > 0);
return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e...-3181.3

```

Adding some `console.log` calls into the program is an excellent path to take if you want to acquire additional information about the execution. We want n

to get the values 13, 1, and then 0. We should state the value at the beginning of the loop'

13  
1.3  
0.13  
0.013...  
1.5e-323

Dividing 13 by will not provide a whole number. Instead of  $n \neq \text{base}$ , what we want is  $n = \text{Math.floor}(n / \text{base})$  so that the number can be “shifted” to the right.

Another great way to specify a breakpoint is to insert a debugger statement (keyword) into your program. If the developer tools will pick it up and pause the program anytime it gets to that statement if your browser is active.

## Exceptions

When a function is cannot go further, the exception handling is the place that understands how to fix that problem. Exceptions are a tool that enables a block of code that runs into difficulties to throw an exception, and an exception could be any value. You can intentionally create problems within your line of code to catch the exception during its zooming down the process so that you can use it to solve the problem and continue your work.

Here's an example:

```
function promptDirection(question) {  
let result = prompt(question);  
if (result.toLowerCase() == "left") return "L";  
if (result.toLowerCase() == "right") return "R";  
throw new Error("Invalid direction: " + result);  
}  
function look() {  
if (promptDirection("Which way?") == "L") {  
return "a house";  
} else {  
return "two angry bears";  
}  
135  
}  
}  
try {
```

```
console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

Raise an exception by using the throw keyword.

Cleaning up after exceptions

An exception's effect is a different type of control flow. This indicates that your code can have different side effects, but an exception could stop them from happening. Below is some bad banking code:

```
const accounts = {
  a: 100,
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!accounts.hasOwnProperty(accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

The transfer function is used to send a particular sum of money from a stated account to another, as well as demanding for the name of the other account. If you input a wrong account name, getAccount outputs an exception. If the money has been sent from the first account and then the program outputs an exception before it gets the money transferred into the other account, the money will disappear. Solve this problem, utilize the finally block code. This is illustrated below:

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
```

```
accounts[from] -= amount;
progress = 1;
accounts[getAccount()] += amount;
progress = 2;
} finally {
if (progress == 1) {
accounts[from] += amount;
}
}
}
```

### Selective catching

When an exception makes it to the bottom of a block of code without errors, it is handled by the environment. Invalid uses such as inspecting up a property on null, citing a nonexistent binding, or calling a non-function will result in raised exceptions. These exceptions can be caught. JavaScript does not support selective catching of exceptions. Below is an example that tries to continue calling promptDirection until it provides a valid answer:

```
for (;;) {
try {
let dir = promptDirection("Where?"); // ← typo!
console.log("You select ", dir);
break;
} catch (e) {
console.log("Invalid direction. Try again.");
}
}
```

The for (;;) construct is used to build a loop that cannot be terminated on its own. You can break out of the loop when you have a valid direction, but a misspelled promptDirection will output an undefined variable error. If you want to catch a particular type of exception, check in the catch block if the exception you have is the same one you want and rethrow it. Let's describe a new kind of error and utilize instanceof to recognize it.

```
class InputError extends Error {}
function promptDirection(question) {
let result = prompt(question);
if (result.toLowerCase() == "left") return "L";
if (result.toLowerCase() == "right") return "R";
```

```
throw new InputError("Invalid direction: " + result);
}
```

The new error class prolongs error. It does not describe its own constructor, it inherits the Error constructor, which anticipates a string message that we can recognize it with.

Now the loop can catch these more carefully.

```
for (;;) {
try {
let dir = promptDirection("Where?");
console.log("You chose ", dir);
break;
} catch (e) {
if (e instanceof InputError) {
console.log("Invalid direction. Try again.");
} else {
throw e;
}
}
}
}
```

## Summary

This chapter deals with debugging. An essential part of programming is identifying and fixing bugs. Sometimes, you can solve problems locally. You can track them with special return values or the use of exceptions.

## Exercise

Fix the below broken code and indicate the problem:

```
for(var I=0; I > 5; I++){
    console.log(i)
}
```

## Solution

```
for(var i=0; i < 5; i++){
    console.log(i)
}
```

Problem – Incorrect condition within the loop.

# Chapter 10: Regular Expressions

Programming concepts and tools advances in a disorganized way, but its success indicates an excellent piece of technology. In this chapter, we will discuss regular expressions. They are used to define patterns within a string data. They create a small, different language that consists of JavaScript as well as other languages.

## Creating a regular expression

A regular expression is constructed with the RegExp constructor or written as a strict value by surrounding a pattern in forward-slash (/) characters.

```
let re1 = new RegExp("ABC");
```

```
let re2 = /ABC/;
```

These two regular expression objects stand for the same pattern. When utilizing the RegExp constructor, write the pattern as a normal string so that the rules apply for backslashes.

The second notation influence backslashes differently, first put a backslash before any forward slash because the pattern displays between slash characters.

## Testing for matches

Regular expression objects contain several methods, one of which is the test. When you pass a string, it returns a Boolean determining if a string contains a match of the pattern in the expression.

```
console.log(/abc/.test("abcde"));
```

```
// → true
```

```
console.log(/abc/.test("abxde"));
```

```
// → false
```

A regular expression contains only non-special characters that represent the succession of characters. If ABC papers anywhere in the string, we are running a test, the test will return true.

## Sets of characters

Regular expressions enable us to indicate difficult patterns. If you want to equal any number in a regular expression, place a set of characters in-between square brackets. The following expressions equal every string within a digit:

```
console.log(/[0123456789]/.test("in 1992"));
```

```
// → true
```

```
console.log(/[0-9]/.test("in 1992"));
```

```
// → true
```

A hyphen (-) between two characters inside square brackets is used to specify a scope of characters that the character's Unicode numbers decide the command. Characters from 0 to 9 stay next to each other in this command (codes 48 to 57), so [0-9] wrapping them all and equals any digit. Common character groups contain individual shortcuts built-in.

Digits are part of them: \d means the exact thing as [0-9].

\d Any character digit

\w an alphanumeric character (“word character”)

\s Any whitespace character (space, tabs, newline, and similar)

\D A character which is not a digit

\W A nonalphanumeric character

\S A nonwhitespace character

. Any character without for newline

So, match a date and time format like 01-30-2003 15:20 with the expression below:

```
let dateTime = /\d\d-\d\d-\d\d\d\d\d\d\d\d:/\d\d\d\d/;  
console.log(dateTime.test("01-30-2003 15:20"));  
// → true  
console.log(dateTime.test("30-jan-2003 15:20"));  
// → false
```

These backslash codes are also utilized within the square brackets. For instance, [\d.] indicates any digit or periodic character but the period has no meaning as well as other unique characters, like +.

```
let notBinary = /[^\d]/;  
console.log(notBinary.test("1100100010100110"));  
// → false  
console.log(notBinary.test("1100100010200110"));  
// → true
```

Repeating parts of a pattern

Now you have a perfect understanding of how to equal single digits. If you want to match a whole number or a succession of more digits, place a plus sign (+) immediately after an input in a regular expression. This signifies that the element could repeat more than once, \d+/ equals one or more-digit characters.

```
console.log(/\d+/.test("123"));  
// → true
```

```
console.log(/\d/.test(""));  
// → false  
console.log(/\d*/.test("123"));  
// → true  
console.log(/\d*/.test(""));  
// → true
```

The star (\*) enables the pattern to equal zero. In the below example, the u character is permitted to rise, but the pattern is also equal when it is not found.

```
let neighbor = /neighbour/;  
console.log(neighbor.test("neighbour"));  
// → true  
console.log(neighbor.test("neighbor"));  
// → true
```

To determine if a pattern should execute a certain number of times, utilize the braces, inserting {4} after an element. If you want it to execute precisely four times, define a range like this:

{2,4} signifies that the element must execute twice and at most four times. Below is another pattern of the date and time pattern, which enables both single and double-digit days, months and hours.

```
let dateTime = /\d{01,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;  
console.log(dateTime.test("01-30-2003 8:45"));  
// → true
```

### Grouping subexpressions

If you want to Utilize an operator like \* or + on one or more element concurrently, use the parentheses. Some parts of a regular expression, which is cited in parentheses, counts as an individual element as long as the operators are concerned.

```
let cartoonCrying = /boo+(hoo+)+/i;  
console.log(cartoonCrying.test("Boohooooohoohooo"));  
// → true
```

The first and second + characters only apply to the second in boo and hoo. The third + indicates to the entire group (hoo+), equaling one or more successions like that. The I ending the first expression in the example is used for case insensitive, enabling it to equal the uppercase B in the input string. Although the pattern is entirely lowercase.

### Matches and groups

The test method is the best way to equal a regular expression, it determines if it matches only. Regular expressions contain an exec (execute) method, which will output null if it could find a match and sends back an object containing data about the match.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

A sent back object from exec contains an index property, which determine what part of the string the thriving match starts. Below is the succession of digits needed:

String values contain a match method that behaves similarly.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

When the regular expression holds subexpressions collated with parentheses, the matching text will display in the array. The whole match is the first, the next is the matched part by the first group, and then the following group, etc.

```
let quotedText = '/([^\']*)';
console.log(quotedText.exec("she said 'hello'"));
// → ["hello'", "hello"]
```

When a group was not finish being matched (if it contains a question mark), it outputs undefined. When you match a group several times, only the last match finishes in the array.

```
console.log(/bad(l)y?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

### The Date classes

JavaScript consists of a standard class that represent dates; it is called Date. If you want to build a date object utilizing new, first get the current time and date.

```
console.log(new Date());
// → Mon Nov 11 2018 16:19:11 GMT+0100 (CET)
```

You can as well Build an object for a particular time.

```
console.log(new Date(2008, 11, 8));
```

```
// → Wed Dec 08 2008 00:00:00 GMT+0100 (CET)
console.log(new Date(2008, 11, 9, 12, 59, 59, 999));
// → Wed Dec 08 2009 12:59:59 GMT+0100 (CET)
```

You should follow the JavaScript date naming convention where month numbers begin at zero (therefore, December is 11), and day numbers begin at 1.

The hours, minutes, seconds, and milliseconds arguments are voluntary and recognized as zero when not specified. Timestamps are saved as the number of milliseconds in the UTC time zone. Following by a convention set by “Unix time”.

```
console.log(new Date(2016, 11, 14).getTimes());
// → 1445677600000
console.log(new Date(1445677600000));
// → Mon Dec 11 2008 00:01:00 GMT+0100 (CET)
```

When the Date constructor is set a single argument, the argument is recognized as a millisecond count. Determine the current millisecond count by building a new Date object and call getTime on it or use the Date.now function. Date objects produce methods such like getDate, getHours, getFullYear, getMonth, getMinutes, and getSeconds to retrieve their components.

```
function getDate(strings) {
let [, month, day, year] =
/(\d{01,2})-(\d{01,2})-(\d{4})/.exec(strings);
return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thursday Jan 30 2003 00:00:00 GMT+0100 (CET)
```

## **String and Word boundaries**

You can ensure that the match spans the entire string by adding the markers ^ and \$. The caret equals the beginning of the input string, while the dollar sign equals the end. Therefore, /<sup>^</sup>\d+\$/ matches a string continuing multiple or one digits, /<sup>^</sup>! / matches any string that begins with an exclamation mark, and /x<sup>^</sup>/ matches no string. If you want the date to begin and end on a boundary word, utilize the marker \b. A word boundary can begin or finish strings containing a word character (\w) on one side and a nonword character on the other.

```
console.log(/cat/.test("concatenate"));
```

```
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

### Choice patterns

If you want to know if chunk of text consists of a number and followed by words such as chicken, pig, or cow, let's write three regular expressions and test them. We utilize the pipe character (|) to mark a choice between the pattern to the left and right:

```
let animalCount = /\b\d+ (pig|cows|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
150
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

### The replace method

String values contains a replace method, which is used to restore part of the string with another one.

```
console.log("papa".replace("p", "m"));
// → mama
```

The first argument can equally be a regular expression. It replaces the first match of the regular expression. If you add a g option (for global) to the expression, every match contained in the string is replaced, and not first only.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

The major importance of utilizing regular expressions with replace is because it refers to matched groups in the replacement string. For instance, you have a large string consisting of people's name, one name per line, using the format Lastname, Firstname. You can as well change and remove these names, eradicate the comma to get a Firstname Lastname format, and utilize the following code:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
  .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
```

```
// John McCarthy  
// Philip Oaks
```

The \$1 and \$2 contained in the replacement string refer to the group in parentheses in pattern. \$1 is restored by the text, which matches opposing the first group, \$2 up to \$9. Each and every match can be referred to with \$&. You can pass a function instead of a string as the second argument to restore. For each replacement, the function is called with the matched group as arguments, and include the return value into the new string. Below is an example:

```
let s = "the cia and fbi";  
console.log(s.replace(/\b(fbi|cia)\b/g,  
str => str.toUpperCase()));  
// → the CIA and FBI
```

Here's a more interesting one:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";  
function minusOne(match, amount, units) {  
amount = Number(amount) - 1;  
if (amount == 1) {// Just one left, remove the 's'  
unit = unit.slice(0, unit.length - 1);  
} else if (amount == 0) {  
amount = "no";  
}  
return amount + " " + units;  
}  
console.log(stock.replace(/(\d+) (\w+)/g, minusOne));  
// → no lemon, 1 cabbage, and 100 eggs
```

Greed

You can use replace to write a function that eradicates all comments from a block of JavaScript code. Look below:

```
function stripComments(code) {  
return code.replace(/\//.*|\/*[^]*\*/g, "");  
}  
console.log(stripComments("1 + /* 2 */3"));  
// → 1 + 3  
155  
console.log(stripComments("x = 10;// ten!"));  
// → x = 10;
```

```
console.log(stripComments("1 /* a *//* b */ 1"));
// → 1 1
```

The section before the or operator matches two slash characters accompanied with any number that is not a newline character. Use [^] to match any character.

Repetitive Operators (+, \*, and {}) are called greedy which means they match and they can also backtrack. If a question mark is placed after them (+?, \*?, ??, {}?), the greed disappears and begin matching no matter how small. The smallest stretch of characters which brings a \*/, is the star match. It absorbs one block comment only.

```
function stripComments(code) {
  return code.replace(/\\/.*|\\*[^\n]*?\\*/g, "");
}
console.log(stripComments("1 /* a *//* b */ 1"));
// → 1 + 1
```

## Dynamically creating RegExp objects

There are few instances whereby you would not understand the specific pattern you should match against when your code is being written. If you want to search for the user name in a chunk of text and wrap it in underscore characters to make it unique. But you can create a string and utilize the RegExp constructor. Below is an example:

```
let name = "harry";
let text = "Harry is a suspicious character.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Harry_ is a suspicious character.
```

If you want to build the \b boundary markers, utilize the two backslashes because you would write them in a normal string. The second argument to the RegExp constructor consists of the options needed for the regular expression. The "gi" represent global and case insensitive. Below is an example:

```
let name = "dea+hl[]rd";
let text = "This dea+hl[]rd guy is super annoying.";
let escaped = name.replace(/\[\.+*\?(){}|^$/g, "\\$&");
let regexp = new RegExp("\\b" + escaped + "\\b", "gi");
console.log(text.replace(regexp, "_$&_"));
```

```
// → This _dea+hl[]rd_ guy is very annoying.
```

### The search method

You cannot call the `indexOf` method on strings a regular expression. You can also use another method, `search` whereby you can call the `indexOf` method a regular expression, and it sends back the first index where it sees the expression and when it does not see it returns -1.

```
console.log(" word".search(/\S/));
```

```
// → 2
```

```
console.log(" ".search(/\S/));
```

```
// → -1
```

### The lastIndex property

The `exec` method does not produce a better way to begin the search from a stated position in the string. Regular expression objects contain properties. One of the properties is called `source`, and it consists of the string that built the expression. `lastIndex` is another property, and it influences some small circumstances, where the next match will begin. The regular expression must contain the global (`g`) or sticky (`y`) option, and the match will occur using the `exec` method. Illustration below:

```
let pattern = /y/g;
```

```
pattern.lastIndex = 3;
```

```
let match = pattern.exec("xyzzy");
```

```
console.log(match.index);
```

```
// → 4
```

```
console.log(pattern.lastIndex);
```

```
// → 5
```

The dissimilarity between the global and the sticky options is that, when you enable sticky, the match succeeds if it begins at `lastIndex`, while with global, it will find a position for the match can begin.

```
let global = /ABC/g;
```

```
console.log(global.exec("XYZ ABC"));
```

```
// → ["ABC"]
```

```
let sticky = /ABC/y;
```

```
console.log(sticky.exec("XYZ ABC"));
```

```
// → null
```

If you are utilizing a shared regular expression value for several `exec` calls, problems will occur with the automatic updates to the `lastIndex` property.

```
let digit = /\d/g;
```

```
console.log(digit.exec("it is here : 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

The global option also transforms the concept the match method on strings works.

### Looping over matches

The best thing to do is to scan across all events of a pattern in a string giving access to the match object in the loop body. Use the lastIndex and exec to achieve this.

```
Let input = "A string with three numbers in it... 42 and 88.";
let number = /\b\d+\b/g;
let match;
while (match = number.exec(input)) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
// Found 42 at 33
// Found 88 at 40
```

The assignment expression value (=) is the specified value. So utilizing match = number.Exec (input) as the condition within the while statement, the match is we execute the match at the beginning of every iteration, store the result within a binding, and stop looping when matches can no longer find.

### Parsing an INI file

```
Searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7
; comments are preceded by a semicolon
; each section concerns an individual enemy
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.google.com/google/
[davaeorn]
fullname=Davaeorn
type=evil witch
outputdir=/home/margin/enemy/davaeorn
```

The rules for this format (which is a generally used format, usually called an INI file) is:

- Ignore blank lines and lines beginning with semicolons.
- Lines enclosed in [and] begin a new section.
- Lines consisting of an alphanumeric identifier accompanied by a = character attach a setting to the current section.
- Everything else is invalid.

Our mission here is to change a string like this to an object, which properties use string to store settings that are written before the subobjects for sections and the first section header. The subobjects hold that section's settings. Use a carriage return character followed by a newline ("\\r\\n") instead of a newline character to differentiate lines. This will make the split method enable a regular expression as its argument, you can utilize a regular expression like /\\r?\\n/ to divide it in a way that enables both "\\n" and "\\r\\n" between lines.

```
function parseINI(string) {  
    // Begin with an object to hold the top-level fields  
    let result = {};  
    let section = result;  
    string.split(/\\r?\\n/).forEach(line => {  
        let match;  
        if (match = line.match(/^([\\w]+)=(.*$/)) {  
            section[match[1]] = match[2];  
        } else if (match = line.match(/^\\[(.*]\\]$))/) {  
            section = result[match[1]] = {};  
        } else if (! /^\\s*(;.*?\\$)/.test(line)) {  
            throw new Error("Line '" + line + "' Invalid.");  
        }  
    });  
    return result;  
}  
  
console.log(parseINI(`  
name=Vasilis  
[address]  
city=Tessaloniki`));  
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

The code advances over the file lines and create up an object. Properties at the top are saved straight into that object, whereby properties can be found in sections are saved in a different section object. The section binding points to the object for the existing section. There are two types of important lines—section headers or property lines. If a line is always property, it is saved in the existing section. If it is a section header, build a new section object, and set section to it.

The pattern if (match = string.match()) is identical to the trick of utilizing an assignment as the condition for a while.

### International characters

Because of JavaScript's initial simplistic implementation and that this approach set in stone as standard behavior, JavaScript's regular expressions are somewhat dumb about characters that don't show in the English language. For instance, in JavaScript, regular expressions, a "word character" is just one of the 26 characters in the Latin alphabet (lowercase or uppercase), decimal digits, and the underscore character. Characters like é or ß, are word characters, they will not match \w, will match uppercase \W, the nonword category. It indicates that characters composed of two code units.

```
console.log(/\u{3}/.test("𠮷𠮷𠮷"));
// → false
console.log(/<.>/.test("<\u>"));
// → false
console.log(/<.>/u.test("<\u>"));
// → true
```

The identified problem here is that \u in the first line is being managed as two code units, and the {3} section is used to the second one only. The dot also matches a single code unit. Add u option (for Unicode) to your regular expression so that it manages that characters properly. The wrong behavior remain set as the default, unfortunately, because transforming that could create problems for current code that depends on it.

You can use \p in a regular expression to match every character that Unicode standard gives a set property.

```
console.log(\p{Script=Greek}/u.test("α"));
// → true
console.log(\p{Script=Arabic}/u.test("α"));
// → false
```

```
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode describes a number of important properties, although searching for the one that you need may not always be trivial.

### Summary

We focused on regular expressions in this chapter. An object that stands for patterns in strings is a regular expression. They consist of their language and utilize it for the exhibition of these patterns.

/ABC/ A succession of characters  
/[abc]/ Character from a range of characters  
/[^abc]/ Any character absent in a range of characters  
/ [0-9]/ Any character within a set of characters  
/x+/ One or more events of the pattern x  
/x+? / One or more events, nongreedy  
/x\*/ Zero or more events  
/x? / Zero or one events  
/x {2,4}/ Two to four events  
/(ABC)/ A group  
/a|b|c/ Any one of various patterns  
/\d/ Any digit character  
/\w/ An word character  
/\s/ Any whitespace character  
/. / Any character excluding newlines  
/\b/ A word boundary  
/^/ Beginning of input  
/\$/ End of input

A regular expression consists of a method test to check if a set string matches it. It also contains an exec method, whereby when it finds a match, it sends back an array having every matched group. This type of array contains an index property that specifies where the match started. Strings use a matching method to match them in opposition to a regular expression as well as a search method to find one, sending back only the beginning position of the match.

### Exercise

Write a JavaScript program working as a trim function (string) utilizing regular expression.

## Solution

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset=utf-8 />
5 <title>Trim function using regular expression</title>
6 </head>
7 <body>
8 </body>
9 </html>
```

## JavaScript Code:

```
1 function Trim(str)
2 {
```

```
3 var result;
4 if (typeof str === 'string')
5 {
6     result = str.replace(/^\s+|\s+$/g, '');
7     return result;
8 }
9 else
10 {
11     return false;
12 }
13 }
14 console.log(Trim(' w3resource '));
```

# **Chapter 11: Modules**

This particular program has a simple layout. It is very to explain its concept, and every part plays a precise role. The organizing and maintaining of the layout can be much more work, but it eventually pays off the next time someone uses the program in the future. Therefore, you can be intrigued to neglect it and enable the program parts to become intermixed, which will create two practical issues. First, when everything touches everything else, it is hard to look at any set piece in separation. You will have to create a comprehensive understanding of the full program. Second, using any of the functionality from the program in different situations, you can rewrite it instead of disengaging it from the context.

## **Modules**

A Module is a type of program that determines which pieces it should depend on and what functionality should it produce for other pieces to utilize. By putting a limit on the ways modules communicate with one another, the system is identical to LEGO, where pieces communicate using straightforward connectors, and not like mud where the entire pieces mix with everything. Dependencies are the relationships between modules. When a module wants a piece from another module, it depends on that module. The module itself illustrates this fact, and it is used to determine which other modules should be available to use a stated module and to load dependencies automatically. Every module needs its scope to separate itself from other modules.

## **Packages**

One significant benefit of building a program out of different pieces, and being able to launch those pieces individually, is that you can apply the same piece in separate programs. When the duplication of code begins, the package comes in to play. A package is a block of code that can be shared. It may consist of one or more modules and holds information about the other depending packages. Packages also contain documentation describing its functions so that people who didn't write it can utilize it. When a problem occurs in a package, or you add a new feature, the package updates automatically. Now programs depending on its upgrades to the latest version. Working like this, you need infrastructure. A place to save and search for packages and an easy way to upgrade and install them. When it comes to JavaScript, NPM provides support.

Divide NPM into two things: an online service where you can upload and download packages and a program that handles the installation and management. Almost every code on NPM is licensed.

### Improvised modules

Before the year 2015, the JavaScript language did not contain any built-in module system, but people have been creating large systems using JavaScript for over a decade now, and they required modules. They developed their module systems and integrated them into the language. You can utilize JavaScript functions to build local scopes and objects to stand for module layouts. Use this module to map between day names and numbers. Its layout contains the weekDay. Name and weekDay.

.number, and it hides its local binding names inside the scope of a function  
The expression is called upon immediately.

```
const weekDay = function() {  
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",  
    "Thursday", "Friday", "Saturday"];  
  return {  
    name(number) {return names[number];},  
    number(name) {return names.indexOf(name);}  
  };  
}();  
console.log(weekDay.name(weekDay.number("Sunday")));  
// → Sunday
```

This style of modules does not proclaim dependencies. They place the layout into the global scope. This style of modules provides isolation to a certain degree, but it does not declare dependencies. Instead, it just puts its interface into the global scope and awaits its dependencies, if there is any, to do the same.

### **Evaluating data as code**

There are different types of ways to take data and launch it as part of the existing program. The best way is to utilize the unique operator eval, which affects a string in the existing scope.

```
const x = 1;  
function evalAndReturnX(code) {  
  eval(code);  
  return x;  
}
```

```
console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

A less complicated way to explain data as code is using the Function constructor. It takes two types of argument, a string with a comma, divided list of argument names and a string with the body function. It encloses the code in a function value to enable it get its own scope.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

You can enclose the module's code in a function and utilize that same function scope as a module scope.

### CommonJS

The common approach used to gobble on JavaScript modules is the CommonJS modules. Node.js uses this approach as well as a lot of packages on NPM. The major idea in CommonJS modules is the function named require. If you call this together with the dependency's module name, it loads the module and sends back its interface. The loader covers the module code within a function, and modules have their local scope automatically. You can call require to see their dependencies and place their layout in the object that exports them. This example module gives a date-formatting function. It utilizes two packages from NPM, the ordinal to transform numbers to strings like "1st" and "2nd", and date-names to obtain the English names for months and weekdays. It exports a single function, formatDate, that takes a template string and a Date object. The template string could contain codes that control the format, such as YYYY (the full year) and Do for the ordinal day of the month. You can set a string to it like "MMMM Do YYYY" to get output like "December 2nd, 2013".

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");
exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
```

```
if (tag == "Do") return ordinal(date.getDate());
if (tag == "dddd") return days[date.getDay()];
});
};
```

The layout of the ordinal is a single function, whereby the date-names exports an object that consists of several other things, months and days are arrays of names. Restructuring is quite easy when building bindings for imported layouts. The module attaches its layout function to exports so that modules that rely on it have access to it. Utilize the module this way:

```
const {formatDate} = require("./format-date");
console.log(formatDate(new Date(2017, 9, 13),
"dddd the Do"));
// → Friday the 13th
```

We can define require, in its most minimal form, like this:

```
require.cache = Object.create(null);
function require(name) {
if (! (name in require.cache)) {
let code = readFile(name);
let module = {exports: {}};
require.cache[name] = module;
let wrapper = Function("require, exports, module", code);
wrapper(require, module.exports, module);
}
return require.cache[name].exports;
}
```

In this code, readFile is a powered function that reads a file and sends back its contents as a string. Standard JavaScript does not offer such functionality, but several other JavaScript environments such as Node.js and the browser offers their unique ways of accessing files. The above example bluffs that readFile exists. To keep away from loading the same module several times, you need a store (cache) of already loaded modules. If called, it examines if the demanded module loads and, if not, it loads it. It requires reading the module's code, enclosing it within a function and calling it. The layout of the ordinal package from earlier is a function and not an object. The CommonJS modules create an empty layout object using the module system for you to export, restore that with values by overwriting module.exports. Modules do this to export a single value instead of a layout object. By describing exports,

require and module as parameters for the created enclosing function (and setting the right values while calling it), the loader ensures that these bindings are accessible within the module's scope. Strings set to require is interpreted to a filename or web address vary in separate systems. When it starts with ". /" or. "/" it is usually translated as relative to the existing module's filename. So ". / format-date" will be the file called format-date.js in the same directory.

If the name is not relative, Node.js will search for a package that is installed by that name.

Now, instead of writing your own INI file parser, you can utilize one from NPM.

```
const {parse} = require("ini");
console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

## ECMAScript modules

CommonJS modules perform well while combining with NPM, and they enable the JavaScript community to begin codesharing on a substantial scale. The notation is quite weird. The things added to exports are not accessible within the local scope. Without running the code of a module, it will be tough to determine its dependencies before taking any argument. That is the reason JavaScript standard establishes its own, separate module system. It is named called ES modules, where ES represents ECMAScript. The initial idea of dependencies and interfaces stay the same, but their details are different. The notation integrates into the language. You can now use a unique import keyword to access a dependency instead of calling a function.

```
import ordinal from "ordinal";
import {days, months} from "date-names";
export function formatDate(date, format) { /* ... */ }
```

Use the export keyword to export things. It could display at the front of a class, function or binding definition (let, const, or var). An ES module's layout does not represent a single value but a set of named bindings. The previous module attaches formatDate to a function. If you import from a different module, the binding comes with it, and not the value, which means the value of an exporting module can change the binding any time, and the importing modules will recognize its new value. If a binding is named default, recognizes it as the module's major exported value. If a module like

ordinal imports in the example, omitting the braces across the binding name, you will find its default binding. To develop a default export, write `export default` ahead of expression, a class or a function declaration.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

You can rename imported bindings using the word like this.

```
import {days as dayNames} from "date-names";
console.log(dayNames.length);
// → 7
```

Another major difference is that the ES module imports occur before a module's script begins to run. Import declarations may not display within functions or blocks, and the dependent names should be quoted strings and not arbitrary expressions. A lot of projects are written through ES modules and then changed to some other format when it is published. During a transitional period, use two separate module systems side by side.

### Module design

Program structuring is one of the nice features of programming. Any unknown functionality can model in different ways. The good program design is a personalized one, and there is a matter of preference and taste. To understand the value of well-organized design, you need to read and work on several programs and take note of performing and non-performing functions. The module design is straightforward to use. The `ini` package module follows the standard JSON object by supplying `stringify` and `parse` (to write an INI file) functions, and, like JSON, changes between plain objects and strings. A lot of the INI-file parsing modules on NPM supply a function that reads that type of file from the hard disk and parses it. Concentrated modules that calculate values are suitable in a larger range of programs than larger modules that execute complex actions that have side effects. An INI file reader that want to read the file from disk is not useful in cases whereby the file's content comes from another source. Similarly, stateful objects are occasionally necessary and useful, but if you can use a function, utilize it. Various INI file readers on NPM gives a layout style that needs you to build an object, then load the file into the object, and use specific methods to achieve the desired results. It is the object-oriented style and tradition. In JavaScript, there is no particular way to represent a graph. There are various pathfinding packages available on NPM, but none utilizes the graph format. They often let graph edges contain a weight, and that is the distance or cost-related with it. For instance, we have the `dijkstrajs` package. A recognized

way to pathfinding, related to the `findRoute` function is the Dijkstra's algorithm, after Edsger Dijkstra, the package first writer. The `js` suffix is regularly attached to package names to specify that you can write in JavaScript. So, if you want to use that package, ensure that the graph saves in its expected format.

```
const {find_path} = require("dijkstrajs");
let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}
console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

It can be a roadblock to composition when several packages use separate data structures to define related things, and it is hard to combine them. Therefore, when you are designing for composability, know exactly the data structures others are using and you can follow their example.

### Summary

In this chapter, we covered the Modules. Modules give structure to more extensive programs by differentiating the code into bits with understandable dependencies and interfaces. The interface is a module which you can see from other modules, and the dependencies are different modules that it uses.

### Exercise

Write a JavaScript program to show the current day and time in the following format.

Today is Tuesday.

Current time is: 10 PM: 30: 38

### Solution



```
1 <!DOCTYPE html>
2
3 <html>
4   <head>
5     <meta charset="utf-8">
6     <title>JavaScript current day and time</title>
7   </head>
8   <body>
9     </body>
10 </html>
```

### JavaScript Code:

```
1 var today = new Date();
2
3 var day = today.getDay();
4
5 var daylist = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];
6 console.log("Today is " + daylist[day] + ".");
7
8 var hour = today.getHours();
9
10 var minute = today.getMinutes();
11 var second = today.getSeconds();
12
13 var prepand = (hour >= 12) ? "PM" : "AM";
14
15 hour = (hour >= 12) ? hour - 12 : hour;
16
17 if (hour==0 && prepand=='PM')
18 [
19   if ((minute/10==0 || minute==0) && second==0)
20   {
21     hour=12;
22     prepand='Noon';
23   }
24 ]
```

```
11 else
12 {
13     hour=12,
14     prepare='AM'
15 }
16
17 if (hour==12 & prepare=='AM')
18 {
19     if (minute==0 & second==0)
20     {
21         hour=12,
22         prepare='Midnight',
23     }
24     else
25     {
26         hour=12,
27         prepare='PM'
28     }
29 }
30
31 console.log("Current Time : "+hour + prepare + ":" + minute + ":" + second);
```

# Chapter 12: Asynchronous Programming

The processor is a vital part of a computer that performs the individual steps that balance our programs. The speed level of something like a loop that influences numbers completion depends on the processor's speed. A lot of programs communicate with things outside of the processor. For instance, they may interact through a computer network or demand for data from the hard disk, which is quite slower than obtaining it from memory. In part, the operating system handles this part and will change the processor between various programs that are running.

## Asynchronicity

In the asynchronous programming model, everything occurs one at a time. If you call a function that executes a long duration action, results will return when the operation completes only. No other action would be able to take place during execution. An asynchronous model enables several things to occur at the same time. When an action begins, the program does not stop running. When the task completes, it notifies the program and can access the result (for instance, the information read from disk). You can differentiate between synchronous and asynchronous programming through a little example: a program that obtains two resources from the network and then unifies the results.

In an asynchronous environment, the request function returns when the work completes. The simplest way to execute this task is to call the requests one after another. It consists of the drawback that begins the second request only when the first task completes. The sum of both response times will be the total time taken. To solve this problem in a synchronous system, start adding threads of control. A thread is a running program that its execution could be rendered with other programs through the operating system because modern computers have several processors, various threads can run at once on separate processors. A second thread may begin the second request, and then the two threads both wait for their results to return, after that, they resynchronize to unify both results. In the asynchronous model, the network time is part of the timeline for a stated thread of control. In an asynchronous model, launching a network action notionally creates a division in the timeline. The program that commences the action does not stop running, and then the action occurs with it, it informs the program when it completes.

Another way to determine the similarities is that waiting for tasks to complete is indirect in the synchronous model, while it is directly under our

control, in the asynchronous one.

## Callbacks

One procedure to asynchronous programming is to enable functions that execute a slow action to gain an extra argument, and the argument is called a callback function. When the action begins and ends, you should call the result with the callback function. For instance, the setTimeout function, accessible both in the browsers and Node.js, waits a stated number of milliseconds and then calls a function. `setTimeout(() => console.log("Tick"), 500);` Waiting is not an ideal type of work, but can be applicable when you are running programs like an animation update or checking if a function is taking longer than the specified time. Running several asynchronous tasks in a row through callbacks signifies that you have to continue setting new functions to control the continuation of the calculations after the actions.

A lot of crow nest computers consist of a longstanding data storage bulb, where data is stored into twigs so that it can be accessed and restored later. Storing, or searching for a piece of data takes little time, so the layout to longstanding storage is asynchronous and utilizes callback functions. Storage bulbs save bits of JSON-encodable data under names. A crow could save data about places it has hidden food under the name "food caches," that holds an array of names that point at other pieces of data, defining the cache. To search for a food cache in the storage bulbs of the Big Oak nest, a crow could run code this way:

```
import {bigOak} from "./crow-tech";
bigOak.readStorage("food caches", caches => {
let firstCache = caches[0];
bigOak.readStorage(firstCache, info => {
console.log(info);
});
});
```

This programming style is practicable, but the indentation level expands with every asynchronous action because it leads in another function. Performing some difficult task such as running several actions concurrently, can get weird. Crow nest computers are created to interact through request-response pairs. That means one nest transfers a message to the other nest, which instantly returns a message, affirming receipt and attaching a reply to the message questions. Every message earmark with a type, which decides how it controls. Your code can describe handlers for certain request types,

and when the request comes in, the handler is called to provide a reply. The layout exported by the “. /crow-tech” module produces callback-based functions for interaction. Nests contain a send method which sends off a request. It awaits the request type, the target nest name, and the requested content as its first three arguments, and it expects a function to call when a reply comes in as its fourth and last argument.

```
bigOak.send("Cow Pasture", "note", "Let's caw loudly at 7PM",
() => console.log("Note delivered."));
```

But to make sure nest can receive that request, you must state a request type named “note.” The code that controls the requests that need to run and, on all nests, can receive this type of messages. Let us state that a crow flies over and installs our handler code on all the nests.

```
import {defineRequestType} from ". /crow-tech";
defineRequestType("note", (nest, content, source, done) => {
  console.log(`${nest.name} received note: ${content}`);
  done();
});
```

The defineRequestType function describes a new type of request. The example attaches support for “note” requests that sends a note to a stated nest. Our implementation calls console.log so that we can check if the request has arrived. Nests consist of a name property that holds its name. The fourth argument stated to the handler, done, is a callback function that calls when it has completed the task. If you use the handler’s return value as the value of the response, that means that a request handler can’t execute asynchronous actions itself. A function performing asynchronous actions returns before the task completes, having a callback ready to be called upon when it finishes. In a way, asynchronicity is contagious. A function that calls a function that performs asynchronously must be asynchronous itself, utilizing a callback to deliver its result. Calling a callback is error-prone and more involved than simply sending back a value.

## Promises

Working with abstract methods is simpler when values represent the methods. In the case of asynchronous actions, instead of making preparations for a function to be called at a certain point in the future, send back an object that stands for the future event. A promise is an asynchronous action that can finish at some point and provide value. It can notify anyone interested when its value is available. The simplest way to build a promise is

by calling `Promise.resolve`. This function guarantees that the value you set encloses in a promise. If it is a promise already, it returns. Otherwise, a new promise will be created instantly with your value as a result.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

To achieve the result of a promise, utilize its former method. It indicates an intended callback function when the promise solves and provides a value. You can attach several callbacks to an individual promise, and they will call the callbacks, even if they were attached when the promise has been completed. The `then` method also send back another promise that solves the value that the handler function returns. It is important to see promises as a device to turn values into an asynchronous reality. A promised value is a value that could be there or could display at a certain point in the future. Computations described in terms of promises work on enclosed values and are performed asynchronously as the values become available. To build a promise, use the `Promise` as a constructor. It contains an odd layout, and the constructor expects a function as an argument, which it calls instantly, sending it a function that can be used to solve the promise. It is how to build a promise-based layout for the `readStorage` function:

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

This asynchronous function returns a significant value.

### Failure

JavaScript computing can fail by outputting an exception. A network request could fail, or some asynchronous code computation could bring an exception. Asynchronous programming callback style most consistent issues are that it makes it very hard to ensure failures reveal correctly to the callbacks. A generally used method is that the first argument to the callback is used to specify the failed action, and the second has the value provided by the action when it completes. Such callback functions always check if they obtain an exception and ensure that any uprising problems, including

exceptions output by functions they call, are given to the correct function. They can reduce when it completes successfully or declined when it fails. Call Reduce handlers it only when the task completes successfully and declines automatically generated to the returned new promise. And when a handler outputs an exception, this automatically triggers the promise provided by its recent call declines. When an element in a chain of asynchronous actions fails, the output of the whole chain declines and no success handlers call past the failing point. Much like solving a promise produces a value, rejecting one also provide one, often called the rejection reason. If an exception in a handler function creates the rejection, use the exception value as the reason. Comparably, when a handler sends back a rejected promise, that rejection goes into the next promise. There's a Promise. Reject function that develops a new, instantly declined promise. If a catch handler output an error, the new promise is equally declined. In shorthand, then also take a decline handler as a second argument, so you should install the two types of handlers in an individual method call. A function sent to the Promise constructor accepts a second argument, together with the resolve function, which it can use to decline the new promise. The chains of promise values developed by calls to then and catch recognized as a pipeline asynchronous values or failures move through. Because those chains built by registering handlers, every link consists of either both or a success handler or a rejection handler related to it. Handlers that do not match the type of output declined. But those that match gets called, and their output decides what type of value comes next, success when it sends back a non-promise value, rejection when it output an exception, and the result of a promise when it sends back one of those.

```
new Promise((_, reject) => reject(new Error("Fail")))
.then(value => console.log("Handler 1"))
.catch(reason => {
  console.log("Caught failure " + reason);
  return "nothing";
})
.then(value => console.log("Handler 2", value));
// → Caught failure Error: Fail
// → Handler 2 nothing
Networks are hard
```

Periodically, there's very little light for the crows' mirror systems to convey a signal is obstructing the way of the signal. It is viable for a signal to be transferred but got declined. Often, transmission failures are unexpected accidents, like a car's headlight obstruct the light signals, and resending the request could make it succeed. So, let's create our request function, and consequently retry the sending of the request more time before it stops. And, since we have established that promises are good, we'll also ensure our request function sends back a promise. When it comes to what they can indicate, promises and callbacks are equal. Callback-based functions are enclosed to uncover a promise-based layout and vice versa. Even when a request and its reply convey successfully, the reply could specify failure, for instance, if the request tries to utilize a request type that has not been described or the handler output an error. To aid this, send and defineRequestType follow the convention declared earlier, where the first argument sent to callbacks is the reason for failure if any, and the second is the Definite result. They are interpreted to promise resolution and rejection by our wrapper.

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;

    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
      }, 250);
    }
    attempt(1);
  });
}
```

Because promises can be accepted or declined once only, it will work. The first time resolve or reject call decides the result of the promise, and any other calls, are disregarded. To create an asynchronous loop, for the retries, use a recursive function, an efficient loop does not allow us to stop and wait for an asynchronous action. The attempt function creates a single attempt to transfer a request. It also specifies a timeout that, if there is comeback response after 250 milliseconds, begins the next attempt or, if it is the fourth try, declines the promise with an instance of Timeout being the reason. Retrying each quarter-second and stops when there is no response after a second is certainly arbitrary if the request shows up, but the handler takes longer for requests to transfer several times. Create your handlers with that problem in mind, and coupled messages are not harmful. To segregate ourselves from callbacks totally, define a cover for defineRequestType that enables the handler function to send back a promise or simple value and wires that do the callback for us.

```
function requestType(name, handler) {  
  defineRequestType(name, (nest, content, source,  
    callback) => {  
    try {  
      Promise.resolve(handler(nest, content, source))  
        .then(response => callback(null, response),  
          failure => callback(failure));  
    } catch (exception) {  
      callback(exception);  
    }  
  });  
}
```

Promise.resolve is used to transform the returned value by the handler to a promise if it is not ready.

### ***Collections of promises***

Every nest computer stores an array of other nests inside transmission distance in its neighbor's property. To examine which is reachable, write a function that tries to transfer a "ping" request (request that demands a response) to all of them and check for which ones come back. If you're working with a group of promises running at once, the Promise.all function is very important. It sends back a promise that holds on for all of the promises in the array to reconcile and then settle to an array of the values

that these promises provided. If any promise is declined, the result of Promise.all is also declined.

```
requestType("ping", () => "pong");
function availableNeighbors(nest) {
let requests = nest.neighbors.map(neighbor => {
return request(nest, neighbor, "ping")
.then(() => true, () => false);
});
return Promise.all(requests).then(result => {
return nest.neighbors.filter(_ , i) => result[i]);
});
```

### Network flooding

Nests can communicate with their neighbors only and that greatly affects the significance of this network. To broadcast information to the entire network, a solution is to create a type of request that is sent automatically to neighbors. These neighbors then send it to their neighbors until the entire network has collected the message.

```
import {everywhere} from "./crow-tech";
everywhere(nest => {
nest.state.gossip = [];
});
function sendGossip(nest, message, exceptFor = null) {
nest.state.gossip.push(message);
for (let neighbor of nest.neighbors) {
if (neighbor == exceptFor) continue;
request(nest, neighbor, "gossip", message);
}
}
requestType("gossip", (nest, message, source) => {
if (nest.state.gossip.include(message)) return;
console.log(` ${nest.name} receive gossip '${{
messages}' from ${source}`);
sendGossip(nest, message, source);
});
```

You can avoid transferring the similar message across the network forever, every nest saves an array of gossip strings that it has seen. To describe this

array, uses the everywhere function that runs code on each nest to attach a property to the nest's state object.

### Message routing

If a set node wants to communicate with a single other node. The best approach is to build a path for messages to jump from node to node until they get to their destination. You need to understand the interface of the network. To send a request to a nest at a distance, it is important to know which neighboring nest can get the job done. Every nest knows about its direct neighbors only, it cannot compute a route, it lacks the information. You must understand how to spread the information about these connections to every nest in a way that enables it to transform with time, when new nests are created. Now you can use flooding, but instead of checking whether a message has been collected, now check if the new set of neighbors for a given nest is similar to the current set.

```
requestType("connections", (nest, {name, neighbors},  
source) => {  
let connections = nest.state.connections;  
if (JSON.stringify(connections.get(name)) ==  
JSON.stringify(neighbors)) return;  
connections.set(name, neighbors);  
broadcastConnections(nest, name, source);  
});  
function broadcastConnections(nest, name, exceptFor = null) {  
for (let neighbor of nest.neighbors) {  
if (neighbor == exceptFor) continue;  
request(nest, neighbor, "connections", {  
name,  
neighbors: nest.state.connections.get(name)  
});  
}  
}  
everywhere(nest => {  
nest.state.connections = new Map;  
nest.state.connection.set(nest.name, nest.neighbors);  
broadcastConnections(nest, nest.names);  
});
```

The comparison utilizes JSON.stringify because ==, on objects or arrays, will send back true when the value of the two are the same only. Differentiating the JSON strings is a very successful way to compare their content. The nodes instantly begin transmitting their connections except to the unreachable nests, which give them a map of the existing network graph. The findRoute function finds a path to reach a set node within the network. But instead of sending back the whole route, it just brings back the next step. That next nest will use its existing information about the network, determine where it transfers the message.

```
function findRoute(from, to, connections) {  
let work = [{at: from, via: null}];  
for (let I = 0; I < work.length; i++) {  
let {at, via} = work[i];  
for (let next of connections.get(at) || []) {  
if (next == to) return via;  
if (!work.some(w => w.at == next)) {  
work.push({at: next, via: via || next});  
}  
}  
}  
}  
return null;  
}
```

You can now create a function that can send messages to long distance. If the message is attached to a direct neighbor, it is conveyed as usual. If not, it is wrapped in an object and transferred to a neighbor closer to the target, through the "route" request type, that will make that neighbor repeat the same character.

```
function routeRequest(nest, target, types, content) {  
if (nest.neighbors.includes(target)) {  
return request(nest, target, types, content);  
} else {  
let via = findRoute(nest.name, target,  
nest.state.connections);  
if (!via) throw new Error(`No routes to ${target}`);  
return request(nest, via, "route",  
{target, type, content});  
}
```

```

}

requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});

```

We have created different layers of functionality at the top of a primary communication system to enable easy use. This is a simplified model of how computer networks work.

### Async functions

To save significant information, crows use the duplication method, they duplicate it across nests. That way, when a bird dismantles a nest, the information would not be lost. To recover a particular piece of information that has no storage bulb, a nest computer would talk to random other nests in the network until the one that has it is found.

```

requestType("storage", (nest, name) => storage(nest, name));
function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n != nest.name);
  function next() {
    if (sources.length == 0) {
      return Promise.reject(new Error("Not found"));
    } else {
      let source = sources[Math.floor(Math.random() *
        sources.length)];
      sources = sources.filter(n => n != source);
      return routeRequest(nest, source, "storage", name)
        .then(value => value != null ? value : next(),
        next);
    }
  }
}

```

```
return next();
}
```

Object.keys cannot work on connections because it is a Map. It contains a key method, but that sends back an iterator instead of an array. An iterator can be transformed to an array using the Array.from function. Various asynchronous actions are tied together in unclear ways. Now you need a recurring function to model looping across the nests. In a synchronous programming model, it is easy to express. JavaScript also lets you write pseudo-synchronous code to define asynchronous computation. An async function is a function that completely bring back a promise and that can wait for other promises in a synchronous way. Rewrite findInStorage like this:

```
async function findInStorage(nest, name) {
let local = await storage(nest, name);
if (local != null) return local;
let sources = network(nest).filter(n => n != nest.name);
while (sources.length > 0) {
let source = sources[Math.floor(Math.random() *
sources.length)];
sources = sources.filter(n => n != source);
try {
let found = await routeRequest(nest, source, "storage",
name);
if (found != null) return found;
} catch (_) {}
}
throw new Error("Not found");
}
```

An async function is marked with the word `async` and then the function keyword. Methods can also become `async` by writing `async` before their name. When the function or method is called, it brings back a promise. When the body returns something, that promise is settled. If it outputs an exception, the promise is declined. Within an `async` function, the word `await` is set in front of an expression to await a promise to get settled and only then proceeds to execute the function. Such functions cannot run from start to finish at once, but it can be frozen at points and can restart later. For non-trivial asynchronous code, this notation is easier than directly through promises.

## Generators

Async functions do not have to be paused and then resumed again. JavaScript consists of a feature called generator functions. These are identical but do not contain the promises. When you call a function with `function*` (put an asterisk after the word `function`), it changes to a generator. When you call a generator, it sends back an iterator.

```
function* powers(n) {  
  for (let current = n;; current *= n) {  
    yield currently;  
  }  
}  
  
for (let power of powers(3)) {  
  if (power > 50) break;  
  console.log(power);  
}  
// → 3  
// → 9  
// → 27
```

Normally, if you call `powers`, the function is frozen at the beginning. Each time you call `next` on the iterator, the function will run until it hits a `yield` expression, which will pause it and make the yielded value the next value provided by the iterator. When the function returns, the iterator completes. Writing iterators is simple when you utilize the generator functions. Write the iterator for the `Group` class with this generator:

```
Group.prototype[Symbol.iterator] = function*() {  
  for (let i = 0; i < this.members.length; i++) {  
    yield this.members[i];  
  }  
};
```

There's no need to build an object to hold the iteration state, and generators store their local state immediately and they yield every time. This `yield` expression can happen only directly within the generator function and not in an inner function that you set within it. The generator saves when yielding. This is its local environment and the position where it yields. An `async` function is a unique kind of generator. It creates a promise when it's called, which resolves when it completes and declines when it outputs an exception.

Anytime it yields a promise, the result of that promise will be the result of the await expression.

## The event loop

Asynchronous programs perform one by one. Every piece may begin a few actions and arrange code to execute when the action completes or fails. Between these pieces, the program does nothing, awaiting the next action. So you cannot call callbacks by the code that arranged them. If you call setTimeout from inside a function, that function will have been sent back before the callback function gets called. Asynchronous character occurs on its empty function called stack. Since each callback begins with an empty stack, your catch handlers will not be on the stack when an exception runs.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (_) {
  // This will not run
  console.log("Caught!");
}
```

No matter how events like timeouts or incoming requests occur, a JavaScript environment runs one program at once. When there's nothing to do, the loop stops. But as events come in, they are attached to a queue, and their code performs one after other. Because two things do not run at once, slow-running code may hold up the handling of other events. The below example sets a timeout but also lingers until after the timeout's fixed point of time, making the timeout to be late.

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

Promises resolve or decline as a new event. Even if a promise is resolved already, waiting for it will make your callback run after the existing script

completes, instead of immediately.

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
// → Me first!
// → Done
```

### Asynchronous bugs

When you run your program synchronously, there is no given occurring transformation besides the ones that program themselves. But when you run an asynchronous program, this is another scenario. They could contain gaps during execution that other code can utilize. Below is an illustration. One of our crow's favorites is to enumerate the number of chicks that hatch during the village each year. Nests keep this count within their storage bulbs. The below code tries to tally the counts from every nest for a particular year:

```
function anyStorage(nest, source, name) {
if (source == nest.names) return storage(nest, name);
else return routeRequest(nest, source, "storage", name);
}
async function chicks(nest, year) {
let list = "";
await Promise.all(network(nest).map(async name => {
list += `${name}: ${{
await anyStorage(nest, names, `chicks in ${year}`)
}\n`;
}}));
return list;
}
```

The `async name =>` displays that arrow functions are also made `async` by placing the word `async` in front of them. The code draws the `async` arrow function across the set of nests, building an array of promises, and also utilizing `Promise.all` to wait before sending back the list they created. But it's broken. It'll send back a single line of output, listing the slowest to respond nests. Do you know why?

The problem is in the `+=` operator that takes the present value of the list when the statement begins executing and when the wait ends, sets the list binding to be the value together with the added string. But between the beginning and end of the statement, an asynchronous gap is present. The `map` expression is the first to run before you can add anything to the list, so

every `+=` operator begins from an empty string and finishes when its retrieval storage completes. As usual, calculating new values is an error-free process rather than transforming existing values.

```
async function chicks(nest, year) {  
let lines = network(nest).map(async name => {  
return name + ": " +  
await anyStorage(nest, names, `chicks in ${year}`);  
});  
return (await Promise.all(lines)).join("\n");  
}
```

Mistakes are elementary to make, especially during the use of `await`, and you should know where gaps are within your code.

### Summary

We learned everything about Asynchronous programming in this chapter. Asynchronous programming enables the expression waiting for action that runs for a long time without giving the program problems throughout the operations. JavaScript environments generally utilize this style of programming through callbacks, functions which to call when actions end. We touched on the event loop as well. An event loop creates a plan for such callbacks to be called when it is time, one after another, to avoid overlapping.

### Exercise

Write a JavaScript program to transform an asynchronous function to return a promise.

### Solution

---

### HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>Convert an asynchronous function to return a promise</title>
6 </head>
7 <body>
8
9 </body>
10 </html>
```

---

### JavaScript Code:

```
1 //#Source https://bit.ly/2newfJ2
2 const promisify = func => (...args) =>
3   new Promise((resolve, reject) =>
4     func(...args, (err, result) => (err ? reject(err) : resolve(result)))
5   );
6 const delay = promisify((d, cb) => setTimeout(cb, d));
7 delay(2000).then(() => console.log('Hi!'));
```

---



# Chapter 13: Parsing

Developing one's programming language is relatively easy and very exciting. This chapter will teach you how to build your language. We will create a programming language called Egg. A small, simple language is yet strong enough to express any thinkable computation. It enables pure abstraction determined functions.

## Parsing

The noticeable part of a programming language is the notation or syntax. A parser is a program that reads the text and provides a data layout that reflects the layout of the program within that text. If the text refuses to create a valid program, the parser would indicate an error. Our language would contain precise and uniform syntax. Everything included in Egg is an expression. An expression could be the binding name, number, string, or application. Utilize Applications for function calls and as well as constructs like if or while.

To keep the simple parser, strings within the Egg do not support backslash escapes. A string is a succession of characters that are not double quotes, enclosed in double-quotes. Name Binding can contain any character that isn't whitespace and that do not contain a unique meaning in the syntax. Write applications in JavaScript, by placing parentheses after an expression and putting any number of arguments within the parentheses, differentiated by commas.

```
do(define(x, 10),  
if(>(x, 5),  
print("large"),  
print("small")))
```

The consistency of the Egg language means that operators in JavaScript (like the `>`) are called bindings in this language, utilized similarly like other functions. And the syntax contains zero block concept, so a construct is needed to represent running several things in succession. The data layout the parser utilizes to define a program containing expression objects. Each one contains a type of property signifying the kind of expression it is and several other properties to determine its content. Expressions of type "value" mean strings or numbers. Their value property consists of the string or number value that they stand for. Use expressions of type "word" for identifiers (names). These types of objects contain a name property that holds the identifier's title as a string. And then the "apply" expressions stand for

applications. They include an operator property that cites the applied expression, together with an args property that holds an array of argument expressions.

The `>(x, 5)` part of the new program would be like this:

```
{  
  type: "apply",  
  operator: {type: "word", name: ">"},  
  args: [  
    {type: "word", name: "x"},  
    {type: "value", value: 5}  
  ]  
}
```

We describe a function `parseExpression`, which recognize a string as input and send back an object that consists of the data layout for the expression at the beginning of the string, together with the string part left after parsing this expression. When you are parsing subexpressions, you can call this function again, capitulating the argument expression with the remaining text. This text could consist of more arguments or the closing parenthesis that ends the argument list.

Below is the parser first part:

```
function parseExpression(program) {  
  program = skipSpace(program);  
  let match, expr;  
  if (match = /^[^"]*/.exec(program)) {  
    expr = {type: "value", value: match[1]};  
  } else if (match = /\d+/.exec(program)) {  
    expr = {type: "value", value: Number(match[0])};  
  } else if (match = /^[^\s(),#"]+/.exec(program)) {  
    expr = {type: "word", name: match[0]};  
  } else {  
    204  
    throw new SyntaxError("Unexpected syntax: " + programs);  
  }  
  return parseApply(expr, programs.slice(match[0].length));  
}  
function skipSpace(string) {  
  let first = string.search(/\S/);
```

```
if (first == -1) return "";
return string.slice(first);
}
```

## The evaluator

The evaluator is used to run the syntax tree for a program. Set it a syntax tree and a scope object that relate names with values, and it will access the expression that the tree stands for and send back the value that it provides.

```
const specialForms = Object.create(null);
function evaluate(expr, scope) {
if (expr.type == "value") {
return expr.value;
} else if (expr.type == "word") {
if (expr.name in scope) {
return scope[expr.name];
} else {
throw new ReferenceError(
`Undefined binding: ${expr.name}`);
}
} else if (expr.type == "apply") {
let {operator, args} = expr;
if (operator.type == "word" &&
operator.name in specialForms) {
return specialForms[operator.name](expr.args, scope);
} else {
let op = evaluate(operator, scope);
if (typeof op == "function") {
return op(...args.map(arg => evaluate(arg, scope)));
} else {
throw new TypeError("Applying a non-function.");
}
}
}
}
}
}
```

The evaluator contains codes for every expression type. A literal value expression provides its value. To bind, check if it describes in the scope and, if yes, get the value of the binding. If it contains a unique form that does not access anything, and yet transfers the argument expressions with scope, to

the function that influences the form. If the call is typical, access the operator, determine if it is a function, and call it with the accessible arguments. Utilize the common JavaScript function values to stand for the value of Egg's function. The recursive layout of evaluating is a similar layout of the parser, and the two mirror the layout of the language. You can combine the parser using the evaluator and evaluate when parsing, but separating them gives a clearer of the program. It is all you need to combine Egg.

## Special forms

The specialForms object describes unique syntax in Egg. It relates words with functions that access such forms. It is presently empty. Let us insert if.

```
specialForms.if = (args, scope) => {
if (args.length != 3) {
throw new SyntaxError("Wrong number of args to if");
} else if (evaluate(args[0], scope) !== false) {
return evaluate(args[1], scope);
} else {
return evaluate(args[2], scope);
}
};
```

Egg's if construct expects three arguments. It will access the first, and if the value is not false, it moves to access the second. If not, it accesses the third. The if form looks more like JavaScript's ternary?: operator than JavaScript's if. It is an expression and not a statement, and provides a value, which are the results of the second or third argument. Egg are also different from JavaScript in handling the condition value to it. It does not relate with things like zero or empty strings as false, only the exact value false. All arguments to functions are accessed before you call the function.

```
specialForms.while = (args, scope) => {
if (args.length != 2) {
throw new SyntaxError("Wrong number of args to while");
}
while (evaluate(args[0], scope) !== false) {
evaluate(args[1], scope);
}
// Since undefined are not in Egg, we return false,
```

```
// for lack of a significant result.  
return false;  
};
```

One other building block is do, that performs its arguments from top to bottom. Its value is provided by the last argument.

```
specialForms.do = (args, scope) => {  
let value = false;  
for (let arg of args) {  
value = evaluate(arg, scope);  
}  
return value;  
};
```

To build bindings and set new values, you should build a form called define. It awaits a word as its first argument and an expression providing the value to set to that word as its second argument. Since define is an expression, it must send back a value. We will return the value that was set (like JavaScript's = operator).

```
specialForms.define = (args, scope) => {  
if (args.length != 2 || args[0].type != "word") {  
throw new SyntaxError("Incorrect use of define");  
}  
let value = evaluate(args[1], scope);  
scope[args[0].name] = value;  
return value;  
};
```

The environment

The scope that evaluate allows is an object containing few properties that have their name tally with binding names. Their values are in accordance to the values those bounded bindings. Below is an illustration of an object to stand for global scope. To enable the utilization of the earlier described if construct, the Boolean values must be accessible. Therefore, there are two Boolean values only, all you need to do is to bind two names to the values true and false and utilize them.

```
const topScope = Object.create(null);  
topScope.true = true;  
topScope.false = false;
```

We can now examine a simple expression that nullify a Boolean value.

```
let prog = parses(`if(true, false, true)`);  
console.log(examine(prog, topScope));  
// → false
```

To provide fundamental arithmetic as well as comparison operators, lets include function values to scope. We will use function on few operator functions within a loop, and not stating them one by one.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {  
topScope[op] = Function("a, b", `return a ${op} b;`);  
}
```

we will enclose console.log in a function to output the values and call it print.

```
topScope.print = value => {  
console.log(value);  
return value;  
};
```

That provides adequate elementary tools to code easy program, analyze and run it in a new scope:

```
function run(program) {  
return evaluate(parse(program), Object.create(topScope));  
}
```

The chains of the object prototype will stand for nested scopes to enable add bindings without altering the high-level scope.

```
run(`  
do(define(total, 0),  
define(count, 1),  
while(<(count, 11),  
do(define(total, +(total, count))),  
define(count, +(count, 1)))),  
print(total))  
`);  
// → 55
```

This concept explains the total numbers from 1 to 10, as indicated in egg.

## ***Functions***

Programming language that has no functions is a really bad one, although it is easy to insert a function that uses its last argument as the function's body. It also utilizes every argument before that as the function's parameters names.

```

specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let param = args.slice(0, args.length - 1).map(expr => {
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
  });
  return expr.name;
};

211
return function() {
  if (arguments.length != params.length) {
    throw new TypeError("Wrong number of arguments");
  }
  let localScope = Object.create(scope);
  for (let i = 0; i < arguments.length; i++) {
    localScope[params[i]] = arguments[i];
  }
  return evaluate(body, localScope);
};
};

```

The egg functions have their own local scope. The function provided by the fun form builds this local scope and attaches the argument bindings to it. It then accesses the function body in this scope and gives back the result.

```

run(`

do(define(plusOne, fun(a, +(a, 1))),
print(plusOne(10)))
`);

// → 11
run(`

do(define(pow, fun(base, exp,
if==(exp, 0),
1,
*(base, pow(base, -(exp, 1))))),
print(pow(2, 10)))
`);

```

```
`);  
// → 1024
```

### Summary

We covered the parsing, functions, special forms, and the evaluator in this chapter. Notation or syntax is a very important aspect of programming. A parser reads text and offers a data layout that shows the program layout in the text. If the text creates an invalid program, the parser would signify an error.

### Exercise

Write a JavaScript function to calculate the factors of a positive integer.

### Solution

## HTML Code:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset=utf-8 />
5 <title>Compute the factors of a positive integer</title>
6 </head>
7 <body>
8
9 </body>
10 </html>
```

## JavaScript Code:

```
1 function factors(n)
2 {
3     var num_factors = [], i;
```

```
5 | for (i = 1; i <= Math.floor(Math.sqrt(n)); i += 1)
6 |   if (n % i === 0)
7 |   {
8 |     num_factors.push(i);
9 |     if (n / i !== i)
10 |       num_factors.push(n / i);
11 |
12 |   num_factors.sort(function(x, y)
13 |   {
14 |     return x - y}); // numeric sort
15 |   return num_factors;
16 |
17 | console.log(factors(15)); // [1,3,5,15]
18 | console.log(factors(16)); // [1,2,4,8,16]
19 | console.log(factors(17)); // [1,17]
```

## **Conclusion**

This book consists of important information connected to the JavaScript programming language. The information in this book will help you to understand and practice the several JavaScript operators, loops and conditional statements, functions, the scope of variables, invent types, and so on. And while edging toward the end of the course, you will have an excellent understanding of the JavaScript language as you will have learned JavaScript and forms, you will understand the basics of jQuery, frames and debugging scripts. The best advice to give a programmer is to keep practicing because that is the only way you can master what you have learned.

## **References**

Javascript.info (2019). The Modern JavaScript Tutorial. Retrieved from <https://javascript.info/>

Eloquent JavaScript 3rd edition. Retrieved from [https://eloquentjavascript.net/Eloquent\\_JavaScript.pdf](https://eloquentjavascript.net/Eloquent_JavaScript.pdf)

JavaScript basic - Exercises, Practice, Solution. Retrieved from <https://www.w3resource.com/javascript-exercises/javascript-basic-exercises.php>.

# Notes

[← 1]

<https://sites.google.com/site/prgimr/sql/ddl-commands---create---drop--alter>

[[←](#) 2]

[https://www.w3schools.com/sql/sql\\_primarykey.asp](https://www.w3schools.com/sql/sql_primarykey.asp)

[[←](#) 3]

<https://www.1keydata.com/sql/sql-create-view.html>

[← 4]

<https://www.1keydata.com/sql/sql-create-view.html>

[ ← 5 ]

[https://www.w3schools.com/sql/sql\\_join\\_inner.asp](https://www.w3schools.com/sql/sql_join_inner.asp)

[ ← 6 ]

[https://www.w3schools.com/sql/sql\\_join\\_right.asp](https://www.w3schools.com/sql/sql_join_right.asp)

[ ← 7 ]

[https://www.w3schools.com/sql/sql\\_join\\_left.asp](https://www.w3schools.com/sql/sql_join_left.asp)

[[←](#) 8]

[https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp)

[[←](#) 9]

[https://www.w3schools.com/sql/sql\\_union.asp](https://www.w3schools.com/sql/sql_union.asp)

[[← 10](#)]

[https://www.techonthenet.com/sql/union\\_all.php](https://www.techonthenet.com/sql/union_all.php)

[[← 11](#)]

[https://www.w3schools.com/sql/sql\\_notnull.asp](https://www.w3schools.com/sql/sql_notnull.asp)

[[← 12](#)]

[https://www.w3schools.com/sql/sql\\_unique.asp](https://www.w3schools.com/sql/sql_unique.asp)

[[←](#) 13]

[https://www.w3schools.com/sql/sql\\_unique.asp](https://www.w3schools.com/sql/sql_unique.asp)

[ ← 14 ]

<https://chartio.com/resources/tutorials/how-to-alter-a-column-from-null-to-not-null-in-sql-server/>

[ ← 15 ]

<https://www.tutorialspoint.com/sql/sql-primary-key.htm>

[ ← 16 ]

<https://www.tutorialspoint.com/sql/sql-primary-key.htm>

[ ← 17 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-2017>

[[←](#) 18]

<https://docs.faircom.com/doc/esql/32218.htm>

[ ← 19 ]

<https://stackoverflow.com/questions/7573590/can-a-foreign-key-be-null-and-or-duplicate>

[ ← 20 ]

<https://stackoverflow.com/questions/7573590/can-a-foreign-key-be-null-and-or-duplicate>

[[← 21](#)]

[https://www.w3schools.com/sql/sql\\_check.asp](https://www.w3schools.com/sql/sql_check.asp)

[ ← 22 ]

<https://stackoverflow.com/questions/11981868/limit-sql-server-column-to-a-list-of-possible-values>

[ ← 23 ]

<https://www.tutorialspoint.com/sql/sql-primary-key.htm>

[ ← 24 ]

<https://www.tutorialspoint.com/sql/sql-primary-key.htm>

[ ← 25 ]

<https://stackoverflow.com/questions/11981868/limit-sql-server-column-to-a-list-of-possible-values>

[ ← 26 ]

<https://stackoverflow.com/questions/11981868/limit-sql-server-column-to-a-list-of-possible-values>

[ ← 27 ]

<https://docs.microsoft.com/en-us/biztalk/core/step-2-create-the-inventory-request-schema>

[[←](#) 28]

[https://www.techonthenet.com/oracle/schemas/create\\_schema.php](https://www.techonthenet.com/oracle/schemas/create_schema.php)

[ ← 29 ]

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-schema-transact-sql?view=sql-server-2017>

[[←](#) 30]

<https://www.postgresql.org/docs/9.3/sql-createschema.html>

[[← 31](#)]

[https://www.techonthenet.com/sql/tables/create\\_table2.php](https://www.techonthenet.com/sql/tables/create_table2.php)

[[←](#) 32]

[https://www.w3schools.com/sql/sql\\_wildcards.asp](https://www.w3schools.com/sql/sql_wildcards.asp)

[[←](#) 33]

[https://www.w3schools.com/sql/sql\\_like.asp](https://www.w3schools.com/sql/sql_like.asp)

[ ← 34 ]

[https://www.w3schools.com/sql/sql\\_like.asp](https://www.w3schools.com/sql/sql_like.asp)

[[← 35](#)]

[https://www.w3schools.com/sql/sql\\_where.asp](https://www.w3schools.com/sql/sql_where.asp)

[ ← 36 ]

[https://www.w3schools.com/sql/sql\\_orderby.asp](https://www.w3schools.com/sql/sql_orderby.asp)

[[←](#) 37]

[https://docs.oracle.com/cd/B19306\\_01/server.102/b14200/statements\\_6014.htm](https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6014.htm)

[ ← 38 ]

[https://docs.oracle.com/cd/B19306\\_01/server.102/b14200/statements\\_6014.htm](https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6014.htm)

[[←](#) 39]

<https://stackoverflow.com/questions/2578194/what-is-ddl-and-dml>

[[← 40](#)]

[https://docs.oracle.com/cd/B19306\\_01/server.102/b14200/statements\\_6014.htm](https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6014.htm)

[ ← 41 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-detach-and-attach-sql-server?view=sql-server-2017>

[[← 42](#)]

<https://www.tutorialspoint.com/sql/sql-like-clause.htm>

[[← 43](#)]

<https://www.tutorialspoint.com/sql/sql-like-clause.htm>

[ ← 44 ]

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

[ ← 45 ]

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

[ ← 46 ]

<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

[← 47]

[ ← 48 ]

<https://docs.microsoft.com/en-us/sql/t-sql/functions/round-transact-sql?view=sql-server-2017>

[ ← 49 ]

<https://docs.microsoft.com/en-us/sql/t-sql/functions/round-transact-sql?view=sql-server-2017>

[ ← 50 ]

<https://docs.microsoft.com/en-us/sql/t-sql/functions/round-transact-sql?view=sql-server-2017>

[[← 51](#)]

<https://docs.microsoft.com/en-us/sql/t-sql/functions/round-transact-sql?view=sql-server-2017>

[ ← 52 ]

<https://docs.microsoft.com/en-us/sql/t-sql/functions/round-transact-sql?view=sql-server-2017>

[← 53]

[ ← 54 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[ ← 55 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[ ← 56 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[ ← 57 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[ ← 58 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[ ← 59 ]

<https://stackoverflow.com/questions/1475589/sql-server-how-to-use-an-aggregate-function-like-max-in-a-where-clause>

[[←](#) 60]

<https://stackoverflow.com/questions/886786/how-to-change-a-table-name-using-an-sql-query>

[[← 61](#)]

<https://www.1keydata.com/sql/alter-table-rename-column.html>

[[← 62](#)]

<https://www.1keydata.com/sql/alter-table-rename-column.html>

[[←](#) 63]

<https://www.1keydata.com/sql/alter-table-rename-column.html>

[ ← 64 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/modify-columns-database-engine?view=sql-server-2017>

[ ← 65 ]

<https://www.tutorialspoint.com/sql/sql-inner-joins.htm>

[[← 66](#)]

<http://www.sql-join.com/sql-join-types/>

[[← 67](#)]

<http://www.sql-join.com/sql-join-types/>

[[← 68](#)]

<http://www.sql-join.com/sql-join-types/>

[[← 69](#)]

<http://www.sql-join.com/sql-join-types/>

[[← 70](#)]

<http://www.sql-join.com/sql-join-types/>

[[← 71](#)]

<http://www.sql-join.com/sql-join-types/>

[← 72]

<http://www.sql-join.com/sql-join-types/>

[ ← 73 ]

<https://www.tutorialspoint.com/sql/sql-top-clause.htm>

[ ← 74 ]

<https://www.tutorialspoint.com/sql/sql-top-clause.htm>

[← 75]

| ID | NAME     | ADDRESS  | AGE | SALARY  |
|----|----------|----------|-----|---------|
| 2  | Mercy    | Mercy32  | 25  | 3500.00 |
| 3  | Joel     | Joel42   | 30  | 4000.00 |
| 4  | Alice    | Alice442 | 31  | 2500.00 |
| 5  | Nicholas | nico442  | 45  | 5000.00 |
| 6  | Milly    | mil342   | 32  | 2000.00 |
| 7  | Grace    | gra361   | 35  | 4000.00 |

[https://www.tutorialspoint.com/sql/sql\\_top\\_clause.htm](https://www.tutorialspoint.com/sql/sql_top_clause.htm)

[ ← 76 ]

[https://www.tutorialspoint.com/sql/sql\\_top-clause.htm](https://www.tutorialspoint.com/sql/sql_top-clause.htm)

[[← 77](#)]

[https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php)

[ ← 78 ]

[https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php)

[[←](#) 79]

[https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php)

[[←](#) 80]

[https://www.techonthenet.com/sql/group\\_by.php](https://www.techonthenet.com/sql/group_by.php)

[[← 81](#)]

<https://www.tutorialspoint.com/sql/sql-not-null.htm>

[ ← 82 ]

<https://www.tutorialspoint.com/sql/sql-default.htm>

[ ← 83 ]

<https://www.tutorialspoint.com/sql/sql-default.htm>

[ ← 84 ]

<https://www.tutorialspoint.com/sql/sql-default.htm>

[ ← 85 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-primary-keys?view=sql-server-2017>

[ ← 86 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-primary-keys?view=sql-server-2017>

[ ← 87 ]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-primary-keys?view=sql-server-2017>

[[← 88](#)]

<https://docs.microsoft.com/en-us/sql/relational-databases/tables/create-check-constraints?view=sql-server-2017>

[ ← 89 ]

<https://www.tutorialspoint.com/sql/sql-index.htm>

[[← 90](#)]

<https://www.tutorialspoint.com/sql/sql-index.htm>

[[← 91](#)]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[[← 92](#)]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[ ← 93 ]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[[← 94](#)]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[[← 95](#)]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[ ← 96 ]

[https://www.techonthenet.com/sql/tables/alter\\_table.php](https://www.techonthenet.com/sql/tables/alter_table.php)

[ ← 97 ]

<https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb>

[ ← 98 ]

<https://codescracker.com/c-sharp/c-sharp-program-structure.htm>

[[← 99](#)]

[https://www.tutorialspoint.com/compile\\_csharp\\_online.php](https://www.tutorialspoint.com/compile_csharp_online.php)

[[← 100](#)]

<https://www.tutorialsteacher.com/csharp/csharp-keywords>

[← 101]

<http://download.microsoft.com/>

[[← 102](#)]

<https://www.guru99.com/c-sharp-data-types.html>

[[← 103](#)]

<https://chrisheydrick.com/2014/12/17/c-struct-sizes/>

[[← 104](#)]

<https://csharp.net-tutorials.com/basics/variables/>

[ ← 105 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table>

[ ← 106 ]

<https://code-maze.com/csharp-basics-type-conversion/>

[ ← 107 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions>

[[←](#) 108]

<https://www.programiz.com/csharp-programming/operators>

[[← 109](#)]

<http://zetcode.com/lang/csharp/operators/>

[[← 110](#)]

<https://www.programiz.com/csharp-programming/if-else-statement>

[[← 111](#)]

<https://www.programiz.com/csharp-programming/switch-statement>

[[← 112](#)]

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto>

[[←](#) 113]

<https://www.tutorialsteacher.com/csharp/csharp-while-loop>

[[← 114](#)]

<https://www.tutorialsteacher.com/csharp/csharp-do-while-loop>

[[← 115](#)]

<https://www.tutorialsteacher.com/csharp/csharp-for-loop>

[[← 116](#)]

<https://www.programiz.com/csharp-programming/foreach-loop>

[[← 117](#)]

<https://csharp-station.com/Tutorial/CSharp/Lesson05>

[[← 118](#)]

<https://www.geeksforgeeks.org/c-sharp-methods/>

[[← 119](#)]

<http://zetcode.com/lang/csharp/methods/>

[ ← 120 ]

<https://www.codeproject.com/Articles/142292/Recursive-methods-in-Csharp>

[ ← 121 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters>

[[← 122](#)]

<http://jonskeet.uk/csharp/parameters.html>

[ ← 123 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>

[[← 124](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_arrays.htm](https://www.tutorialspoint.com/csharp/csharp_arrays.htm)

[[← 125](#)]

<https://www.geeksforgeeks.org/c-sharp-arrays/>

[[←](#) 126]

[https://www.tutorialspoint.com/csharp/csharp\\_classes.htm](https://www.tutorialspoint.com/csharp/csharp_classes.htm)

[ ← 127 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes>

[[← 128](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_struct.htm](https://www.tutorialspoint.com/csharp/csharp_struct.htm)

[ ← 129 ]

<https://www.tutorialsteacher.com/csharp/csharp-struct>

[ ← 130 ]

[https://www.tutorialspoint.com/compile\\_csharp\\_online.php](https://www.tutorialspoint.com/compile_csharp_online.php)

[[← 131](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_encapsulation.htm](https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm)

[[←](#) 132]

[https://www.tutorialspoint.com/csharp/csharp\\_inheritance.htm](https://www.tutorialspoint.com/csharp/csharp_inheritance.htm)

[ ← 133 ]

[https://www.tutorialspoint.com/csharp/csharp\\_polymorphism.htm](https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm)

[[← 134](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_regular\\_expressions.htm](https://www.tutorialspoint.com/csharp/csharp_regular_expressions.htm)

[ ← 135 ]

[https://www.tutorialspoint.com/csharp/csharp\\_exception\\_handling.htm](https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm)

[[← 136](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_file\\_io.htm](https://www.tutorialspoint.com/csharp/csharp_file_io.htm)

[[←](#) 137]

<http://csharp.net-informations.com/file/csharp-file-tutorial.htm>

[ ← 138 ]

<https://www.tutorialsteacher.com/csharp/csharp-file>

[ ← 139 ]

[https://docs.microsoft.com/en-us/dotnet/api/system.io.file.appendtext?  
view=netframework-4.7.2](https://docs.microsoft.com/en-us/dotnet/api/system.io.file.appendtext?view=netframework-4.7.2)

[ ← 140 ]

[https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?  
view=netframework-4.7.2](https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.7.2)

[[← 141](#)]

[https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?  
view=netframework-4.7.2](https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.7.2)

[[← 142](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_delegates.htm](https://www.tutorialspoint.com/csharp/csharp_delegates.htm)

[ ← 143 ]

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/>

[[← 144](#)]

[https://www.akadia.com/services/dotnet\\_delegates\\_and\\_events.html](https://www.akadia.com/services/dotnet_delegates_and_events.html)

[[← 145](#)]

[https://www.tutorialspoint.com/csharp/csharp\\_multithreading.htm](https://www.tutorialspoint.com/csharp/csharp_multithreading.htm)

[ ← 146 ]

<https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading>

[ ← 147 ]

<https://o7planning.org/en/10553/csharp-multithreading-programming-tutorial>

[ ← 148 ]

[https://www.tutorialspoint.com/csharp/csharp\\_events.htm](https://www.tutorialspoint.com/csharp/csharp_events.htm)

[[← 149](#)]

<https://www.tutorialsteacher.com/csharp/csharp-event>

# Table of Contents

[Introduction](#)

[Chapter 1: Getting Started](#)

[Why Python?](#)

[Installing Python](#)

[Summary](#)

[Chapter 2: Software Design Cycle](#)

[Solving Problems](#)

[Summary](#)

[Chapter 3: Variables and Data Types](#)

[Python Identifiers](#)

[Introduction to Variables](#)

[Python and Dynamic Typing](#)

[Basic Text Operations](#)

[Numbers](#)

[Summary](#)

[Chapter 4: Decision Making in Python](#)

[Conditional Statements](#)

[“While” Loops](#)

[“For” Loops](#)

[Summary](#)

[Chapter 5: Python Data Structures](#)

[Sequences](#)

[Creating a Basic Game](#)

[Summary](#)

[Chapter 6: Functions](#)

[Creating Functions](#)

[Global Variables](#)

[Handling Global Variables](#)

[Writing Tic Tac Toe](#)

[Summary](#)

[Chapter 7: Introduction to Object Oriented Programming](#)

[Classes](#)

[Python Inheritance](#)

[Polymorphism](#)

[Creating Modules](#)

[Summary](#)

[Chapter 8: Exceptions](#)

[When Things Go Haywire](#)

[Summary](#)

[Bibliography](#)