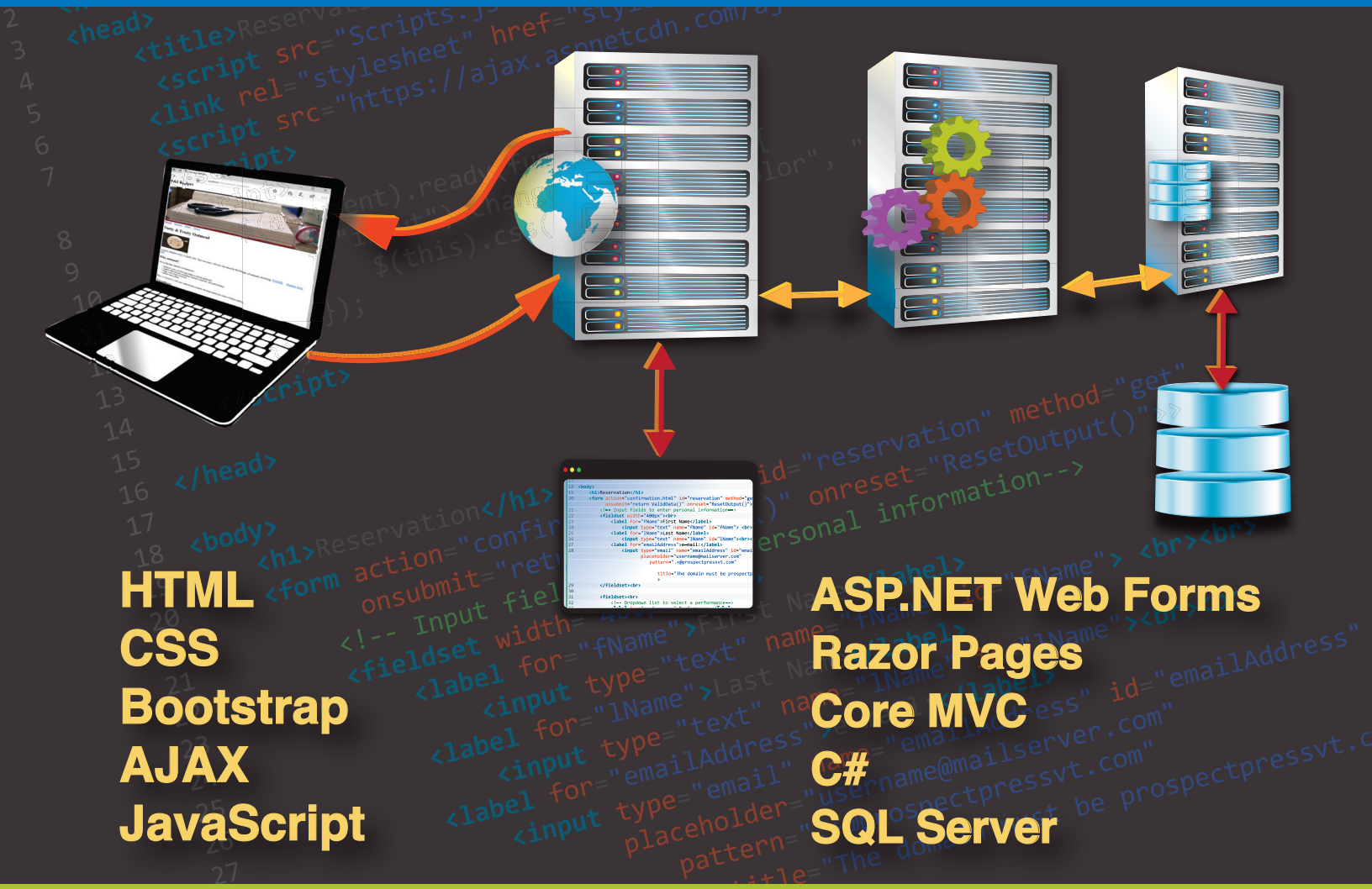


Web Development for Business

A Complete Approach to Front-End and Server-Side Development

George C. Philip | Jakob H. Iversen

Prospect Press

WEB DEVELOPMENT FOR BUSINESS

A Complete Approach to Front-End and Server-Side Development

APPENDICES A-E

George C. Philip

Jakob H. Iversen

The University of Wisconsin Oshkosh

Prospect Press

Founded in 2014, Prospect Press serves the academic discipline of Information Systems by publishing innovative textbooks across the curriculum including introductory, core, emerging, and upper level courses. Prospect Press offers reasonable prices by selling directly to students. Prospect Press provides tight relationships between authors, publisher, and adopters that many larger publishers are unable to offer. Based in Burlington, Vermont, Prospect Press distributes titles worldwide. We welcome new authors to send proposals or inquiries to Beth.Golub@ProspectPressVT.com.

Editor: Beth Lang Golub
Production Management: Rachel Paul
ePub Conversion: Scribe Inc.
Cover Design: Annie Clark

Web Development for Business
eTextbook
- Edition 1.0
- ISBN: 978-1-943153-89-3
- Available from RedShelf.com and VitalSource.com

Printed Paperback
- Edition 1.0
- ISBN 978-1-943153-93-0
- Available from RedShelf.com

For more information, visit https://www.prospectpressvt.com/textbooks/philip-web-development

Contents

Appendix A HTML Elements 564

Appendix B Data Organization and SQL 566

Appendix C Creating a Database 574

Appendix D Object-Oriented Programming 579

Appendix E C# Programming Using Visual Studio Code and Visual Studio Community 588

Appendix A

HTML Elements

Tag	**Description**
<!-- -->	Comment
<!DOCTYPE>	Document type
<a>	Hyperlink
<audio>	Audio
<b>	Bold
<body>	Document's body
 	Line break
<button>	Clickable button
	A generic section that groups elements
	Emphasized text
<form>	HTML form
<h1> to <h6>	Headings
<head>	Information about the document
<html>	Root element that contains all other elements
<i>	Italics
<img>	Image
<input>	Control that accepts user input
<label>	Label for an <input> element
	List item
<link>	Link to external resources like style sheets

<menu>	List of commands
<menuitem>	Menu item
<meta>	Metadata about HTML document
<nav>	A section of navigation links
object	Embedded objects like video, PDF, and Flash
	Ordered list
<option>	Option in a dropdown list
<output>	Result of a calculation
<p>	Paragraph
<param>	Parameter for an <object> element
<pre>	Preformatted text
<script>	Client-side script
<section>	Section in a document
<select>	Drop-down list
<small>	Makes the text one size smaller
<source>	Resources for media elements
	Group of in-line elements
	Important text
<style>	Style information
<table>	Table
<td>	Cell in a table
<th>	Header cell in a table
<time>	Date/time
<title>	Title for the document
<tr>	Row in a table
<var>	Variable
<video>	Video or movie

Appendix B

Data Organization and SQL

Accessing data from a database requires an understanding of how data is organized in the database. Almost all business applications use a type of database called *relational database*, which organizes data into tables and uses data to link different related tables. In this appendix, you will access data from a relational database using SQL (Structured Query Language), which is an easy-to-learn language universally used to maintain and query data stored in relational databases. Microsoft SQL Server, Oracle, MySQL, and Access are examples of relational databases. First, we will do a brief overview of organizing data into tables.

B.1 Organizing Data into Multiple Tables

Typically, the data used in web projects consists of different groups of related data items. For example, an order processing system may include data on customers, orders, and products. Minimizing data duplication and inconsistency, requires that each group of data is stored in a separate table, as shown here:

Customer (*CustId,* CustName, Street, Zip, …)
OrderHeader (*Order#*, OrderDate, ShipMethod, **CustId**)
Product (*Prod#*, ProdName, UnitPrice)
OrderLine (*OrderLine#*, **Order#**, **Prod#,** OrderQty)

Each table in a database represents an entity, and the fields represent the attributes of the entity.

Let's look at the key characteristics of a well-designed table. Each table should have a **primary key**, which is a field that represents a unique identifier of the entity. That means each record has a unique value for the primary key; no two records can have the same value for the primary key. The primary keys are underlined in the above tables.

To link a record in one table with a record(s) in another table, store the value of the primary key of a record from one table in the record(s) of the other table. For example, a Customer record is linked to one or more OrderHeader records by storing the customer's id in the CustId field of the OrderHeader record(s). Such a field that links records in one table to a record in another table is called the **foreign key**. Thus the CustId field in the OrderHeader table is a foreign key**.**

The relationship between Customer and OrderHeader (Customer -> OrderHeader) is said to be **one-to-many** because a customer may have many orders but an order belongs to only one customer. We will call the Customer table (the "one" side of the relationship) the parent table and the OrderHeader table (the "many" side) the child record. **The foreign key is added only to the child table**. Adding a foreign key to the parent key wouldn't work because it will result in multiple values for the foreign key.

In a well-designed table, each attribute should have only a single value for each value of the primary key. Thus, in the OrderHeader table, each Order# should have only one value each for OderDate, ShipMethod, and CustId. That is, **the primary key uniquely determines all other attributes**.

Another important requirement of a well-designed table is that **if an attribute uniquely determines another attribute, it must uniquely determine all other attributes.** The process of making sure that a table meets this requirement is called **normalizing** the table. A table that meets the above requirement is said to be in **Boyce-Codd normal form**. First, second, and third normal forms are additional normal forms that address certain problems that are subsets of the problems addressed in Boyce-Codd normal form. That is, a table in Boyce-Codd normal form also will be in first, second, and third normal forms.

As an example of normalizing tables, consider an improperly designed table that combines the customer information with some order data as shown here:

```
OrderHeader (Order#, OrderDate, ShipMethod, CustId, CustName, Street, Zip)
```

Order#	OrderDate	ShipMethod	CustId	CustName	Street	Zip
108213	12/10/2021	Standard	**212575**	**L.A. Rams**	**29899 Agoura**	**91301**
118312	12/16/2021	5-Day	**212575**	**L.A. Rams**	**29899 Agoura**	**91301**
124132	12/20/2021	2-Day	**212575**	**L.A. Rams**	**29899 Agoura**	**91301**
125235	12/26/2021	Standard	124542	N.E.Patriots	1 Patriot Place	02035

In this table, CustId uniquely determines CustName, Street, and Zip because for a specific value of CustId, there is only one value for CustName, Street, and Zip. But CustId doesn't uniquely determine the order data because a customer may have multiple orders. So, this is not a well-designed table. This table will duplicate the Customer data for each order placed by the customer. The name, street, and zip code for L.A. Rams, which has three orders, is repeated three times.

Splitting the table into two tables, Customer and OrderHeader, as shown here, would eliminate the duplication of data, except for the CustId values appearing in both tables:

Customer (*CustId*, CustName, Street, Zip)
OrderHeader (*Order#*, OrderDate, ShipMethod, **CustId**)

Thus normalization minimizes the duplication of data that would make maintenance of data difficult. For example, in the unnormalized OrderHeader table, a change in customer address would require changing the data in three records.

However, normalization is not without drawbacks. For example, creating a single report that shows every order with the corresponding customer information requires combining information from the two tables. This is done by matching the value of CustId in the OrderHeader table with the value of CustId in the Customer table, which is called joining the tables. Joining tables using the SQL language is described in the next section. Thus, though normalization minimizes data duplication, it increases the need to join tables, which takes additional computing power.

In addition to one-to-many relationships, there could be **many-to-many** relationships between entities. For example, an order may include many products and a product could be in many orders. Such a relationship is called a *many-to-many relationship*. When the relationship between two entities is many-to-many, in addition to the two tables representing the two entities, you need a third table that shows which records in one table are related to the records in the other table.

The OrderLine table is such a table that links the records in the Products table with the records in the OrderHeader table. It shows which products are in an order and which orders have a product. The OrderLine table, like any table that links two other tables, includes two foreign keys, Order# and Product#, corresponding to the primary keys of the two tables, OrderHeader and Product.

Next, you will learn how to use the SQL language to extract data from databases in relatively simpler cases. We will start with basic SQL statements that let you retrieve data from a single SQL Server database table and then look at retrieving related data from multiple tables.

B.2 Structured Query Language (SQL)

Though there are some variations in the SQL used in different products, the core SQL, which follows a common standard, is the same across all of them. So, once you learn one version of SQL, it is fairly easy to use it in almost any database.

Selecting Data from a Single Table

The general syntax of the SQL statement to retrieve data from a single table is:

SELECT *fieldnames* *(Field names are separated by commas.)*
FROM *table name*
WHERE *criteria* *(Specifies the criteria, if any, to select records.)*
(additional clauses, if any)

Here is an example of an SQL statement that selects the performance name and base price of all performances with a base price of less than 30 from the Performance table

```
SELECT  PerformanceName, BasePrice
FROM    Performance
WHERE   BasePrice < 30;
```

Clause	Purpose	Syntax
SELECT	Specify columns to show in the query, with or without aggregation.	SELECT column1, column2, column3
FROM	Specify the tables the data is to be retrieved from.	FROM x FROM x join y on x.pk = y.fk
WHERE	Specify rows to INCLUDE in the result set if they meet the criteria.	WHERE with IN WHERE with =,<,> WHERE with AND WHERE with OR

We will look at several examples.

Tutorial: Create and execute SQL statements

This tutorial creates and executes SQL statements.

For practice, you can type in and execute any SQL statement in Visual Studio by creating a query. Later, you will use the SqlDataSource object to create and execute SQL in web pages.

Step 1: Create a new query to execute SQL statements.

In Server Explorer, expand Data Connections, ConnectionString.

Right-click Tables. Select New Query.

You should see a new window named SQLQuery1.sql.

To run an SQL, type in the SQL statement and click the Execute button (first button on the left of the Toolbar at the top).

Step 2: In each of the following examples, type in the SQL and execute it.

Example 1: Select the name and base price of all performances:

```
1  SELECT  PerformanceName, BasePrice
2  FROM    Performance;
```

Type the above SQL into the query window without the line numbers, and click Execute. It is a standard practice to type key words like SELECT and FROM in all caps. Names of tables, fields, and so forth are not case sensitive in the SQL statement.

When you execute the statement, you should see a list of names and base prices of all performances.

Note that SQL will only extract and display the data. If the extracted data is to be used in a program, it can be stored in variables.

Example 2: Select all fields from the Performance table.

```
1  SELECT  *          -- The asterisk (*) represents all fields
2  FROM    Performance;
```

The dashes (--) show comments. You will see data from all fields listed, including the binary representation of the images. Though the asterisk (*) is easier to type, specifying the field names, separated by commas, would make the SQL more clear.

Using the WHERE Clause

When you do not want data from all records in the table, you have to specify the conditions for selecting records using the WHERE clause.

Example 3: Select the name and base price of all performances with a base price greater than or equal to 30.

```
1  SELECT  PerformanceName, BasePrice
2  FROM    Performance
3  WHERE   BasePrice >= 30;
```

The list should include only performances that meet the criterion.
Other operators include:

“>”, “<”, “<=”, “=”

Note that to check equality, SQL uses the equal (“=”) symbol, not “==”.

Example 4: Select the base price of the performance *The Hunts*.

```
1  SELECT  BasePrice
2  FROM    Performance
3  WHERE   PerformanceName = 'The Hunts';
```

Note that strings must be enclosed in single quotes, whereas numeric data doesn't have to be enclosed in quotes. The string within the quotes is case sensitive, and it must match the data in the table.
You should see the value 20 displayed.

The Wild Card Character

Example 5: Select names and base prices of all performances whose description contains the word “holiday.”

```
1  SELECT  PerformanceName, BasePrice
2  FROM    performance
3  WHERE   Description Like '%holiday%'
```

The % sign is a wild card character that represents zero, one, or more characters. The key word LIKE is required for criteria involving wild cards.

The AND Operator

You may use the logical operators “AND” and “OR” to specify multiple conditions in a WHERE clause.

Example 6: Select names, base prices, and presenters of all performances that are presented by “AKF” and have a base price that is less than 30.

```
1  SELECT PerformanceName, BasePrice
2  FROM   Performance
3  WHERE  Presenter = 'AKF'
4  AND BasePrice < 30;
```

The AND operator selects only those records that meet both conditions.

The OR Operator

The OR operator selects all records that meet one or the other condition, or both.

Example 7: Select names, base prices, and presenters of all performances by “AKF” or “OSO.”

```
1  SELECT  PerformanceName, BasePrice, Presenter
2  FROM    performance
```

```
3  WHERE   Presenter = 'AKF'
4  OR    Presenter = 'OSO';
```

Note that the following SQL is invalid:

```
1  SELECT  PerformanceName, BasePrice, Presenter
2  FROM    performance
3  WHERE   Presenter = 'AKF' OR 'OSO';   -- invalid
```

The IN Operator

The IN operator is a simpler alternative to using one or more OR operators to select records that may contain one of several string values.

Example 8: The following statement is equivalent to the statement in Example 7:

```
1  SELECT  PerformanceName, BasePrice, Presenter
2  FROM    performance
3  WHERE   Presenter IN ('AKF', 'OSO');
```

The IN operator is particularly easier to use when the criteria involve multiple values, as in

```
1  SELECT  PerformanceName, BasePrice, Presenter
2  FROM    performance
3  WHERE   Presenter IN ('AKF', 'OSO', 'OAC Band');
```

Criteria Involving AND and OR Operators

When a criterion involves both AND and OR, the AND operator is considered before OR. It is recommended that you use parentheses to make the precedence clear.

Suppose you want all performances that have a base price less than or equal to 25, and the presenter is either "OSHO" or "ACG". You may run the first SQL shown below, which gives an incorrect result, and compare it with the correct result given by the second SQL that uses parentheses.

```
1  SELECT  PerformanceName, BasePrice, Presenter
2  FROM    performance
3  WHERE   BasePrice <= 25 AND Presenter ='OSO' OR Presenter ='ACG' ;
```

Working with Dates

Example 9: Select the id and date of all performances on or after March 1, 2021, at 7:30 p.m.

```
1  SELECT  PerformanceId, PerformDate
2  FROM    PerformDate
3  WHERE   PerformDate >= '01-Mar-2021 7:30 pm';
```

Line 3 shows the default format for date/time. If time is omitted, the default value is 12:00 a.m .

Example 10: Select the id and date of all performances in the month of March.

```
1  SELECT  PerformanceId, PerformDate
2  FROM    PerformDate
3  WHERE   Month(PerformDate) = 03;
```

The function Month gives the month number of the date (3).
Similar functions are available to get the day number, weekday, year, and so on.

Sorting Records

Example 11: Select the Performance name and base price of performances with a base price greater than 30, sorted by performance date.

```
1  SELECT      PerformanceName, BasePrice
2  FROM        Performance
3  WHERE       BasePrice > 30
4  ORDER BY    BasePrice;
```

The ORDER BY clause must follow the WHERE clause.

Grouping Records and Computing Group Summaries

Example 12: Compute the average base price of performances for each presenter.

```
4  SELECT      Presenter, AVG(BasePrice) AS AveragePrice
1  FROM        Performance
2  GROUP BY    Presenter;
```

The key word AS is used to specify the alias, AveragePrice, for the average price. The column that displays the average price uses the alias AveragePrice as the caption.

Selecting Related Data from Two Tables Using JOIN

The JOIN operation lets you combine related data from two tables. The general pattern of the statement to join two tables is:

```
SELECT    fields
FROM      table1 JOIN table2      -- or, table1 INNER JOIN table2
ON        foreign key in one table = primary key in another table
```

Example 13: Join the **Performance** and **PerformDate** tables, shown below, to display the name and base price, along with the date of each performance.

Performance (*PerformanceId*, PerformanceName, BasePrice, Description, Presenter, Image)

PerformanceId	PerformanceName	BasePrice	Description	Presenter	image
11	Young Irelanders	35	Each and eve...	AKF	*NULL*
12	Justin Hayward	30	Justin David ...	*NULL*	*NULL*
13	TedxOshkosh	59	We believe in...	*NULL*	*NULL*
14	Orchestral Presents	20	Join the Osh...	OSO	0xFFD8F...
15	Ladies of Laughter	30	Since 2012 t...	*NULL*	*NULL*

PerformDate (*PerformDateId*, **PerformanceId**, PerformDate)

PerformDateId	PerformanceId	PerformDate
101	11	1/3/2021 7:30:00 PM
102	11	1/4/2021 7:30:00 PM
103	11	1/5/2021 7:30:00 PM
104	11	1/10/2021 7:30:00 PM
105	12	1/11/2021 7:30:00 PM
106	12	1/6/2021 7:30:00 PM
107	13	1/7/2021 8:00:00 PM
108	13	1/7/2021 7:30:00 AM
109	14	1/8/2021 8:00:00 PM
110	15	1/9/2021 2:00:00 PM
111	15	1/12/2021 7:30:00 PM

Note that each record in the PerformDate table is linked (related) to a record in the Performance table by the foreign key PerformanceId. For example, the first four records that have a value of 11 for PerformanceId are linked to the first record in the Performance table.

Here is the SQL statement that joins the two tables and retrieves the data.

```
1  SELECT      Performance.PerformanceId, PerformanceName, BasePrice, PerformDate
2  FROM        Performance JOIN PerformDate
3  WHERE       Performance.PerformanceId = PerformDate.PerformanceId
```

When you select fields from multiple tables, if the same field name appears in two tables, the field name must be preceded by the table name followed by a period, as shown in lines 1 and 3.

Line 3 specifies the primary key in the parent table (Performance) and the matching foreign key in the child table (PerformDate).

Note that the PerformDate table has a field named PerformDate, which could be confusing.

The following is the subset of the data set retrieved by the SQL statement, corresponding to the first five Performance records:

	PerformanceId	PerformanceName	BasePrice	PerformDate
1	11	Young Irelanders	35	2021-01-03 19:30:00.0000000
2	11	Young Irelanders	35	2021-01-04 19:30:00.0000000
3	11	Young Irelanders	35	2021-01-05 19:30:00.0000000
4	11	Young Irelanders	35	2021-01-10 19:30:00.0000000
5	12	Justin Hayward	30	2021-01-11 19:30:00.0000000
6	12	Justin Hayward	30	2021-01-06 19:30:00.0000000
7	13	TedxOshkosh	59	2021-01-07 20:00:00.0000000
8	13	TedxOshkosh	59	2021-01-07 07:30:00.0000000
9	14	Orchestral Presents	20	2021-01-08 20:00:00.0000000
10	15	Ladies of Laughter	30	2021-01-09 14:00:00.0000000
11	15	Ladies of Laughter	30	2021-01-12 19:30:00.0000000

The JOIN operation compares each record in the parent table with each record in the child table. If the foreign key matches the primary key, then the two records are joined and displayed.

This type of join operation, called INNER JOIN, displays data from only those records from the parent table that have a match in the child table. For example, the Performance record for *A Grand Night* (PerformanceId: 23) doesn't appear in the result because there are no corresponding records in the PerformDate table.

You may use another type of join called OUTER JOIN that lets you display records that don't have matching records in the child table.

Note that the join operation does not create a new table; it simply collects data from the two tables and displays it.

You may extend the join operation to select data from more than two tables.

Nested Queries

In previous examples involving the WHERE clause, the values used to select records were hardcoded into the statement. However, some queries may involve selecting records from a table using a criterion that involves values that are selected from another table or the same table. In such cases, it is convenient to use one SQL statement within another one.

Example 14: Select names and base prices of all performances scheduled for the month of February.

The performances that are scheduled for a specific month (February) can be found in two steps:

1. Select from the PerformDate table all PerformanceId's that have a corresponding date in February. For example, if February is the month, then the set of PerformanceId's (18, 19, 21) corresponds to dates in February, in the subset of the data shown here.

PerformDateId	PerformanceId	PerformDate
121	19	2/1/2021 8:00:00 PM
122	18	2/2/2021 8:00:00 PM
123	19	2/3/2021 2:00:00 PM
124	20	3/1/2021 7:30:00 PM
125	20	3/2/2021 7:30:00 PM
126	20	3/3/2021 7:30:00 PM
127	20	3/4/2021 7:30:00 PM
128	21	2/5/2021 7:30:00 PM

The following SQL would select the PerformanceId's for the specific month:

```
Select PerformanceId
From PerformDate
WHERE Month(PerformDate) = 2
```

2. Select the records from the Performance table whose PerformanceId matches those selected in step 1.

   To use the set of PerformanceId's from step 1 in an SQL, you would nest the SQL from step 1 in an outer SQL as shown in Figure B-1.

```
SELECT PerformanceName, BasePrice
FROM [Performance]
Where PerformanceId IN
         (Select PerformanceId
         From PerformDate
         WHERE Month(PerformDate) = 2);
```

Figure B-1: Nested SQL

Line 2 is the outer SQL that uses the inner SQL from step 1 (line 3) to get the PerformanceId's. The IN operator compares each PerformanceId from the Performance table with each PerformanceId selected by the inner SQL, and if there is a match, the corresponding performance name and base price are selected.

Appendix C

Creating a Database

SQL Server databases can be created in Visual Studio because the Express edition of SQL Server is automatically installed on your computer when you install Visual Studio.

Tutorial: Creating the HR Database

This tutorial creates a database called HR and a table named Employee:
Employee (EmpId, LastName, FirstName, DateOfBirth, Phone, DeptNo, CountryId, Salary, FullTime).

The HR database will be created on the MSSQLLocalDB server, which is a version of the SQL Server Express database server. Note that the HR database is not used in the tutorials in the chapters. It is created for practice.

Step 1: Create a new SQL Server database named HR.

Open Visual Studio and select ***Create a new project*** from the start window.

You will see the Create a New Project window. Type in "database" in the search box and select ***SQL Server Database Project*** template, as shown in Figure C-1.

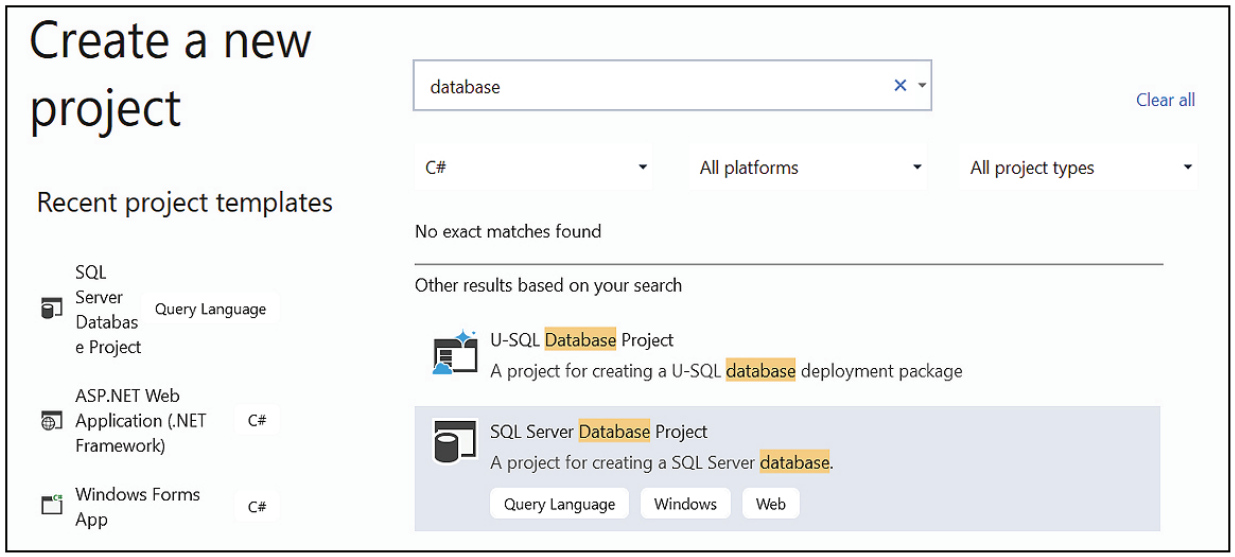


Figure C-1: *Create a new project* window

Click ***Next***. You will see the ***Configure your new project*** window,

Enter **HR** for project name and WebProjects for location, as shown in Figure C-2.

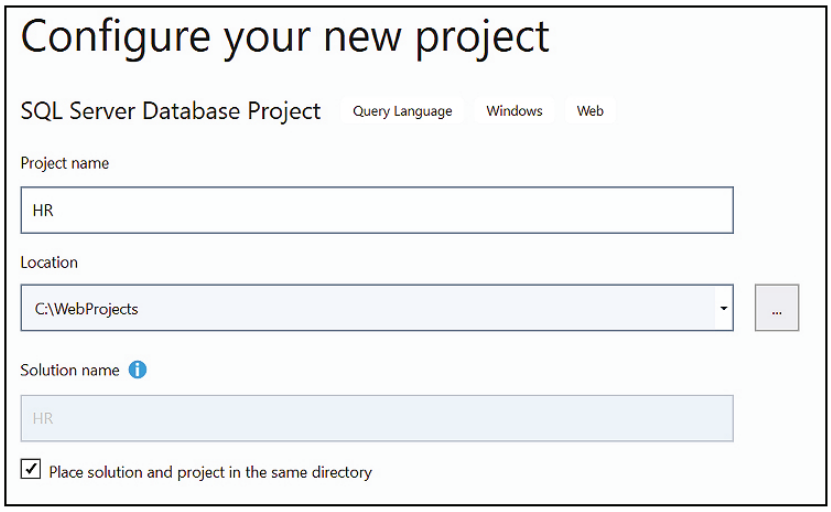


Figure C-2: *Configure your new project* window

Click ***Create***.. Visual Studio will open and display the Solution Explorer window that looks as shown in Figure C-3.

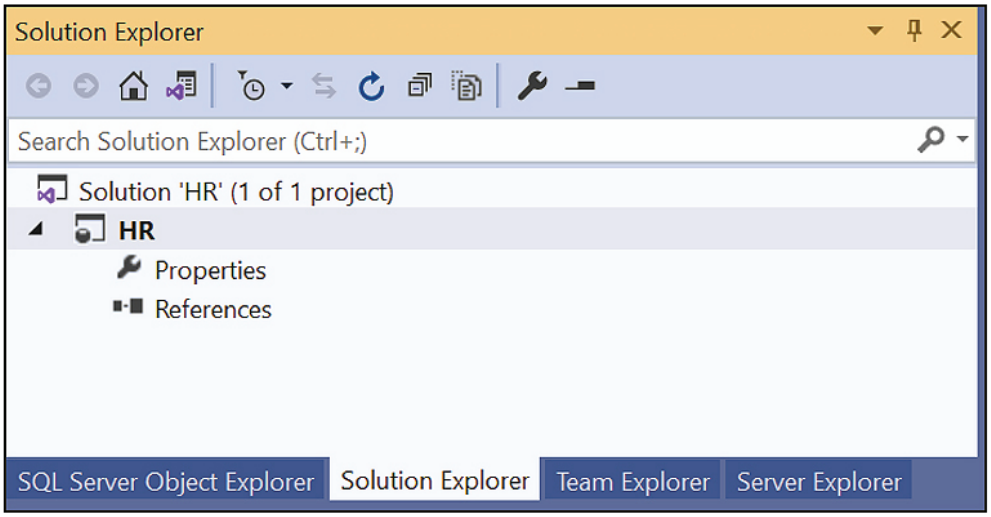


Figure C-3: Solution Explorer window

Step 2: Create a new table named Employee

From the Visual Studio menu, select ***View*** > ***SQL Server Object Explorer***.

The ***Object Explorer*** window opens as shown in Figure C-4.

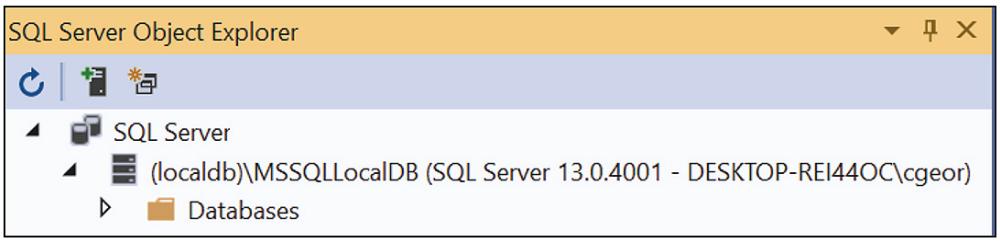


Figure C-4: SQL Server Object Explorer window

Expand the Databases node to display the HR database as shown in Figure C-5.

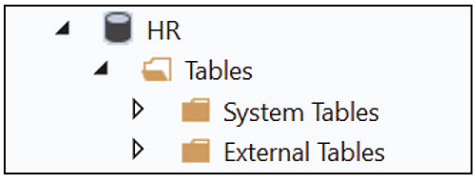


Figure C-5: HR database

Right click the *Tables* node and select *Create New Table*

The Table Designer will open displaying a grid to specify the names, data types and other attributes of the columns of the table. The bottom half of the window will display the T-SQL script pane.

Change the table name in the T-SQL statement (CREATE TABLE [dbo].[Table]) to Employee, as shown in Figure C-6.

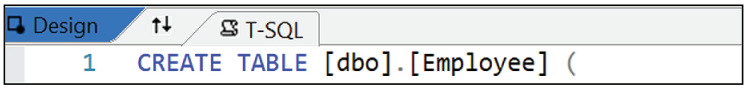


Figure C-6: The script pane.

Step 3: Specify the columns of the table.

Specify the name, data type and other attributes for each column as shown in Figure C-7.

Notice that the T-SQL script is automatically generated as you add the columns.

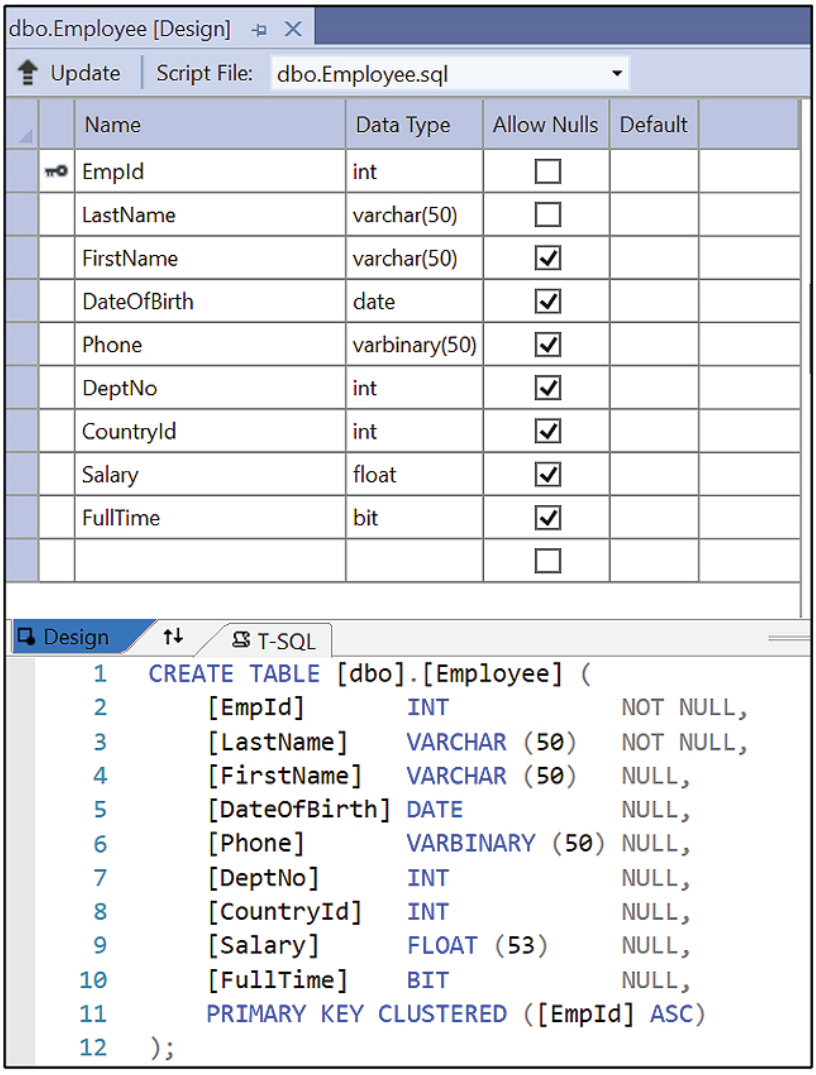


Figure C-7: The Table Designer

Click the *Update* button. You will see the *Preview Database Updates* window.

Click the *Update Database* button.

To view the newly created table, right-click the Tables node in the Object Explorer and select Refresh. You will see the dbo.Employee node, as shown in Figure C-8.

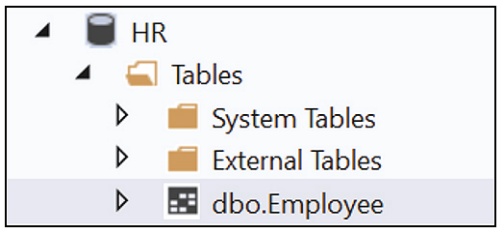


Figure C-8: The Employee table.

Step 4: Add data to the table

Right-click dbo.Employee and select ***View Data***.

Enter data for two employees as shown in Figure C-9.

dbo.Employee [Data] | dbo.Employee [Design]

Max Rows: 1000

EmpId	LastName	FirstName	DateOfB...	Phone	DeptNo	CountryId	Salary	FullTime
11002	Iwuji	Chris	11/1/1992	920-426-...	1	62	46200	True
11005	Mathews	Ruth	3/24/1985	646-233-...	4	22	44600	True

Figure C-9: Employee records

You may create additional tables within the HR database using the same process.

The Data File property of the HR database shows that the database file is stored in the following folder:

C:\Users***username***\AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances\MSSQLLocalDB\HR_Primary.mdf

Note that AppData is typically a hidden folder

You also may create a database that is stored in a specific folder like WebProjects.

Step 5: Create a database that is stored in the WebProjects folder.

Right-click Databases in the Object Explorer and select Create New Database.

Enter the database name and location as shown in Figure C-10.

Create Database

Database Name: HR2

Database Location: C:\WebProjects ...

OK Cancel

Figure C-10: Create Database window

Click Ok. You will see the database name HR2 under the Databases node, and the database file HR2.mdf in the WebProjects folder.

Appendix D

Object-Oriented Programming

Most professional programming is done in languages based on the object-oriented paradigm. C# is no exception and is in fact one of the languages with the strongest support for proper object-oriented programming. Throughout the previous chapters of this book, you have actually made frequent use of many object-oriented principles and techniques. For instance, every user interface element, such as Button, TextBox, and Label, are all objects. In this appendix, you will learn some basics of object-oriented programming (OOP) and how you can create and work with your own objects.

D.1 Introduction to Objects and Classes

Object-oriented programmers generally distinguish between the problem domain and the application domain. The problem domain involves the parts of the real world that the computer system is working with and solving problems for. For instance, the problem domain for a payroll system would contain the employees, the actual hours worked by the employees, and all the rules governing how salaries, taxes, and other deductions are calculated and paid out. The application domain, on the other hand, is the actual payroll computer system and its users. The users will work with representations of the problem domain (real world) in order to solve the problems the system is intended to solve.

With object-oriented programming, we start by creating representations of the problem domain inside the application. The problem domain typically contains multiple entities like employee and payroll. The instances of entities we want to keep track of in the problem domain are represented in the application domain by *objects*. So, in the payroll system, each employee becomes an object, every pay period becomes an object, every paycheck becomes an object, and so on. The set of objects that represents the instances of an entity is described in computer code with a construct called a *class*. A class is like a blueprint that can be used as a template to create many instances (objects) that have the same properties. In this way, a class represents an entity of the real-world problem addressed by the application. For example, a payroll application for an organization may use an Employee class to represent employees of the organization.

D.2 Classes versus Objects

With object-oriented programming, you start by creating classes that represent real-world entities. A class consists of code defining (1) data items that represent an entity's attributes or fields, and (2) the behaviors or methods that entities of that class understand. The name, birth date, and address of the employee are examples of the data items of the Employee class. The data items in a class are accessed using an interface of publicly available methods and properties of the class. Behaviors are the activities or functions of an entity. Updating the hourly rate for an employee would be a behavior of the employee entity, which may be implemented by a method called PayRaise within the Employee class.

Figure D-1 shows the Employee class in Unified Modeling Language (UML) notation. In UML, a class is represented by a box divided into three sections. The top portion contains the class name, the middle portion contains the attributes or properties, and the lower portion contains the methods. The plusses and minuses signify whether the element is public or private, respectively. Public elements make up the public interface that other classes in the system can access, whereas private elements can only be accessed from within the class itself. The notation for the methods is similar to C# but also slightly different. The name of the method is given first, followed by parentheses that list the parameters that the method accepts, as well as the data type for each parameter. If the method returns a value, the data type for the return value is given after the parameters. In this example, the PayRaiseAmount method is public and takes a single parameter of the type decimal. It doesn't return anything.

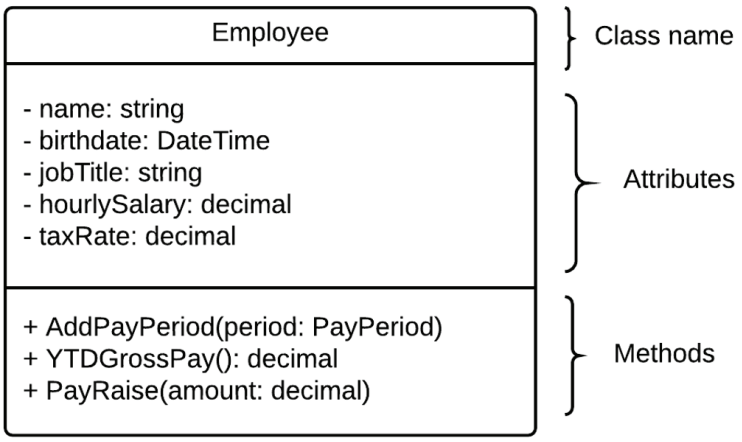


Figure D-1: Employee class in UML notation

The Employee class describes the attributes and behavior of the employees of an organization. To represent an individual employee like John Smith, an application creates an instance of the Employee class in memory, called an object. So, the object is an *instantiation* of the class.

The class is an abstraction of the real-world entity written in computer code; thus, the same Employee class can be used to create multiple objects, each object representing a different employee. Objects in C# are characterized by three general concepts:

- **Identity**: Just like every employee is distinct from every other employee, so too is every object in the computer system distinct from every other object. Object-oriented systems automatically implement an identity mechanism.
- **State**: The state of an object is the set of values of the attributes that we care about regarding that object. Each object has specific values for the things we care about. So, we might have two employees with these two states. Table D-1 provides some examples of objects.

Table D-1: Examples of objects

Attributes	Employee 1	Employee 2
Name	John Smith	Rebecca Jones
Birthdate	12/10/1993	10/5/1994
Address	200 Main St.	100 Elm St.
Job Title	Network Engineer	Software Developer
Hourly Pay	$35	$45

It's important to realize that the state of an object can change over time.

- **Behavior:** Each object has a specific behavior that is also modeled in the system. In the problem domain, we might have employees punch in for work, punch out, get their salary paid out, get a pay raise, and so on. In the application domain, behavior is implemented as methods that can be called on an object. The method code specifies what action happens when the method is called on a particular object.

The concepts of class and object are often confused and described in overlapping terms, but they are two distinct concepts that are important to keep separate. Classes are described in code and are used as the blueprints to instantiate (create) the objects. Each object in an application represents an instance of a real-world entity. There would only be one Employee class, but many Employee objects (one for each actual employee in the organization).

D.3 Information Hiding (Encapsulation)

One of the defining principles of object-oriented programming is that of Information Hiding (sometimes also referred to as *encapsulation*). The idea is that the way the data is presented by an object to other parts of the system is independent of how it is actually stored in the object.

This distinction provides several advantages. First, it allows for a simple and consistent internal representation of data in an object. For example, the total time worked by an employee could be stored in minutes, but it could be presented in the public interface by a method that returns fractional hours. Second, it protects the state of the object from being changed in inappropriate ways. For example, if the time worked by an employee is really represented by

successively punching in and out, it would be inappropriate to be able to change the total time worked directly; it should only be changed through the transactions. Lastly, it also allows for restricting how the data inside an object can be accessed. Some attributes of an object should not be changed from outside of that object.

To achieve this, each object provides a private implementation of data and a public interface, and only the public interface is available to other parts of the system. The implementation is thus "hidden" or "encapsulated" inside the object. In Figure D-2, the attributes and methods marked with a plus represent the public interface, whereas the ones with a minus are private. Attributes are implemented in C# using private fields that are exposed through public **properties** and methods.

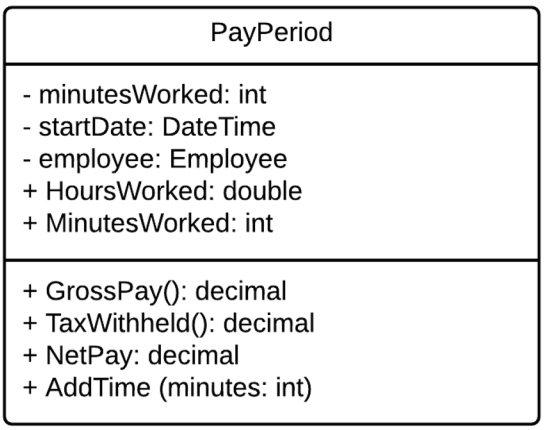


Figure D-2: PayPeriod class

As an example, consider the Employee object described above. We have specified that it has a Name attribute. How should that actually be stored in the object? A simple approach could be to store it in a single string. However, you could also split it in two and store the first and last name separately. In that case, the attribute that provides the full name would be responsible for combining the first and last name and presenting it to its clients in that form.

Tutorial: Creating an Employee Class

In this tutorial, you will create your first class and several objects from this class to be used in a payroll system. The class is the Employee class that was mentioned above. In the context of this simplified system, an employee will have the following attributes:

- Name—of type string
- Birthdate—of type DateTime
- Hourly salary—of type decimal
- Job title—of type string
- Tax rate—of type double (must be between 0 and 1)

Employee objects will have the following behavior:

- Given a number of hours worked during a pay period, calculate the gross salary, the tax amount, and the net amount of money to be paid out.

Step 1: Start Visual Studio and create a new Windows Forms Application project called Payroll.

Step 2: Add the Employee class.

Right-click the project in the Solution Explorer and select Add > Class...

Name the class Employee.cs and click Add.

Fields

When implementing classes, fields are variables at the class level that represent the attributes for the class and allow for storing the values of the attributes (the state) of an object. Fields should always be marked private. If you do not add an access modifier, C# will default to the most restricted access you could declare for that member. This would be private for fields, but it is recommended to still add the private key word to signal your intent clearly.

Step 3: Add fields to the Employee class.

Add the code shown in Figure D-3 to the Employee class.

```
7   namespace Payroll
8   {
9       class Employee
10      {
11          private string name;
12          private DateTime birthday;
13          private decimal hourlySalary;
14          private string jobTitle;
15          private decimal taxRate;
16      }
17  }
```

Figure D-3: Fields of the Employee class

Each of the fields declared here helps describe an Employee. Notice how they correspond to the attributes shown in Table D-1. When you create different objects from the Employee class, each object will have a different set of values for the fields because each object represents a different employee.

Each field is declared using the private key word, which is a way to express that the variable cannot be directly accessed from outside the class and that details of how the data is stored are to be kept internal to the object. This will allow you to change the implementation without affecting the external usage of the class.

The Constructor

The constructor is a special method that is called when creating an object from a class. The constructor has two purposes: create the object and initialize the fields. Creating the object happens behind the scenes via the runtime environment when your program is running; you don't have to do anything special in the constructor code. However, you will have to write the code to initialize the fields, which you will see in the next step.

The constructor takes parameters that can be used to initialize the fields. In this case, there are parameters for each of the fields except the tax rate, which is set to a default value of 0.25 (all employees pay 25% tax by default).

In C#, the constructor follows a specific pattern:

- You don't get to choose the name of the constructor; it must be the same as the name of the class.
- You don't get to specify the return-type of the constructor; it must return an object of the class. Since mentioning the return-type would be redundant, the return-type is entirely omitted—so you would just use "Employee(...) {...}" rather than "Employee Employee(...) {...}".
- Each field is initialized in the constructor. If you do not explicitly initialize a field, it will get a default value, depending on its data type (numbers become zero, strings become null, Booleans become false, etc.). However, it is good practice to explicitly initialize your fields, even if it's what the default would be, to communicate to others that you weren't accidentally overlooking a field.
- As we'll discuss in a moment, calling a constructor is conceptually the same as calling any other function, but it gets its own special syntax: you must use the additional keyword "new" in front of the constructor-name/class-name.

Step 4: Add the constructor to the Employee class.

Add the code shown in Figure D-4 after the declaration of the fields, before the closing brace of the Employee class.

```
16
17      public Employee(string name, DateTime birthday, decimal hourlySalary, string jobTitle)
18      {
19          this.name = name;
20          this.birthday = birthday;
21          this.hourlySalary = hourlySalary;
22          this.jobTitle = jobTitle;
23          taxRate = 0.25m;
24      }
25
```

Figure D-4: Employee constructor

The key word "this" is used to refer to the current object instance and allows you to distinguish between fields and parameter values, as in line 19:

```
this.name = name;
```

At first glance, this line looks a little strange, but it is actually quite simple: this.name refers to the field name (line 11 in Figure D-3), and the name on the right side of the equal sign refers to the parameter name. You always access fields/methods via the syntax *object.field*.

You can see this in Visual Studio by placing the cursor inside each of the two occurrences of *name*. When you click on the one on the right, the parameter is highlighted, and when you click on *this.name*, the field is highlighted. It isn't a requirement that the constructor parameters match the name of the fields, but it is fairly common that they do.

The ToString Method

It is often useful to have an object be able to provide a brief one-line description of itself. This is usually done by implementing a method called ToString. There are technical reasons for the method to be called ToString that will be covered later, when we discuss the concept of *inheritance*. For now, just add this method to the Employee class.

Step 5: Add the ToString method to the Employee class.

```
26      public override string ToString()
27      {
28          return string.Format("Name: {0}, Birthday: {1:d}, Hourly Salary: {2:c}, Job Title: {3}",
29              name, birthday, hourlySalary, jobTitle);
30      }
```

Figure D-5: ToString method

As you can see, the method just returns a string that includes most of the fields. The meaning of the override key word in line 26 will also be discussed later during the coverage of inheritance.

Step 6: Create the user interface.

Rename Form1.cs in Solution Explorer to EmployeeForm.cs (and rename in the code as well when you're asked). Change the Text property of the form to Employees.

In order to demonstrate how objects are created, add a label named lblEmployee to the Form in the project.

Double-click the background of the form (not the label) to switch to the code view of the form. It should look like Figure D-6.

```
11  namespace Payroll
12  {
13      public partial class EmployeeForm : Form
14      {
15          public EmployeeForm()
16          {
17              InitializeComponent();
18          }
19
20          private void EmployeeForm_Load(object sender, EventArgs e)
21          {
22
23          }
24      }
25  }
```

Figure D-6: Initial code in the EmployeeForm class

As you can see from line 13, the Form is in fact a class, called EmployeeForm. This means that when the program is running and the Form is being displayed, an EmployeeForm object is created. This class currently has two methods:

1. A constructor, specified in lines 15–18, that takes no parameters and calls the method InitializeComponent, which is created by Visual Studio to create all the user interface components in the form. If you want to see this auto-generated code, you can right-click on the method name and select Go To Definition.
2. A method called EmployeeForm_Load, specified in lines 20–23, that takes two arguments. The EmployeeForm_Load method is called automatically by the system when the form is loaded. There are a number of other event handler methods that you can add to your form and insert code into to execute code at certain times in the life cycle of a form.

Creating an Object

When creating an object, you follow a specific pattern:

```
ClassName variableName = new ClassName();
```

This line declares a variable on the left side of the equal sign and on the right side calls the constructor for the class. The right side generates an object that is then assigned to the variable on the left side.

The key here is the use of the *new* key word in front of the call to the constructor. The *new* key word is used to indicate that an object is created from the specified class. Let's create our first Employee object.

Step 7: Create objects and display ToString.

Add the following code to the EmployeeForm_Load event handler:

```
DateTime birthday = new DateTime(1999, 5, 1);
Employee spongebob = new Employee("Spongebob", birthday, 10.5m, "Burger flipper");
lblEmployee.Text = spongebob.ToString();
```

This code actually creates two objects. First, a DateTime object is created for May 1, 1999. This is assigned to the variable birthday. Then, an Employee object is created using the birthday object and several other pieces of data. This is assigned to the spongebob variable. Finally, the ToString method is called on the spongebob object, and the result is assigned to the Text property of the label you added to the user interface above.

Run the program and you should see the description of the Employee object displayed on the form, as shown in Figure D-7.

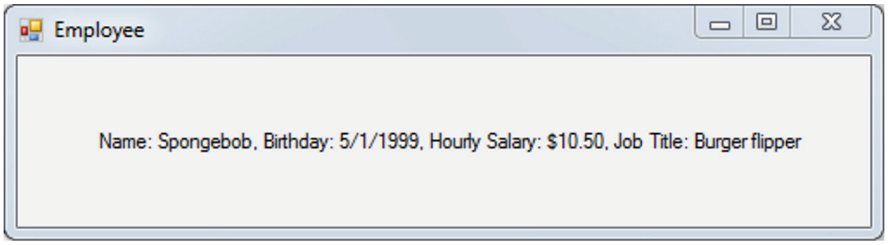


Figure D-7: The first object created

To make sure you understand how the object is created and displayed, add a Breakpoint on the second line of code in the EmployeeForm_Load event handler. Once the debugger stops on that line, examine the variables available at that point.

Next, step into the Employee constructor, and see how the values are initialized in the object. Once you return to the Form object, reexamine the variables. You should now see that the spongebob variable has values associated with each of the fields.

Continue stepping into the ToString method, and notice that you are back in the Employee class looking at the spongebob object.

It's time to practice! Do Step 8.

Step 8: Add a second label and create a new Employee object with different values, and show the new object on the new label.

D.4 Properties

So far, you have seen fields and methods in a class. The fields contain the state of the object, and the methods implement the object's behavior. Because of encapsulation, the fields are always kept private so as not to allow other parts of the system to access the fields directly. However, in many cases, we want to be able to expose the fields and be able to read and change them in a controlled manner. For instance, it is easy to imagine that you could create a form with controls for all the fields for the Employee class and allow a user to enter values into the controls and assign those values to the fields. But because the fields are private, this would not be possible. You can try right now in the program you created previously to see if you can assign a different value to one of the variables in the spongebob object after it has been created by adding this line after creating the object:

```
spongebob.name = "Spongebob Squarepants";
```

You will get a syntax error from trying this, because name is declared as private in the Employee class, so go ahead and comment out that line of code.

So, how can we allow the fields to remain private but still change their values? The C# language to a large extent was created by Microsoft as an improved version of Java. The best way to understand why C# has a concept of properties is to go back to the Java language.

In Java, we would create public "getters" and "setters" to retrieve and change the field values. For the field, name, these would look like this:

```
public string getName()
{
    return this.name;  //return the value of the name field
}
public void setName(string value)
{
    this.name = value; //assign a value to the name field
}
```

If a programmer wanted to not allow a client to change a field, he/she would simply not write the set method. By keeping the fields private, access to them could be controlled through the getters and setters. For instance, if it

was necessary to do any kind of conversion, like splitting the name into last and first names, this could be done in the getters and setters.

If you implemented these methods, you could change the name with this line of code:

```
Spongebob.setName("Spongebob Squarepants");
```

However, one problem with this approach is that getters and setters are not a part of the Java language; they are merely a convention that most Java programmers follow. So, the C# language designers decided to introduce the concept of a **Property** that takes the place of the getters and setters in Java. Here is a property that provides the same functionality as the getters and setters above:

```
public string Name
{
    get
    {
        return this.name; //return the value of the name field
    }
    set
    {
        this.name = value; //assign a value to the name field
   }
}
```

This is a single construct called Name that includes a get section and a set section. The get simply returns the value in the named field. The *set* property uses the reserved keyword *value* to provide the value to be stored in the field. Just like with getters and setters in Java, you can add any valid C# code to the properties. You're not restricted to just assigning and returning values from fields. You could do conversions or have properties whose values are calculated and do not have an associated field.

Once the property has been added, you can change the name in this way:

```
spongebob.Name = "Spongebob Squarepants";
```

This is very similar to your first attempt at changing the name, but because the property is declared public, it will actually work.

How do you retrieve the name? You have seen many examples throughout this book of using properties, as they are very common with user interface controls and other parts of the .Net framework, so a single additional example here should suffice. Imagine a form that has a label called lblName and a TextBox named txtName:

```
lblName.Text = spongebob.Name;
spongebob.Name = txtName.Text;
```

In the first statement, the Name property of the spongebob object retrieves the value of the name field using the get section of the property. The value is assigned to the Text property of the Label.

In the second statement, the content of the textbox is retrieved using the Text property of the txtName object and assigned to the Name property of the spongebob object. When a value is assigned to a property, the set section of the property is used to assign the value to the field through the value key word in the set section of the property. So, the content of txtName is assigned to the name field of the spongebob object.

The naming convention for properties is to change the first letter of the corresponding field to uppercase. If you wanted to have a read-only property, you would simply leave out the set section.

D.5 Automatic Properties

One final note about properties is that very often there is a lot of boilerplate code when creating a class:

- A private field
- A public property that simply gets and sets the values for the fields

When this is the case, the field and property code can be written using a simplified version, called an *Automatic Property*. Here's the Name property from earlier using the automatic property:

```
public string Name { get; set; }
```

This one line of code replaces the following private field and the property code:

```
private string name;
public string Name
{
    get
    {
        return this.name;
    }
    set
    {
        this.name = value;
   }
}
```

As you can see, Automatic properties are much shorter and easier to read. The usage remains the same as you saw above, except the field can no longer be accessed directly inside the property or in the rest of the class.

With an automatic property, the compiler will automatically create a backing field of the correct type and implement the standard get and set code you saw for the fully implemented property. So, the only difference is that it is much shorter to write and read. When you use automatic properties, you have to remember that you cannot implement any conversion or other code in the get and set sections. You must also supply both get and set, so you cannot create automatic read-only or write-only properties.

Appendix E

C# Programming Using Visual Studio Code and Visual Studio Community

This appendix shows you how to create and run C# code in VS Code and Visual Studio Community. You will learn how to create console applications (console-type projects) that allow users to interact with the application in a console window. Users can run commands at a *command line*, enter input data, and display output in the console window. This approach, which doesn't require you to create a user interface to run C# code, will help you focus on programming logic. Both VS Code and VS Community have a built-in *terminal* that allows you to run applications in a console window.

Running C# in VS Code

As you learned in Chapter 1, Visual Studio Code (VS Code) is a lightweight yet powerful source code editor, available for Windows, macOS, and Linux. You can configure VS Code to develop and run programs in a variety of languages, including C#. The instructions in this section are based on VS Code, version 1.54.

Creating and running a C# project in VS Code involves the following simple steps:

1. Create a project folder. You may do this in Windows Explorer or Finder on a Mac.
2. Open VS Code and Install a **C# extension**.
3. Open the project folder in VS Code
4. Open VS Code's built-in **terminal**. You can run *dotnet* commands in the terminal to do a wide variety of tasks, including initializing a project, running C# code, and changing the directory.
5. Initialize the project. This is done by typing in the dotnet command **dotnet new Console**. This command creates the initial files and folder for the C# project.
6. Run the C# project using the command, **dotnet run.**

Tutorial 1: Create a C# Console App and Run It in VS Code

Step 1: Create the project folder.

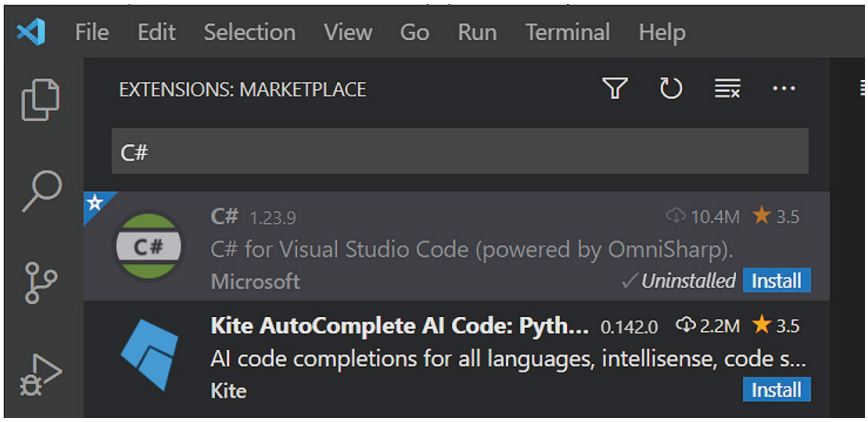
Create a new folder, within the WebProjects folder, where you want to create the C# project. Name it *Csharp-intro-VSCode*.

Step 2: Install a C# extension in VC Code.

Open Visual Studio Code.

Click the ***Extensions*** button on the Toolbar on the left side or select ***View>Extensions*** from the menu to see a list of available extensions.

To narrow down the list, type in ***C#*** in the search box. Select the first extension, which is Microsoft's C# extension powered by OmniSharp.



Close the Extensions tab, and click the ***Extensions*** button to close the list.

After you install an extension, if you see the message "Required assets to build and debug are missing ..." at the bottom right of the window, click ***Yes*** to install them.

Step 3: Open the project folder, Csharp-intro-VSCode, in VS Code.

Open VS Code's Explorer window and click ***Open Folder*** (or select ***File>Open Folder***).

Select the ***Csharp-intro-VSCode*** folder, and click ***Select Folder***.

Step 4: Open VS Code's terminal.

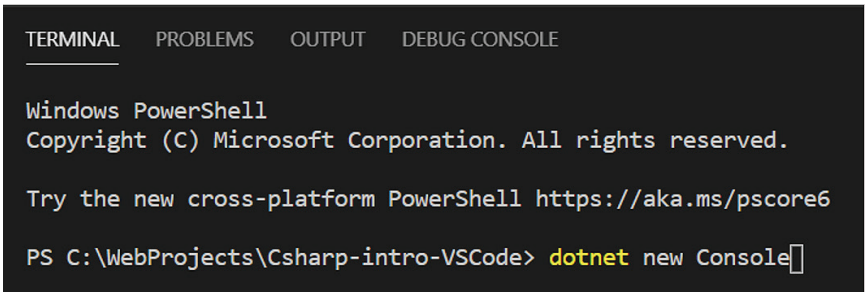
Select ***View>Terminal***.

Visual Studio opens the terminal, which in turn opens an instance of Microsoft's ***PowerShell*** software, a command-line shell that provides a toolset for task automation and configuration management. The figure in Step 5 shows the terminal with the command line that displays the current directory.

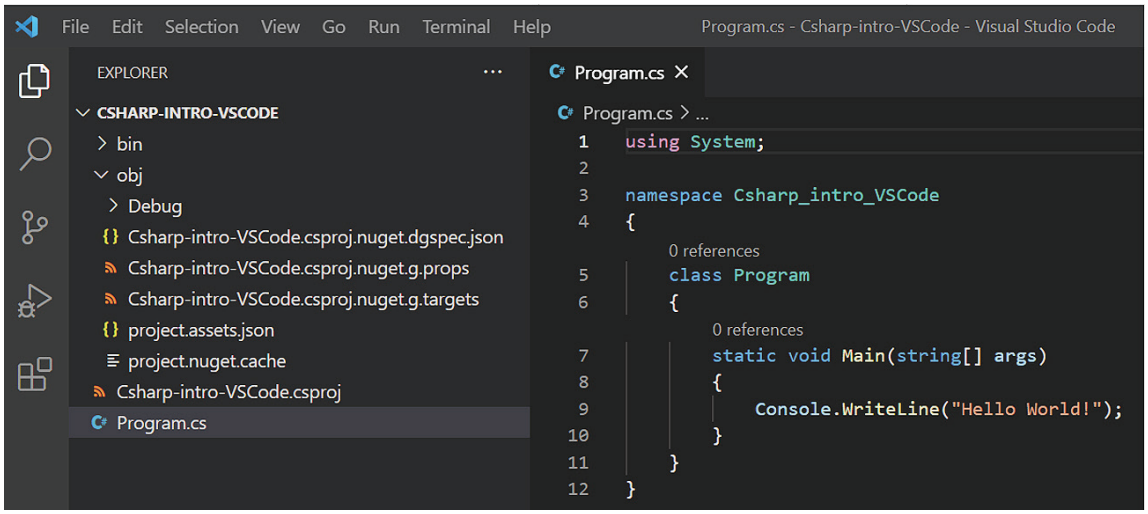
To close an existing terminal and open a new terminal, select ***Terminal>New Terminal***.

Step 5: Initialize the project.

To initialize the project and create the initial files for the project, type in the dotnet command, ***dotnet new Console,*** at the prompt, as shown. Click ***Enter***.



VS Code adds the initial C# project files (.csproj and .cs files) and the obj folder, within the Csharp-intro-VSCode folder, as shown in the following figure.



Click the program.cs file to open it. You will see the Main() C# method that contains the single statement,

```
Console.WriteLine("Hello World!");
```

The Console.WriteLine() method displays the specified string ("Hello World").

To display a list of folders and files in the current folder, type in **dir** in the terminal and click Enter.

Step 6: Run the C# method, Main().

To run the Main() method, type in **dotnet run** at the terminal prompt. You will see the output of the program "Hello World" is displayed in the terminal, as follows:

```
PS C:\WebProjects\Csharp-intro> dotnet run
Hello World!
PS C:\WebProjects\Csharp-intro>
```

Step 7: Edit the C# code.

To gain insight into writing and editing C# code, replace the existing Console.Write() statement with the three statements shown in lines 3–5 in the following code.

You may use shortcuts like ***CW*** followed by a tab to write the Console.Write() method, Notice how the IntelliSense feature helps you to complete the statements.

```
1  static void Main(string[] args)
2  {
3     Console.WriteLine("Please enter the first name:");
4     string firstName = Console.ReadLine();
5     Console.WriteLine("Hello " + firstName);
6  }
```

Here is a brief description of the statements:

Line 3 writes the prompt to enter the first name.

In line 4, the Console.ReadLine() method causes the application to wait for user input (first name) and reads the user input when the user hits the Enter key. The statement stores the name read by the method in the string type variable, ***firstName***.

Line 5 displays the string "Hello," followed by the first name stored in the variable ***firstName***.

Step 8: Run the new code.

To run the changed code, save the file and type in ***dotnet run*** at the prompt. Press ***Enter***. You can also press the ***Up Arrow*** key to recall the latest command entered in the terminal.

You will be prompted to enter the first name. After entering the name, press the ***Enter*** key.

The output "Hello," followed by the name, will be displayed in the terminal, as follows:

```
PS C:\WebProjects\Csharp-intro-VSCode> dotnet run
Please enter the first name
George
Hello George
PS C:\WebProjects\Csharp-intro-VSCode>
```

Step 9: Enable debugging in Visual Studio Code.

To be able to debug C# programs that accept user input, you need to change the value of the ***console*** property within the lauch.json file.

Open the lauch.json file and change the value of console property from "internalConsole" to "integratedTerminal," as shown here.

```
18          "cwd": "${workspaceFolder}",
19          "OS-COMMENT5": "For more information about the 'console' field,
20          "console": "IntegratedTerminal",
```

Save and close launch.json.

Step 10: Debug the code.

An important way to identify errors in a program is to stop program execution by putting a break in the statement where you want to stop execution.

Open program.cs and put a break in line 11 by clicking on the left margin, as shown here:

```
 7          static void Main(string[] args)
 8          {
 9          Console.WriteLine("Please enter your first name:");
10          string firstName = Console.ReadLine();
11          Console.WriteLine("Hello " + firstName );
12          }
```

To run the program in debug mode, select ***Run>Start Debugging***.

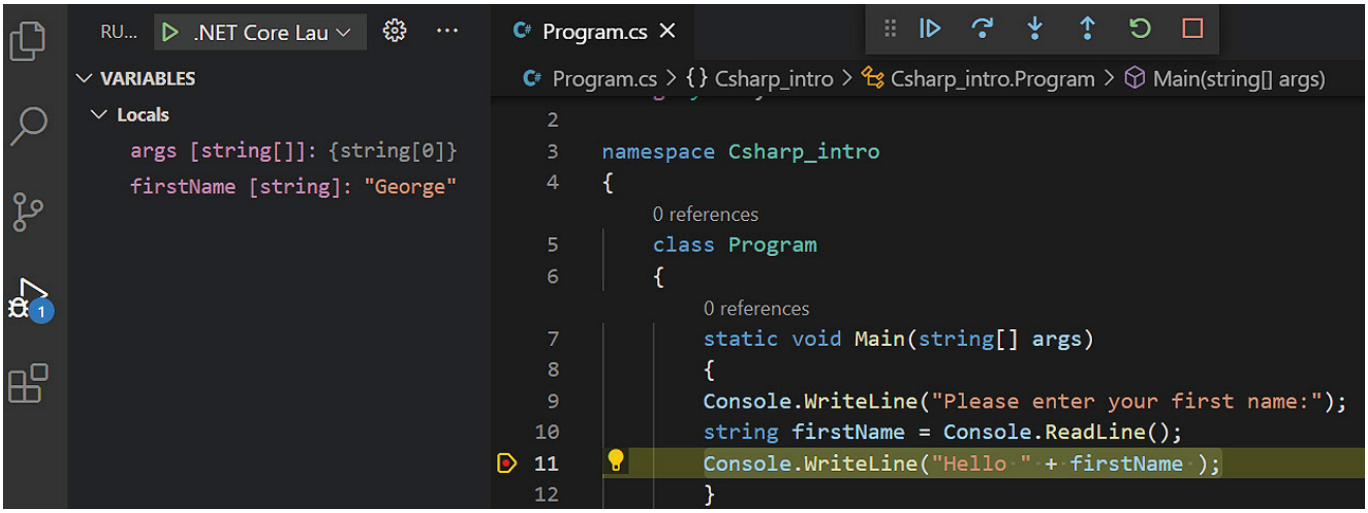
The terminal opens the ***Debug Console*** tab. Click the ***Terminal*** tab to open it.

You will see the prompt to enter the first name. Enter the name at the prompt, and press ***Enter***.

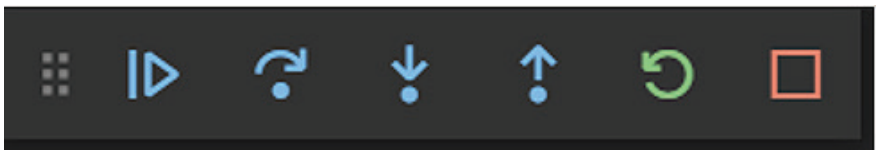
Because there is a break at line 11, the program stops at line 11 without executing it, as shown in the next figure.

You may move the cursor over the variable, firstName, to display its value.

You also may click the ***Run and Debug*** button on the side Toolbar to display values of variables and other information that helps you debug.



Click the ***Continue*** button from the group of debugging buttons on the toolbar to continue the execution of the program.



VS Code executes the Console.Write() statement to display the output in the terminal.

Running C# in VS Community

In Visual Studio, you can create user interfaces using HTML Web Forms, Windows Forms, or ASP.NET Web Forms to let the user interact with the application. However, here you will learn how to create a console application (console type project) that doesn't require you to create the user interface. You will run the application using the built-in terminal.

Creating a C# console-type project in VS Community is similar to how you create other types of projects. The instructions in this section are based on VS Community, version 16.9.

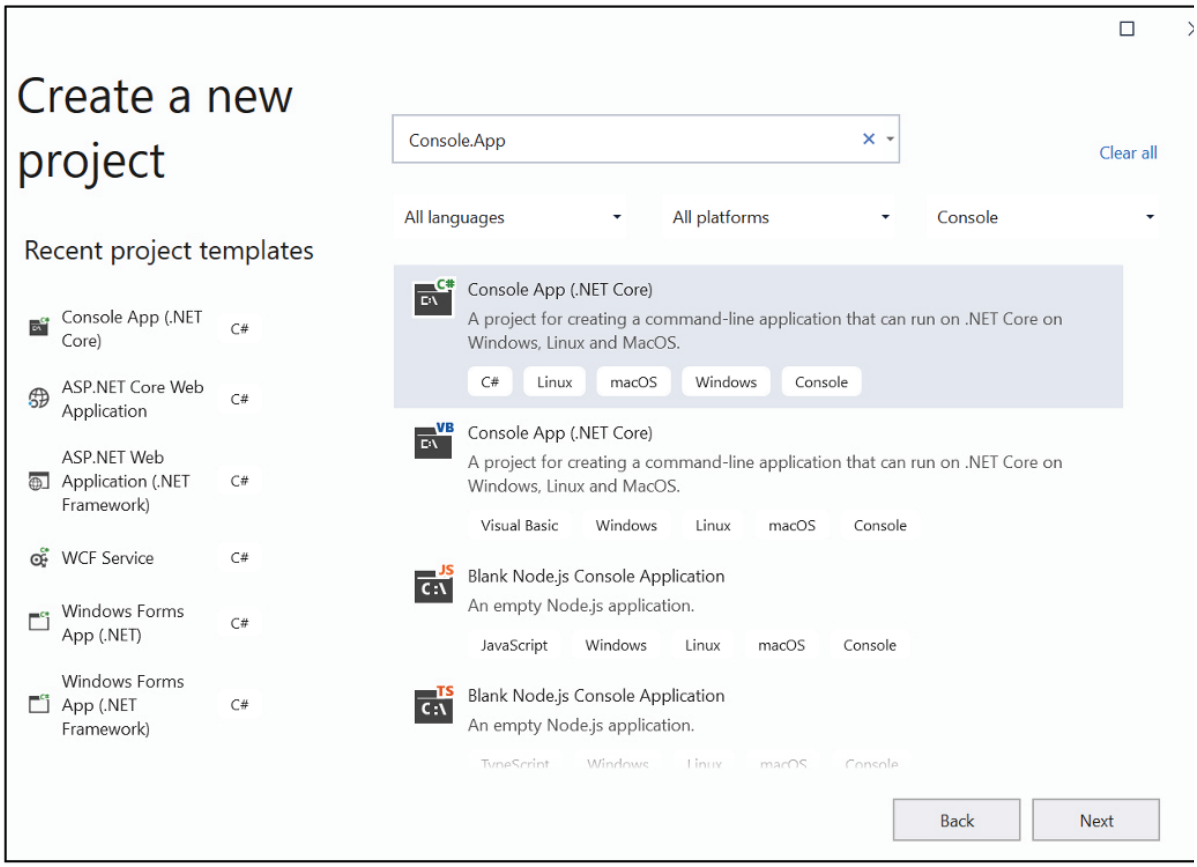
Tutorial 2: Create a C# Console App and Run It in VS Community

Step 1: Create a console application.

Open Visual Studio. You will see the initial window with different options for getting started, including opening a project and creating a project.

Select ***Create a new project***. You will see the ***Create a new project*** window similar to the one shown as follows.

To make it easy to find the template, type in ***Console.App*** in the search box and select ***Console*** for the project type.



Select *Console.App (.NET Core)* for **C#**.

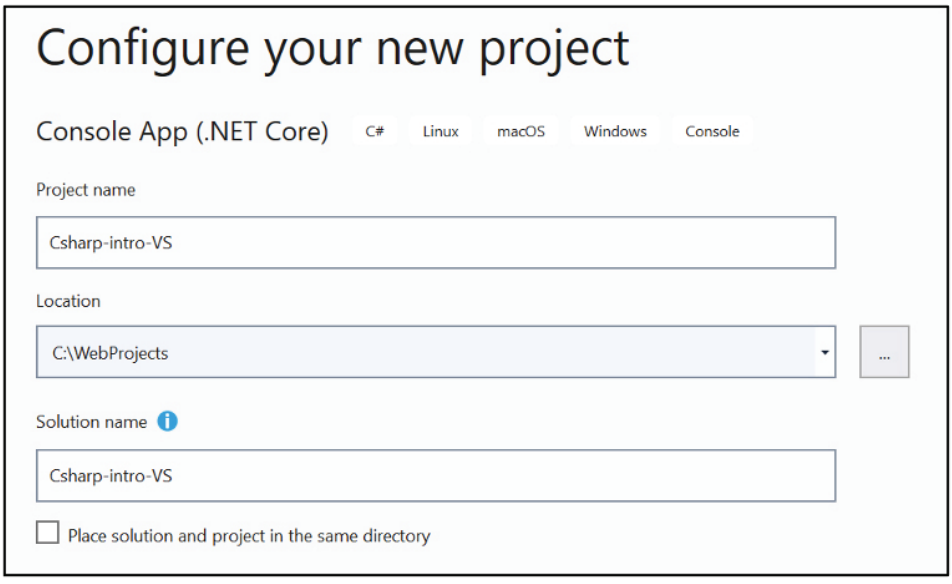
If you do not see the Console App template, you can install it as follows:

Scroll to the bottom of the list of templates and click the ***Install more tools and features*** link.

In the Visual Studio Installer, click the Workloads tab and select *.NET Core cross-platform development*. Click the ***Modify*** button.

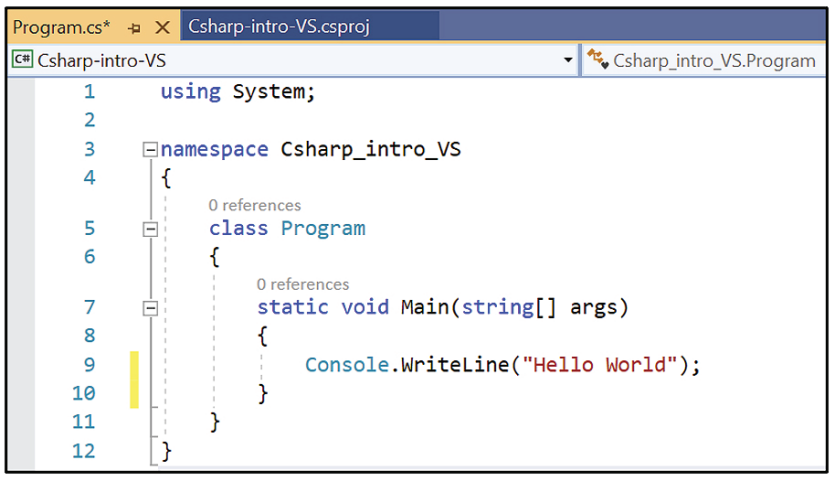
Go back to the initial window, select ***Create a new project*** and select ***Console.App***.

Click ***Next***. You will see the ***Configure your new project*** window, shown next.



Type in ***Csharp-intro-VS*** for the project name, and select ***WebProjects*** folder for the location.

Click ***Create***. Visual Studio creates the new project and the Program.cs file, which contains the code for the ***Hello World*** program, as shown here:



The Main() C# method in the Program.cs file contains the single statement

```
Console.WriteLine("Hello World!");
```

The Console.WriteLine() method displays the specified string ("Hello World").

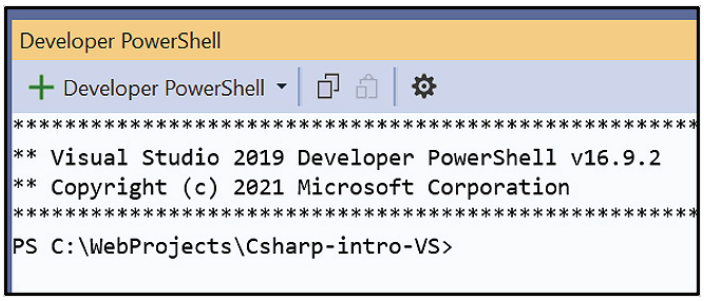
Click any key to close the console window.

If you close the Progams.cs file, you can open it by double-clicking Program.cs in the Solution Explorer window.

Step 2: Run the project.

To run the project, first open the built-in terminal. From the menu bar, select ***View>Terminal.***

Visual Studio opens the terminal, which in turn opens an instance of Microsoft's ***PowerShell*** software, a command-line shell that provides a toolset for task automation and configuration management. The following figure shows the terminal with a command line showing the current directory:



Second, change the current directory to the project directory, Csharp-intro-VS within WebProjects\Csharp-intro-VS. Type in ***CD Csharp-intro-VS*** at the command line prompt and press ***Enter***.

The current directory changes to WebProjects\Csharp-intro-VS\ Csharp-intro-VS, as shown:

```
PS C:\WebProjects\Csharp-intro-VS> CD Csharp-intro-VS
PS C:\WebProjects\Csharp-intro-VS\Csharp-intro-VS>
```

Third, to run the project, type in **dotnet run** at the prompt and press Enter.

You will see the output of the program, "Hello World," displayed in the terminal.

Step 3: Edit the C# code.

To gain insight into writing and editing C# code, replace the existing Console.Write() statement with the three statements shown in lines 3–5 in the following code.

You may use shortcuts like ***CW*** followed by a tab to write the Console.Write() method. Note how the IntelliSense feature helps you complete the statements:

```
1  static void Main(string[] args)
2  {
3     Console.WriteLine("Please enter the first name:");
4     string firstName = Console.ReadLine();
5     Console.WriteLine("Hello " + firstName);
6  }
```

Here is a brief description of the statements:

Line 3 writes the prompt to enter the first name.

In line 4, the Console.ReadLine() method causes the application to wait for user input (first name) and reads the user input when the user hits the Enter key. The statement stores the name read by the method in the string type variable, ***firstName***.

Line 5 displays the string "Hello," followed by the first name stored in the variable ***firstName***.

Step 4: Run the new code.

To run the changed code, save the file and type in ***dotnet run*** at the prompt. Press ***Enter***.

You will be prompted to enter the first name. After entering the name, press the ***Enter*** key.

The output "Hello," followed by the name, will be displayed in the terminal:

```
PS C:\WebProjects\Csharp-intro-VS\Csharp-intro-VS> dotnet run
Please enter the first name
George
Hello George
PS C:\WebProjects\Csharp-intro-VS\Csharp-intro-VS>
```

Step 5: Debug the code.

An important way to identify errors in a program is to stop program execution by putting a break in the statement where you want to stop execution.

Open program.cs and put a break in line 11 by clicking on the left margin, as shown here:

```
 7        static void Main(string[] args)
 8        {
 9        Console.WriteLine("Please enter your first name:"
10        string firstName = Console.ReadLine();
11        Console.WriteLine("Hello " + firstName );
12        }
```

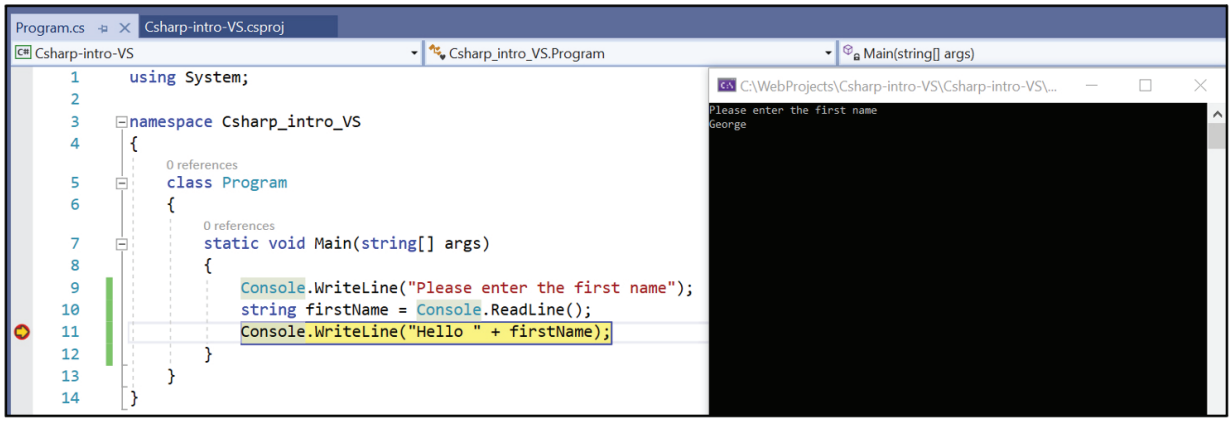
To run the program in debug mode, select ***Debug>Start Debugging***.

You will see the console window with the prompt to enter the first name.

Enter the name at the prompt, and press ***Enter***.

Because there is a break at line 11, the program stops at line 11 without executing it, as shown in the next figure.

You may move the cursor over the variable, firstName, to display its value.



Click the ***Continue*** button on the Toolbar to continue execution of the program.

Visual Studio will execute the Console.Write() statement and display the output in the console window.

Those who are new to programming or new to C# may use online resources like https://www.learncs.org, https://www.w3schools.com/cs, and https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/ to become more familiar with programming concepts and C#.