Prince of Programming

Beginner's Guide

A.C. THOMAS

ISBN: 978-1-0674563-0-6
First Edition

Overview

Table of Contents

Bonus I

Chapter 5

Chapter 7

Chapter 8

Chapter 9

Bonus III

Introduction

"Everyone should learn to code because it teaches you how to think." — Steve Jobs

About This Book

This book teaches you how to program with the help of C#. Unlike many programming books that try to be "complete" and cover as much as possible we removed all unnecessary clutter and only included what someone learning how to code need to know.

You'll learn the core fundamentals, the essential 20% of concepts that handle 80% of real-world programming. Each concept comes with clear code examples, detailed comments, and expected output right on the page, so you can learn anywhere. On the bus, at the beach, or lying in bed and know what to expect when you write code. Because learning should be fun, not a march through boring source code. Also, the examples are game-themed with a back story, so imagine a text-based game where a captured Prince in a dungeon is trying to escape and rescue his Princess.

Chapters build naturally on one another, one concept at a time, explained clearly, followed by practice. Examples use simple console applications to remove visual distractions, letting you focus on the code. The skills you learn are the same ones used in professional apps, enterprise systems, and even games, you're just learning them without the noise.

By the end, you'll understand the fundamentals every programmer relies on: variables, logic, loops, functions, classes and proper program structure. You'll have a strong foundation, the ability to continue learning on your own, and the mindset to think like a programmer. Breaking problems into pieces, debugging effectively, and building software that work.

Who This Book Is For

I originally set out to find something I could use to help teach my son how to program. After searching, I realized there really wasn't a resource that focused on strong fundamentals while still being fun and engaging. So, I decided to create one.

You don't need any prior programming experience to use this book. If you have never written a line of code, or are not even sure what code really is, you are exactly who this book was written for. We start at the beginning and assume nothing. You will learn step by step, focusing on understanding why things work instead of just copying code and hoping for the best. Concepts are introduced one at a time, helping you move from "Why are we doing this?" to "Oh, that actually makes sense."

This book is beginner friendly, but it is also valuable for intermediate developers. If you taught yourself programming and have already built a few projects, this book will help strengthen your foundation and sharpen the way you think about problem solving and code structure.

You don't need advanced math skills or an expensive computer to get started. What you do need is curiosity, a little patience when things don't work the first time, and the willingness to keep going. Those qualities matter far more than any technical background.

What You'll Be Able to Do by the End

By the time you finish this book, you'll have something more valuable than a list of memorized syntax, you'll understand how programmers think. When you face a problem, you'll automatically start breaking it down into smaller, manageable pieces. When your code doesn't work (and it won't, sometimes), you'll know how to investigate systematically instead of randomly changing things and hoping for the best. You will know where to start and how to structure your approach.

You will understand the core principles that underpin all programming, regardless of language: how to store and manipulate information, how to make programs respond intelligently to different situations, how to avoid repeating yourself, and how to organize code so it stays manageable as it grows. These aren't all C#-specific tricks, they're fundamental concepts that translate to any programming language or platform you might work with in the future.

More practically, you'll be able to write real, working programs. Useful tools that solve actual problems. You'll be comfortable reading other people's code and understanding what it does. You'll know how to search for solutions when you get stuck (a critical skill that professionals use daily). And you'll have the foundation needed to dive into any specialization you're interested in: web applications, game development, mobile apps, data analysis, whatever catches your interest.

Why C#?

Why not Python, which some might say is "easier"? Or JavaScript, which runs in browsers? Or C++, used for high-performance games?

C# hits the sweet spot. It's powerful enough to build professional software. You'll find it at many enterprise companies use it. It's also clean and readable, without the confusing syntax that makes some languages hard to learn.

It's strongly typed, which means you tell the computer what kind of data you're working with. That might sound strict at first, but it actually helps catch mistakes early. Think of it as guardrails while you're learning, keeping you away from mysterious errors later.

C# is incredibly versatile. Desktop apps? Web apps? Mobile apps with Xamarin? Games? Some game engines like Unity, Godot, Stride and MonoGame support it. Plus, C# is cross-platform, running on Windows, macOS, and Linux, so your skills aren't tied to a single operating system.

C# tools are excellent! Visual Studio and Visual Studio Code are free, powerful, and help you learn as you go by catching errors and suggesting fixes.

Finally, once you know C#, picking up other languages is much easier. The syntax may change slightly, but the core concepts stay the same. Learn C# well, and you'll have skills that transfer across the programming world.

Acknowledgments

Writing this book has been a long and rewarding process, and I am thankful to everyone who helped make it possible.

First and foremost, I want to thank my wonderful wife. Her love, support, and encouragement kept me going through the late nights. Her belief in me and in this project made all the difference.

To my children, thank you for putting up with the hours I spent working instead of playing co-op games with you. Your energy, curiosity, and enthusiasm reminded me why this work matters. My son helped me shape many of the programming concepts in this book, and my daughters, whose hugs and cheerful presence provided just the right motivation when I needed it.

This book is for all of you, and because of all of you.

About the Author

I started programming when I was about ten years old, piecing together computers from scavenged spare parts and teaching myself how to make them do something useful. What began as curiosity quickly turned into a lifelong passion for building things and figuring out how they work.

After earning a degree in Information Technology, I spent my career working with a variety of software companies, learning new technologies and programming languages along the way. I've worked on everything from GIS systems and web applications to flight procedure design software and large-scale enterprise SaaS microservices. Every project brought new challenges, and every challenge was an opportunity to learn something new.

Along the way, while leading many teams, I discovered that I enjoy helping others grow just as much as I enjoy writing code. I've mentored many developers, reviewed more pull requests than I can count, and learned that the best software is built when people learn together. This book comes from a long-held desire to give back, share what I've learned, and make the journey of learning to program a little more enjoyable.

Resources for This Book

All source code and supplemental materials for this book are available online.

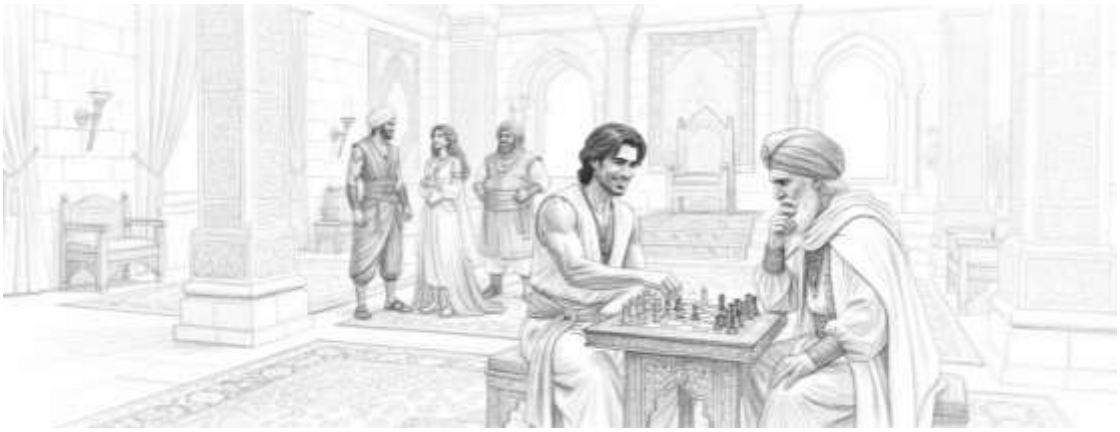
Website: https://www.PrinceOfProgramming.com
Source Code: https://github.com/PrinceOfProgrammingBook/BeginnersGuide

The website may contain updates, errata, and additional resources. The GitHub repository includes all code examples from the book, organized by chapter.

These resources may evolve over time to reflect updates, fixes, and improvements.

Before You Start

"The best way to predict the future is to invent it." — Alan Kay



Getting Ready for Your Journey

Welcome to the training grounds. Before you jump into writing code, let's take a few minutes to understand how this book works. Think of this as the tutorial level, the part where you learn the controls before facing your first spike trap. Knowing how to navigate this book will make your learning journey smoother and help you get the most out of every chapter.

This isn't the kind of book where you absolutely must read every word in order (though that's definitely a good way to do it). Some sections you'll breeze through, others you'll want to spend more time on. That's normal. This chapter explains how the book is structured, what all those icons and sidebars mean, and when it's okay to move forward even if you don't understand everything perfectly yet.

Don’t forget about the **Glossary** and **Index** at the back, they’re there to make your life easier. The glossary gives quick, plain-language explanations of terms you might want to revisit, while the index helps you jump straight to the exact topic, concept, or keyword you’re looking for without flipping through chapters. Whether you’re stuck on a term, refreshing your memory, or hunting for that one example you *know* you saw earlier, these sections can save you time and frustration.

Let's get you oriented.

The Story of Our Prince

You're probably wondering why the book is named *"Prince of Programming"*. Everyone loves a good adventure right, so why not use coding examples that follow the story of a Prince on an adventure?

Here's our back story for our Prince:

(Inspired by Classic Arabian Nights Tales)

In an ancient Arabian desert kingdom, the sultan's palace, intrigue and dark magic, a young prince finds himself trapped in a deadly struggle for love and power. He is in love with the Sultan's daughter, but their future is threatened by the Sultan's most trusted advisor, the Evil Vizier, a sorcerer determined to claim the throne for himself.

When the Vizier learns that the prince could marry the princess and ruin his plans, he acts swiftly. Using deception and dark magic, he has the prince captured and thrown into a vast dungeon hidden beneath the palace. There, the Vizier intends to keep him imprisoned until he can force the princess into marriage and secure his rule.

The dungeon is a brutal maze of twisting corridors, patrolling guards, and deadly traps: spike pits, swinging blades, and heavy metal gates controlled by ancient switches. To survive, the prince must run, jump, climb, and carefully plan every move where one mistake could be fatal. But it's not all death and gloom, hidden are some magic potions to help the Prince when he's injured and somewhere the Prince can find a sword he will need to fight his way back to save the Princess.

This dungeon becomes the backdrop for the examples throughout the book. The story will continue at the end of each chapter before you get to the fun experiment. As the prince navigates traps and puzzles, you'll practice the same skills needed to learn programming: thinking clearly, solving problems, handling unexpected situations, and choosing the right actions at the right time.

Imagine a text-based game with a mix of Aladdin and Indiana Jones: racing against a ticking clock, navigating deadly traps with precise timing. That's the vibe we're going for in this book, learning to program while having a little adventure along the way.

How to Read This Book

The Basic Structure

Each chapter opens with an inspiring quote and an illustration of the Prince in action, setting the tone for what's ahead. This is followed by a brief introduction explaining what you'll learn and why it matters, then a table listing the core programming concepts covered. The main content teaches these concepts clearly with code examples, comments, and expected output, building from simple to complex. At the end, you'll find a Fun Experiment which is a small coding challenge to practice what you just learned (with a sample solution provided).

Icons and Sidebars

Trivia Block

Trivia:

Before diving into the actual content, every chapter includes a trivia block with fascinating facts that loosely connect to what you're about to learn. These aren't random, they're chosen to spark your curiosity and get your brain warmed up.

Pro Tip Block

Pro Tip:

"Pro Tips" contain helpful information that goes beyond the basics. These might explain why professionals do something a certain way, share common shortcuts, or give you some insight into how something works under the hood. You don't need to memorize Pro Tips to move forward, but they'll deepen your understanding.

Throughout the content, you'll see various icons highlighting specific types of information.

Here's what each one means:

Do this:

When you see this icon, pay attention. It's highlighting the correct way to do something. The best practice, the recommended approach, or a technique you should make a habit of using. Think of these as the right moves in a sword fight.

Don't do this:

This icon warns you about common mistakes or bad habits to avoid. These are the traps and pitfalls that trip up beginners (and sometimes experienced programmers too). Learning what not to do is just as important as learning what to do.

Important:

When you see this icon, stop and make sure you understand what it's saying. These are critical concepts, common sources of confusion, or important warnings. Missing these can lead to frustration later.

Code Examples and Output

Every code example in this book follows the same format to make it easy to identify, read and understand:

The code itself is in a clearly marked block, with comments (lines starting with `//`) explaining what each part does:

```
// This is a comment explaining what the code does
int princeHealth = 100;                  // The Prince starts with full health
princeHealth = princeHealth - 20;        // He takes damage from a spike trap
Console.WriteLine("Prince health: " + princeHealth);
```

Below that, you will usually see the expected output. This is what you should see when you run the code. This means you can follow along and understand what's happening even if you're not sitting at a computer.

Expected Output:

```
Prince health: 80
```

When you see code samples, don't just skim over them quickly as something only a computer needs to care about. The names used for variables and methods combined with the comments are part of the explanation and is written in such a way that it's easy to read and understand. Code comments usually start with double forward slashes "`//`", if you see them, read them!

Read Front to Back vs. Skip Around

The short answer: Start from the beginning and work your way through in order, at least for your first read.

The longer answer: This book is designed to build knowledge progressively. Each chapter assumes you've understood the previous ones. If you skip Chapter 3 and jump to Chapter 7, you'll likely encounter concepts that were explained earlier, and you might get a bit lost if you don't know those concepts already.

That said, programming isn't a perfectly linear journey. Sometimes you'll need to review an earlier concept, and that's completely normal. The structure of this book makes it easy to flip back and refresh your memory on specific topics.

Here's a good approach

- **First time through:** Read chapters in order, front to back. Work through the examples, try the Fun Experiments, and don't skip ahead until you understand the current chapter.
- **When reviewing:** Use the Core Concepts tables at the start of each chapter to quickly find what you need. The Glossary of the book at the end are also great for quick reference.

- **When stuck:** If something in a later chapter doesn't make sense, there's a good chance you need to review an earlier concept. Don't be afraid to go back, that's how we learn.
- **After you've finished once:** Feel free to skip around and use the book as a reference. By then, you'll have the foundation and context to jump to specific topics as needed.

When It's Okay to Move On

You don't need to understand everything 100% perfectly before moving to the next chapter!

Learning to program is like learning to ride a bike. You can read about balance and momentum and steering all you want, but the real understanding comes from actually doing it. Sometimes a concept will not fully click until you have seen it used in a few different ways, or until you have tried (and failed) to use it yourself a few times.

You should move on when:

- You understand the main concept, even if some details are fuzzy
- You can follow the code examples and predict what they'll do (roughly)
- You've attempted the Fun Experiment, even if your solution wasn't perfect
- You feel like you're spending more time confused than learning

You should not move on when:

- The code examples look like complete gibberish
- You can't explain the main concept in your own words at all
- You skipped the previous chapter entirely
- Everything from the last three chapters is still a mystery

Think of it this way: if our Prince story was a real game when the user needs the Prince to jump across a pit, he doesn't need to understand the physics of projectile motion. The user just needs to know when to press the jump button and roughly how far he'll go. The deeper understanding comes with practice.

If you find yourself stuck on a concept for too long, sometimes the best thing to do is mark it with a bookmark, move forward, and come back to it later. Often, seeing how something is used in the next chapter will make the previous concept suddenly click.

 **Important:**

If you find yourself completely lost multiple chapters in a row, stop and go back. There's a foundational concept you missed somewhere, and moving forward will only make things worse. It's better to spend an extra day on Chapter 3 until it makes sense than to muddle through to Chapter 10 understanding nothing.

Pro Tip:

Keep a notebook (physical or digital) while you read. When something doesn't quite make sense, write it down. Often, the act of writing the question helps you understand it better. And if it doesn't, you'll have a list of specific questions to revisit or ask someone about later.

That's it for the ground rules. You know how the book is structured, what the icons mean, and how to approach your reading. Now let's get into the actual programming. The dungeon awaits.

Final Note About Debugging

As you work through this book, you'll be writing code and running examples. Most of the time, if you follow the examples carefully, everything will work as expected. But here's the reality of programming: sometimes things don't work, and you'll need to figure out why.

Maybe you made a small typo, or get curious and try changing something to see what happens. Maybe you'll start experimenting with your own ideas. When that happens (and it will), your code might not behave the way you expect. That's completely normal and actually a sign you're learning!

When you run into problems and need help figuring out what's going wrong, flip to "**Appendix A - Debugging"**. This chapter is packed with techniques for investigating and fixing issues in your code. You don't need to read it right now; it will make more sense once you've written some code and understand the basics. But keep it in mind as your troubleshooting guide.

Think of **Appendix A** as your emergency toolkit. You don't need to carry it with you everywhere, but when something breaks, you'll be glad you know where it is.

For now, just follow along with the examples in each chapter. If something doesn't work and you can't figure out why, that's when you visit the debugging appendix. By the end of this book, when you start writing your own programs from scratch, those debugging skills will become essential tools you'll use every day.

Getting Started

"Give me six hours to chop down a tree and I will spend the first four sharpening the axe." — Abraham Lincoln

Before you can bring your ideas to life in code, you need the right tools and knowledge to get started. This chapter is all about setting up your programming environment and understanding the fundamental workflow that every developer uses: writing code, compiling it into something your computer can execute, and running it to see the results. Whether you're building the next viral mobile app, creating automation tools that save hours of repetitive work, or developing the backend systems that power websites visited by millions, this same basic process applies. Every piece of software you've ever used, from your web browser to your favorite music streaming app all started exactly where you are now, with a developer setting up their tools and writing their first lines of code.

You'll also learn one of the most crucial skills in programming: how to read and fix errors. Professional developers don't write perfect code on the first try. They write code, encounter errors, read what went wrong, and fix it. This cycle happens hundreds of times a day, and mastering it is what separates someone who can write code from someone who can build real software. By the end of this chapter, you'll have a working development environment, you'll understand how your code transforms into a running program, and you'll know how to troubleshoot the inevitable mistakes that every programmer makes. These fundamentals will serve you whether you're coding for fun, building tools for yourself, or starting a career in software development.

Core Concepts Covered

- ✓ Setting up a professional development environment
- ✓ Understanding the software development workflow
- ✓ The compilation process and how code becomes executable programs
- ✓ Reading and interpreting error messages
- ✓ Debugging and troubleshooting techniques
- ✓ File systems and project organization
- ✓ The relationship between source code and compiled applications

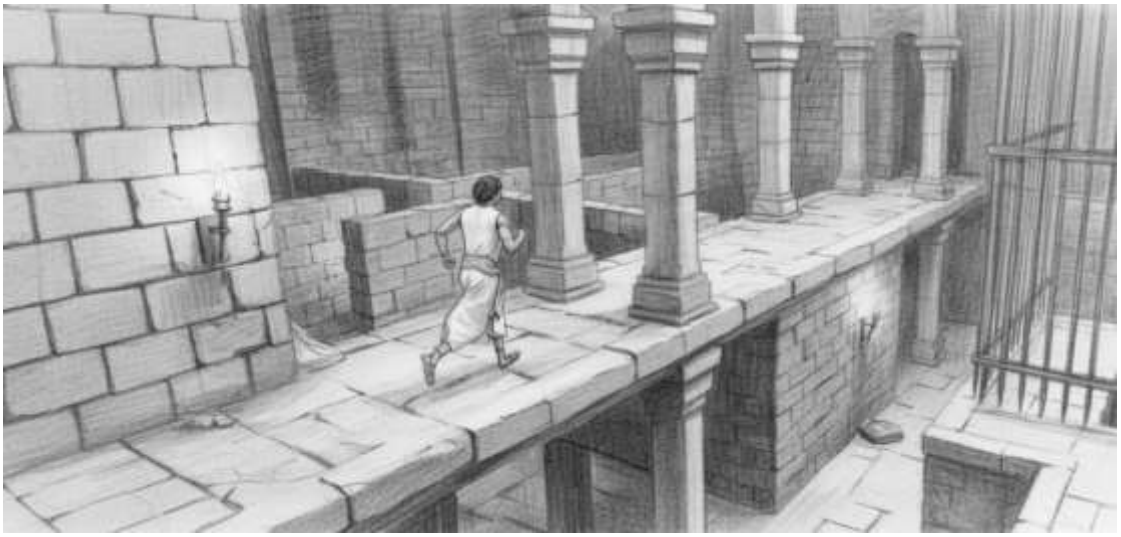
This is a preview sample.

The rest of the Getting Started Front Matter Section was removed from this sample document.

More of the sample can be found on the next page.

Code structure

"First, solve the problem. Then, write the code." — John Johnson



Welcome to your first real look at code! Before you can tell a computer what to do, you need to understand how to speak its language. Just like how English has rules about sentences, paragraphs, and punctuation, programming languages have their own structure. In this chapter, you'll learn how C# code is organized, where things go, why they go there, and how to make your code readable not just for the computer, but for other humans (including future you). Every piece of software you've ever used, from your favorite apps to the operating system running on your phone, follows these same fundamental organizational principles. Whether someone's building a music streaming service, a photo editing app, or a weather forecasting system, they all start with the same basic code structure you're about to learn.

You'll also discover why programmers are obsessed with things like formatting, naming, and comments. It might seem picky at first, but here's the truth: code is read far more often than it's written. Professional developers spend most of their time reading and understanding existing code, not writing new code from scratch. A well-structured file with clear names and helpful comments can mean the difference between fixing a bug in five minutes or spending hours trying to figure out what's going on. The habits you build now like organizing your code cleanly, choosing descriptive names, and writing thoughtful comments will save you countless headaches down the road and make you a better programmer from day one.

Core Concepts Covered

- ✓ How programming languages organize code into hierarchical structures
- ✓ Reading and executing code sequentially from top to bottom
- ✓ Using visual formatting and indentation to communicate code organization
- ✓ Documenting code intent for human readers while the computer ignores it
- ✓ Following naming conventions to make code predictable and professional
- ✓ Understanding the difference between code that runs and code that explains

The Anatomy of Code

Trivia:

The classic side-scroller Prince of Persia written by Jordan Mechner in 1989 was coded mostly in Assembly, which is like programming with stone tools, and yet the result blew minds. You, on the other hand, have **C#,** a modern, shiny sword compared to that old stick he used. So, let's start your own hero's journey into code!

Alright, let's crack open a C# file and see what makes it tick. Think of code like a recipe, but instead of making cookies, you're telling the computer exactly what to do. And just like how recipes have a specific format (ingredients first, then instructions), code has a structure too.

Here's the most basic C# program, the legendary "Hello World" but slightly modified for our own Prince's story:

```
using System;

namespace PrinceOfProgramming
{
    class Game
    {
        static void Main(string[] args)
        {
            Console.WriteLine("The Prince awakens...");
        }
    }
}
```

Expected Output:

```
The Prince awakens...
```

Now, I know what you're thinking: "That's a lot of weird words just to say 'The Prince awakens...'" You're not wrong! But each part has a purpose. Let's break it down piece by piece.

Breaking Down the Code Palace

The `using` Directive

```
using System;
```

Think of **`using`** like opening your toolbox before starting a project. **`System`** is a collection of pre-built tools (we call them "libraries") that Microsoft gives you for free. In this case, we need **`System`** because it contains **`Console`**, which lets us write text to that black screen (we'll get to that in a bit).

In game terms, imagine you're about to start building a level for your Prince. You'd need access to your sprite editor, your tile map tools, and your animation frames. The **`using`** statement is like saying, "Hey, I'm going to need my animation tools for this level."

The `namespace` Keyword

```
namespace PrinceOfProgramming
{
    // Everything else goes in here
}
```

A namespace is like a folder that keeps your code organized. When game developers work on a game, they usually have different folders for enemy sprites, player animations, and level designs. A namespace does the same thing in code, it groups related stuff together.

Namespaces help prevent confusion. Imagine you create something called **`Timer`** for your program, but Microsoft also has something called **`Timer`**. How does the computer know which one you mean? Namespaces solve this problem by giving everything a "last name":

```
using System.Timers.Timer;        // Microsoft's Timer
using MyGame.Timer;               // Your Timer
```

For now, you can name it whatever you want. I called mine **`PrinceOfProgramming`** because, well, we're learning to program while thinking about that awesome Prince.

Notice those curly braces: **`{ }`**? They're super important! Everything between an opening "**`{`**" and closing "**`}`**" belongs together. Think of them like the walls of a room where everything inside the room is part of that room.

Pro Tip:

The standard convention is to name namespaces after your project or company. For example, Microsoft uses **`Microsoft.Something`**, while a company's inventory software might use **`MyCorp.Inventory`**. This helps everyone know where code came from.

The `class` Keyword

```
class Game
{
    // Your code goes here
}
```

A class is like a blueprint. In our Prince game we should have a blueprint for "Guard" to define how guards look, how they move, how much health they have, and what happens when they get hit by your sword. Every guard in our game is then created from that same blueprint. More about classes later.

For now, just know that **`class Game`** is our main blueprint, and we'll put all our starting code inside it.

Pro Tip:

Class names should start with a capital letter and describe what the class represents. **`Game`**, **`Player`**, **`Enemy`**, and **`TreasureChest`** are all good class names. **`thing`**, **`stuff`**, or **`code1`** are terrible names. Future you will thank you for using good names.

The `Main` Method

```
static void Main(string[] args)
{
    Console.WriteLine("The Prince awakens...");
}
```

Here's the big one: **`Main`** is where your program starts. Every C# program needs exactly one **`Main`** method, and when you run your program, the computer says, "Okay, where's the **`Main`** method?" and starts there.

Think of **`Main`** as the "START" button on any program. When you boot up a game, the game doesn't randomly jump to level 7, it starts at the beginning and runs in order. Same with your code!

Don't worry too much about **`static void`** and **`string[] args`** right now. For now, just know they need to be there. We'll cover what they mean later when it makes more sense.

Inside **`Main`**, we have:

```
Console.WriteLine("The Prince awakens...");
```

`Console.WriteLine` is a command that writes text to the console (that black screen where your program runs) and then moves to the next line. Whatever you put inside the quotation marks will appear on the screen. It's like the text boxes in old adventure games!

Top to Bottom: The Code Execution Highway

Here's a crucial concept: **code runs from top to bottom, one line at a time**, just like reading a book. The computer doesn't skip around unless you specifically tell it to.

Let's see this in action:

```
namespace PrinceOfProgramming
{
    class Game
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Level 1: The Prison");
            Console.WriteLine("You pull the lever...");
            Console.WriteLine("The gate slowly opens.");
            Console.WriteLine("You step forward into the darkness.");
        }
    }
}
```

Expected Output:

```
Level 1: The Prison
You pull the lever...
The gate slowly opens.
You step forward into the darkness.
```

As we can see from the output Line 1, then line 2, then line 3, then line 4. Always starting at the top and going down in sequence.

This is how all programs and games that to react to user input work. For example, every frame a game need to:

1. Check for player input
2. Update the Prince's position
3. Check for collisions
4. Update enemy positions
5. Draw everything on screen
6. Wait until it's time for the next frame
7. Repeat

Your code works the same way, step by step and in order.

Pro Tip:

The computer is incredibly literal. It does *exactly* what you tell it to do, in *exactly* the order you tell it. If your code isn't working, 99% of the time it's because you told the computer to do the wrong thing, not because the computer messed up. This is actually good news—it means bugs are fixable!

Curly Braces: The Indentation Dance

Curly braces are your code's fences. They show exactly where things start and end. I personally like lining up my opening and closing braces so they're easy to follow. Other styles exist, and that's okay, but clear code is always better than clever-looking code:

```
namespace PrinceOfProgramming
{
    class Game
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Good alignment!");
        }
    }
}
```

See how each **{** has a matching **}** directly below it? And notice how everything inside is indented (pushed to the right)? This isn't just to look pretty, it helps you see the structure at a glance.

Here's what bad formatting looks like:

```
namespace PrinceOfProgramming { class Game { static void Main(string[] args) {
Console.WriteLine("This works but hurts to read"); } } }
```

Technically, this runs fine. The computer doesn't care about spacing. But humans? We care a LOT. Reading this is like trying to play a game when all the walls and floors look the same. Technically possible, but why would you do that to yourself?

Imagine you're building a trap mechanism. There are pressure plates that trigger spikes. In code, that might look like:

```
if (princeLandsOnPressurePlate)
{
    TriggerSpikes();
    PlaySound("spikes.wav");
    _princeHealth = _princeHealth - 1;
}
```

The braces show you that all three things happen *only* when the Prince lands on the pressure plate. Good formatting makes it obvious what belongs together.

Comments: Notes to Your Future Self

Comments are lines in your code that the computer completely ignores. They're for us humans, for you when you come back to your code later and think, "What was I doing here?". And yes, for other humans as well when you work in a team, so don't use comments that only you can understand.

There are two types of comments in C#:

Single-Line Comments

```
// This is a comment. The computer ignores this line completely.
Console.WriteLine("This runs!"); // You can also put comments at the end of lines
```

Everything after `//` on that line is ignored.

Multi-Line Comments

```
/*
This is a multi-line comment.
I can write as much as I want here.
The computer will ignore all of this.
Useful for longer explanations!
*/
Console.WriteLine("This runs!");
```

Everything between `/*` and `*/` is ignored, even if it spans multiple lines.

When to Use Comments

Comments are great for explaining *why* you did something, not *what* you did. Here's an example:

 **Bad comment:**

```
// Add 1 to health
health = health + 1;
```

We can already see you're adding 1 to health. The comment doesn't help.

 **Good comment:**

```
// The Prince starts with 3 health, but play testers kept dying on Level 1
// so we are secretly giving player 1 extra health here
health = health + 1;
```

Now *that's* useful information!

Where to Place Comments

Good comments live right above or next to the code they're explaining:

```
// Check if the prince landed on a spike trap
if (currentTileType == "spikes")
{
    princeHealth = princeHealth - 1;  // Ouch!
}
```

Avoid putting comments far away from what they explain, that will make things confusing when you're scrolling around. And when code gets modified it might even lose its position and then it's really confusing.

When NOT to Use Comments

Keep this in mind, the best code barely needs comments because it should explain itself. When you decide on the name for a variable, method or class, always try to give it a name that explains exactly what its purpose is.

Here we need a comment because code is unclear:

```
int t = 60;  // Time limit in seconds
```

Here the code is self-explanatory!

```
int timeLimitInSeconds = 60;
```

The second version doesn't need a comment because the variable name tells you everything! This is called "clean code", code that reads like English and require no comments to explain what it is or what's its doing.

Pro Tip:

If you find yourself writing many comments to explain what your code does, it might mean your code is too complicated. Try using better variable names or breaking complex code into smaller, simpler pieces. Comments should explain the "why," not the "what."

The Commenting-Out Trick

Sometimes you want to temporarily skip some code without deleting it. Just comment it out!

```
Console.WriteLine("Level 1 starts...");
// Console.WriteLine("Playing intro music..."); // Not ready yet, add this later
Console.WriteLine("The gate opens.");
```

When writing code this is super common. Sometimes you just want to test using a different method, just comment current method and add the new one. You can just remove the comments when you're done and the code will be like it was before. Or in our Prince's case, maybe you're testing a level and don't want to sit through the intro cut scene every time. Just comment it out temporarily!

Naming Conventions: The Rules of the Road

Programmers follow certain naming rules to keep things consistent. It's like how in English, we start sentences with capital letters, it's a convention that makes code easier to read.

PascalCase

Used for classes, methods, and namespaces. Every word starts with a capital letter and when it has multiple words then each new word is capitalized (you can't use spaced in names):

```
class EnemyGuard        // Good!
class enemyguard        // Nope
class enemy_guard       // Also nope for classes
```

camelCase

Used for variables (we'll cover these in the next chapter). First word is lowercase, then every word after starts with a capital:

```
int princeHealth        // Good!
int PrinceHealth        // Nope, that's PascalCase
int prince_health       // Nope, that's snake_case (used in Python, not C#)
```

Make Names Descriptive

Your code should read almost like English.

Compare these:

Bad naming

```
int h;
int e;
string n;
```

vs.

Good naming

```
int playerHealth;
int enemyHealth;
string playerName;
```

Which one would you rather come back to a few weeks later when you want to start using those variables?

Pro Tip:

If you find yourself adding numbers to variable names (like `guard1`, `guard2`, `guard3`), you're probably doing something wrong. There's usually a better way (we'll learn about collections later). The exception is if the numbers mean something specific, like `level1Boss` vs. `level2Boss`.

The Console: Your Testing Ground

Now let's talk about that "`black screen`" aka the console. You might be wondering, "Why are we using this boring text window. It would be so much nicer if we had good looking windows, buttons or controls, maybe even some graphics and sound like when we play as the Prince?"

Great question! The console is like a software developer's sketchpad with many uses.

What Is the Console?

The console (also called a terminal or command prompt) is a text-based window where your program can display output and receive input. It's been around since before graphical user interfaces existed. Remember those old green-text computer screens from 1980s movies.

```
Console.WriteLine("This text appears in the console!");
```

When you run this, a black window pops up with white text. Simple, but incredibly useful.

Why We Use the Console for Learning

Here's why well be using console programs:

1. **Zero Distractions**

   When you're learning the fundamentals like variables, loops and logic you want to focus on the *thinking* part, not on "Why is my sprite showing up in the wrong place?" or "How do I load this image file?"

   When writing a program, you want to make sure it does what it's intended to do. What's the use of having a window with some controls show up but the program crashes the moment you try to click on a button? In a real game for our Prince we don't need to start by drawing the Prince. We first start with logic like this: "If the player is falling, increase falling speed. If falling speed is too high, the player dies." That logic works the same whether it's displayed as fancy graphics or just text that says "You fell and died."

2. Instant Feedback

Want to see if your code works? Run it, and BOOM! instant text output. No need to compile textures, load sprite sheets, or set up a rendering pipeline. You get immediate results:

```
Console.WriteLine("Guard health: 5");
Console.WriteLine("You attack!");
Console.WriteLine("Guard health: 4");
```

Expected Output:

```
Guard health: 5
You attack!
Guard health: 4
```

See? You just simulated combat without drawing a single pixel.

3. Easy Debugging

When something goes wrong (and trust me, things *will* go wrong), the console makes it easy to see what's happening. You can print out values to check your work:

```
int princeHealth = 3;
// Check if it's what you expect
Console.WriteLine("Prince health is: " + princeHealth);
```

This is exactly how professional developers debug their code, even in massive enterprise software and you guessed it games. They print values to logs to see what's going on behind the scenes.

4. The Core Is the Same

Notice that the logic you write for console programs is *exactly the same* as the logic in graphical applications or games. Whether you display "Guard defeated!" in a console or show a fancy animation of a guard falling down, the underlying code is identical:

```
Console.WriteLine("Guard defeated!");
// Console version
// OR
// PlayDefeatAnimation();// Graphics version
```

The `if` statement, the comparison, the logic, it's all the same. We're just using text output instead of graphics for now.

5. Real Programs and even Games Started Here

Many complex software systems start with the console. Creating spikes (doing research and experiments) to see if what they are planning to do will work or not.

Even modern indie hits like Undertale and Hollow Knight started with developers testing their game logic in simple forms before adding the fancy graphics. Toby Fox (creator of

Undertale) prototyped combat mechanics and dialogue systems before ever drawing Flowey's creepy face.

Console vs. Windows vs. Graphics

Think of it this way:

Console Program:

- You enter the dungeon.
- Do you go left or right?
- You press the left button
- You found a health potion!

Application with graphics:

- Shows dungeon room with animated torches
- Displays two doorways
- Player clicks left door
- Shows potion pickup animation

Same logic, different presentation. We'll tinker with some graphics later in a bonus chapter, but for now, the console lets us focus on the thinking part of programming. And this is all you need to learn to program like a pro!

Pro Tip:

Many professional developers still write console programs for testing algorithms, processing data, or creating tools. Command-line utilities are everywhere in game development—build scripts, asset converters, level generators. Learning console programming isn't a step backward; it's learning a tool you'll use forever.

Fun Experiment: Chronicling the Journey

Time to put this knowledge to work!

The Prince's Journey: Finding His Voice

The story so far…

The Prince steadies himself, sword in hand, and begins to navigate the twisting corridors of the dungeon. Stone pillars loom like silent sentries, and passages branch off in every direction. Some leading deeper into darkness, others perhaps toward freedom.

As he moves cautiously forward, testing each step, he realizes something crucial: he needs to understand the layout of this place. Where did he start? Which passages has he already explored? Where might the exit be? Before he can map out an escape route or plan his next move, he must first learn to chronicle his journey, to create a record of his progress through this labyrinth.

He pauses in a quiet alcove, catches his breath, and decides to document the beginning of his tale. If nothing else, should he fall, perhaps someone will find his story and know that he tried.

The Challenge

Create a program that tells a short story about the Prince entering the first level of the dungeon. Use at least 5 `Console.WriteLine` statements, add some comments explaining parts of your story, and make sure your braces are properly aligned.

Try to include:

1. A title for your level
2. Some action happening
3. At least one comment explaining something interesting
4. Proper indentation and brace alignment

Give it a shot before looking at the example answer!

Example Answer

Here's one way to solve it:

```
using System;

namespace PrinceOfProgramming
{
    class Game
    {
        static void Main(string[] args)
        {
            // The beginning of the Prince's journey
            Console.WriteLine("=== LEVEL 1: THE DUNGEONS ===");
            Console.WriteLine("");

            Console.WriteLine("You land gracefully in a dark corridor.");
            Console.WriteLine("The stone floor is cold beneath your feet.");

            // In the actual game, this is where the famous running jump
            // animation would play
            Console.WriteLine("You take a running leap across a chasm...");
            Console.WriteLine("And barely catch the ledge on the other side!");

            Console.WriteLine("");
            Console.WriteLine("Your adventure begins now, Prince.");
        }
    }
}
```

Expected Output:

```
=== LEVEL 1: THE DUNGEONS ===

You land gracefully in a dark corridor.
The stone floor is cold beneath your feet.
You take a running leap across a chasm...
And barely catch the ledge on the other side!

Your adventure begins now, Prince.
```

Notice how the empty `Console.WriteLine("")` adds a blank line for spacing? That's a simple way to make your output more readable!

Wrap-Up: What You've Learned

Congratulations! You just learned the fundamental structure of C# code.

Key Takeaways

- **Code files have a specific structure** with `using` statements, namespaces, classes, and methods
- **The `using` statement lets you access tools** from .NET, external libraries, or your own namespaces
- **Namespaces organize your code** like folders and let you share classes between different files
- **The `Main` method is where every program starts:** It's your application's start button
- **Code runs from top to bottom**, one line at a time, just like reading a book
- **Curly braces `{ }` define scope** and group code together, and we always line them up vertically
- **Comments are for humans**, not computers. We use `//` for single lines and `/* */` for multiple lines
- **Good comments explain "why," not "what"** and clean code should be self-explanatory, not full of comments to try and tell what each variable stores.
- **Naming conventions matter**: We use *PascalCase* for classes and methods and *camelCase* for variables
- **Good names are descriptive,** for example `playerHealth` beats `h` every time
- **The console is your testing ground**, it lets you focus on logic without graphics getting in the way
- **Console programs teach the same skills** as graphical application or games, just without the visual complexity

Think of this as your "code map". Just like how most game levels have a layout, your code has a structure. Master this structure, and you have just picked up the sword you need to get to the next level!

Variables

"The measure of intelligence is the ability to change." — Albert Einstein

Imagine you're using Spotify and you press play on your favorite song. The app needs to remember which song you're listening to, how far into it you are, your volume level, whether shuffle is on, and dozens of other pieces of information all at the same time. Or think about Instagram: when you scroll through your feed, the app is constantly tracking which post you're viewing, how many likes it has, who posted it, and whether you've already liked it yourself. Every piece of software you use daily, be it your web browser keeping track of open tabs or your phone's calculator remembering your last calculation, they all rely on one fundamental concept: storing and managing information that can change.

This is what variables are all about. They're the foundation of all programming, the basic building blocks that let software remember things and work with data. Without variables, programs would be like a person with no memory, not able to track anything, respond to anything, or do anything useful. In this chapter, you'll learn how to create these memory containers, what types of information they can hold, how to work with different kinds of data, and the rules about where and when you can use them. Master variables, and you're well on your way to understanding how all software really works under the hood.

Core Concepts Covered

- ✓ Creating named containers to store information your program needs to remember
- ✓ Understanding different categories of data and choosing the right container for each
- ✓ Converting between different data types when needed
- ✓ Working with text and manipulating it in useful ways
- ✓ Getting input from users and making programs interactive
- ✓ Tracking time and creating time-based behaviors
- ✓ Organizing related values into meaningful groups
- ✓ Understanding where data exists and how long it lives in your program
- ✓ All types share a common foundation and how to work with any type of data

The Program's Brain Cells

Trivia:

Programmers in the 1960s-70s stored years using only two digits (like "99" for 1999) to save memory—every byte was precious. This seemed fine until people realized that on January 1, 2000, computers would think it was 1900. Banks might calculate you owe 100 years of interest, power grids might shut down, and planes might think their maintenance was overdue by a century. The world spent an estimated $300 billion updating variables from two digits to four digits.

Think about any game when you'd open a potion bottle and your health would instantly go from "basically dead" to "ready to fight guards again"? That potion didn't just magically appear, the game was tracking your health the entire time using something called a **variable**. Variables are basically labeled boxes where your program stores information it needs to remember.

Variables are the foundation of everything you'll ever code. Without them, your program would be like a game character with amnesia that is forgetting everything the moment it happens. Let's look at how these magical boxes work.

Different Data, Different Types

Think of variables as different sized containers for different types of stuff. You wouldn't store your sword in a potion bottle, right? Same deal with variables, different types is needed for different data.

Basic Variable Types

Type	What It Stores	Example Values	Common Uses
int	Whole numbers (integers)	42, -7, 0, 1000	Age, score, level number, enemy count
double	Decimal numbers	3.14, -2.5, 127.89	Pi, damage multiplier, precise positions
float	Decimal numbers (less precise)	5.5f, 1.25f	Quick calculations, less precise than double
bool	True or false	true, false	Is deleted? Is door locked? Is player alive?
char	Single character	'A', '5', '!'	Keyboard input, simple text symbols (In single quotes)
string	Text (multiple characters)	"Hello", "Prince"	Name, messages, dialogue (In double quotes)

These are the basic types you'll use most often. There are many more types and if you want to see them you can take a look in the Quick Reference chapter at the end of the book, but for now this is all you'll need.

Let's see them in action:

```
int playerHealth = 100;
int currentLevel = 3;
int enemiesDefeated = 0;

double playerSpeed = 5.5;
double damageMultiplier = 1.25;

float jumpHeight = 2.5f;

bool hasSword = true;
bool isAlive = true;
bool doorIsLocked = false;

char playerInput = 'A';
char rank = 'S';

string playerName = "Prince";
string message = "The door opens with a creak";
```

These just store values, so there is no output yet, but we'll use them soon!

In most games your health would be an **`int`**, whether you're hanging from a ledge would be a **`bool`**, and your exact position on screen might be a **`double`** for smooth movement.

Pro Tip:

Notice that `float` values have an `f` after the number (like `2.5f`)? That tells C# it's a float and not a double. Most of the time, just use `double`—it's more accurate and you don't need the `f`. Use `float` only when you're doing heavy 3D graphics and need to save memory for example.

Naming Things: The Hardest Problem in Programming

Here's the deal: you can name variables almost anything, but that doesn't mean you should. Let's compare:

Bad Names:

```
int x = 100;
int a = 3;
bool b = true;
double spd = 5.5;
string n = "Prince";
```

What does these mean? No clue. You'll forget in five minutes, and anyone else looking at your code will be completely lost. Even abbreviations like `spd` can be confusing, is it speed or spawned or maybe special damage?

Good Names:

```
int playerHealth = 100;
int currentLevel = 3;
bool hasSword = true;
double movementSpeed = 5.5;
string playerName = "Prince";
```

NOW we're talking! These names tell you exactly what they're storing. When you're debugging at 2 AM trying to figure out why the Prince keeps falling through the floor, you'll thank yourself for using clear names.

Naming Rules in C#:

- Start with a letter or underscore (not a number)
- Use camelCase for variables (firstWordLowercase, restCapitalized)
- No spaces (use camelCase instead: `playerHealth` not `player health`)
- Be descriptive but not ridiculously long (`enemyCount` is better than `theNumberOfEnemiesCurrentlyInTheRoom`)
- Avoid single letters except maybe for loops (which we'll cover later)

Imagine working on some code and seeing variable with names like this:

```
int x = 5;  // What is x? The prince's health? The level number? A secret?
```

vs.

```
int guardAttackDamage = 5;  // Ah! That's how much damage a guard's sword does!
```

See the difference? Good names make your code readable, which means you can actually understand what you wrote last week!

Pro Tip:

If you can't come up with a good name for a variable, that might mean you don't fully understand what it's supposed to do. Take a moment to think about its purpose—the right name will help you (and others) understand your code better.

Strings: Your Text Holder

Okay, time for one of the most useful types: **strings**. A string stores any text, words, sentences, messages, anything made of characters. Think of every message in any game ("Press Button to Continue") any dialogue box, those are all strings.

Creating Strings

Here's a few examples of how to set some string values. These do nothing more than just storing the text inside the string variable so there will be no output yet.

```
string playerName = "Prince";
string message = "The door opens with a creak";
string gameTitle = "Prince of Programming";
```

Concatenation (Joining Strings with +)

You can stick strings together using the + operator:

```
string playerName = "Prince";
string message = "Welcome, " + playerName + "!";
Console.WriteLine(message);
```

Expected Output:

```
Welcome, Prince!
```

This works, but it gets messy fast when you're combining lots of things:

```
int health = 100;
int level = 3;
string status = "Player: "+ playerName + " | Health: " + health + " | Level: " + level;
Console.WriteLine(status);
```

Expected Output:

```
Player: Prince | Health: 100 | Level: 3
```

See how cluttered that looks with all those plus signs and quotes? There's a better way called string interpolation.

String Interpolation (The Better Way)

Instead of all those plus signs, use string interpolation with **$** and curly braces:

```
string playerName = "Prince";
int playerHealth = 100;
int level = 3;
string status = $"Player: {playerName} | Health: {playerHealth} | Level: {level}";
Console.WriteLine(status);
```

Expected Output:

```
Player: Prince | Health: 100 | Level: 3
```

Way cleaner and easier to read, right? The **$** before the quotes tells C# "I'm going to put variables inside this string," and anything in **{}** gets replaced with the variable's value.

Here's another example of what you might see in our Prince's adventure:

```
int timeRemaining = 45;
string urgentMessage = $"You have {timeRemaining} minutes to save the princess!";
Console.WriteLine(urgentMessage);
```

Expected Output:

```
You have 45 minutes to save the princess!
```

Escaping Special Characters

Sometimes you need to include special characters in your strings. For example, we use quotes to tell C# that the next text is inside a string. But what if we want to have quotes inside our string? C# will think you are stopping the string and throw a syntax error.

In cases like these, and other we use the backslash **** as our escape character, to tell C# to use character as the actual text:

```
string dialogue = "The guard shouts, \"Stop right there!\"";
Console.WriteLine(dialogue);
```

Expected Output:

```
The guard shouts, "Stop right there!"
```

Here are some common escape sequences:

- **\"** - Double quote
- **\n** - New line (like pressing Enter)
- **\t** - Tab
- **** - Backslash itself

Example with new lines:

```
string gameOver = "GAME OVER\nYou were crushed by spikes\nBetter luck next time!";
Console.WriteLine(gameOver);
```

Expected Output:

```
GAME OVER
You were crushed by spikes
Better luck next time!
```

You can also create multi-line strings that don't need all the special new line characters using the **@** symbol (called a verbatim string):

```
string asciiArt = @"
    /\
   /  \
  / /\ \
 / ____ \
/_/    \_\
";
Console.WriteLine(asciiArt);
```

Expected Output:

```
    /\
   /  \
  / /\ \
 / ____ \
/_/    \_\
```

Common String Methods

Strings come with built-in superpowers (methods) that let you manipulate them.

To use them just add a dot (`.`) to the end of the string variable name and the helper method you want to use:

```
string weapon = "Sword";

// Make it LOUD (Uppercase)
string loud = weapon.ToUpper();
Console.WriteLine(loud);

// Make it quiet (Lowercase)
string quiet = weapon.ToLower();
Console.WriteLine(quiet);

// How long is it?
int length = weapon.Length;
Console.WriteLine($"The word is {length} characters long");

// Check if it contains something
bool hasSword = weapon.Contains("Sword");
Console.WriteLine($"Has sword: {hasSword}");

// Replace parts of it
string newWeapon = weapon.Replace("Sword", "Dagger");
Console.WriteLine(newWeapon);

// Remove extra spaces
string messy = "  Prince  ";
string clean = messy.Trim();
Console.WriteLine($"Before: '{messy}'");
Console.WriteLine($"After: '{clean}'");
```

Expected Output

```
SWORD
sword
The word is 5 characters long
Has sword: True
Dagger
Before: '  Prince  '
After: 'Prince'
```

These methods are incredibly useful. Imagine checking if a player typed "NORTH" or "north" or "North". If you use normal C# comparison it will see the difference in casing (upper-case and lower-case letters).

For all these you can use `.ToLower()` to make them all the same before comparing:

```
Console.WriteLine("Which direction? (North, South, East, West)");
string input = Console.ReadLine();
string direction = input.ToLower();

if (direction == "north")
{
    Console.WriteLine("You head north into the darkness...");
}
```

Expected Output: *if you type "NORTH" or "North" or "north")*

```
Which direction? (North, South, East, West)
NORTH
You head north into the darkness...
```

Well look into the `if` statement later but it just checks if a condition is true. In this case it checks if two values are the same using the `==` equals to operator. More about this in the Logic chapter!

Pro Tip:

Strings in C# are **immutable**, which means once you create them, they can't be changed. When you do `weapon.ToUpper()`, it doesn't change the original—it creates a NEW string. This is actually a good thing for preventing bugs, but if you need to build up a long string piece by piece (like generating a complex game map), look into `StringBuilder` later on.

This is a preview sample.

The Rest of this chapter and the book was removed from this sample document.

Index