Introduction to Computer Science in C#

Release 1.0

Andrew N. Harrington
George K. Thiruvathukal

26-July-2015 03:38:09

CONTEXT

1.1 Motivation for This Book

This is really a preface, but the otherwise very capable Sphinx publishing environment that we use is not set up for a separate preface.

1.1.1 Pedagogy

Our first aim is to provide a good introduction and conceptual framework for more computer science, not an encyclopedic coverage of C#. C# will not be most students' only language, or necessarily the most used. Designing and creating algorithms in a particular language is an important skill, requiring ongoing effort, so most of the text is still centered on C#.

C# is an object-oriented language. There is the ongoing argument about when to introduce details of object-oriented programming. We last taught Java, objects first. Students dutifully followed our lead. Later, we saw quick programs that students wanted to write for themselves, that were layered with totally unnecessary and distracting instances of objects.

We have seen less problem with the opposite order, which we use: start off with more procedural programming, then introduce the use of instances of existing classes of objects, and then move to designing classes with instance variables, constructors, and instance methods, and see where they are truly useful. If you prefer, after the chapter on functions you can read the first couple of sections in

Classes and Object-Oriented Programming, that cover defining your own simple objects.

We tend to introduce examples first, and then the general syntax, and then more examples and exercises. Later examples on a subject are sometimes essentially links to documented code that is both directly visible on the web and in the separate download of all of the example source code.

There are review questions at the end of most chapters. The review questions may seem to be in a strange order: Often we invite students to consider a general overarching theme in an early question. In case that was too much to bite off, later questions often explicitly address a specific point that would have been an implicit part of an earlier general question. Sometimes a later more pointed question even gives an answer to part of an earlier question.

Labs are intended as early practice on a subject, with generally small bits requested at a time. They are usually included in the main body of the book soon after the needed background is introduced. There are also larger assignments as some of the appendix sections.

1.1.2 C# and Mono

We have taught introductory programming for many years, through a progression of programming languages. Our last language was Java, still the language of the AP test, which drives so many introductory texts.

We had C# in mind: It is a more modern language. Its designers got to reflect on the glitches with Java, and address them effectively.

The key problem with C# used to be that it was totally a Microsoft language for Windows. Many of our students have their own machines: many are OS-X machines from Apple; some are Linux. We did not want to cut those students off. Nor did we want to limit students to thinking of a computer as a Windows machine. Meanwhile the open source implementation of C#, Mono, has been maturing, along with its tool chains.

While many open-source tools have hackers jumping in to eliminate bugs, and maybe providing enough documentation for a professional, documentation for a beginner is often lacking. This book contributes there, partly in the documentation for Mono's lovely interactive environment csharp, and also for the integrated development environment, Xamarin Studio. We show beginners how to start using

the Xamarin Studio environment, with its large array of features (not all needed by the beginner), and introduce more features as needed.

We aim to end up with a book that provides a solid conceptual framework for beginning computer scientists in the context of the clean, well-established modern language, C#, using multi-platform free and open-source tools, with clear documentation.

We hope that you find this to be a winning combination.

1.2 Resources Online

This book is designed for Comp 170 at Loyola University, Chicago. The materials are available to all on the web. Here are some important web links:

- The course example file, is the *essential resource* to download and unzip onto your machine. The zip file and the folder it unzips to have a long name, introcs-csharp-master-examples. We suggest you *rename the unzipped folder* examples to match later references.

  Computer programs are designed to run on a computer and solve problems. Though the initial problems will be tiny and often silly, they will serve as learning tools to prepare for substantive problems.

- http://introcs.cs.luc.edu is an online text version for your web browser. See also *Downloading and Reading Options* for pdf and epub versions. Except on very geometrically oriented topics, *text-oriented learners may be happiest just reading the book* in one of these formats.

- https://luc.box.com/CSharpVideos is a box.com folder containing all the videos. The numbers at the beginning of the titles are chapter and section numbers from the text. These make it easy to sequence the videos, even though there is not an automatic *playlist*. The listing in box.com takes up several pages. These box.com videos may be

  - streamed, including at full size (though generally after an initial delay),

  - downloaded individually or

- the whole gigabyte folder can be downloaded at once to play later on your machine. (This is a choice on the menu under the page's Folder Options.)

There is a mixture of new high-def videos and older lower resolution 800x600 pixel videos. In that folder see 00README.html, https://luc.box.com/s/2lqak4pbsdcyw08ds3ia, for a description of the differences between the old videos and the latest update of the online book.

For those who learn best with spoken words combined with written words, the videos should be a good start. Even if you use a video for a section, you are encouraged to *review the written text afterward*. Then be aware of the written version for quick reference. The written text may include extra details and exercises, and it will have the latest revisions.

In various formats, be aware of these helpful features:

- We have picked out particularly important words, phrases, and symbols, and put them in our index, which is accessible from the banner of each web page. The banner stays visible if the window is big enough, and if the window is smaller, the baner only appears at the beginning of the page with a dropdown menu for most of the options.

- In a web version on our website, you can use the Search option to look for words, in general. The search link is also accessible from the web page banner. This does *not* work on a downloaded local html copy.

- The web version and pdf versions display mathematical formulas prettily. If you use the local html version when not connected to the internet, you see only the LaTex encoded source for formulas. While offline reading sections with math formulas, we suggest using the pdf version.

- We start with a brief table of contents for the whole book. *In the web versions* you can get the most detailed table of contents for a single chapter by clicking on a chapter title in the main table of contents or a chapter title in the Site drop-down menu in the banner.

- The web version shifts the behavior of the top banner depending on screen resolution, so it adapts for anything from large monitors to small mobile devices. With a wide screen the banner stays at the top of your screen, and shows all the standard internal links and drop-down menus. On a narrower screen the banner is only at the top of each webpage, and scrolls off the screen; the only direct link is to the main table of contents, while the rest of

the links of the wide version can be accessed via the icon at the right side of the banner.

Here are further links that may be useful in our repository:

- https://github.com/LoyolaChicagoBooks/introcs-csharp/ is the home page for the repository of all the sources for the book. To read this book, you do *not* need to go to that URL, but if you do, the home page gives you an idea of what *updates have been made recently* to the book or accompanying examples. Since improvements are made on an ongoing basis, the notes about recent changes may be useful to you.

  The multiple production versions are generated largely by Sphinx software from the common set of sources in the repository. The sources are largely plain text files.

- https://github.com/LoyolaChicagoBooks/introcs-csharp-examples/ is another repository containing the *latest versions* of the source code files. You can quickly browse and view individual files under the Source tab. Example file links throughout this text refer to these repository files.

1.3 Downloading and Reading Options

Most readers will want to download the C# examples in ZIP format. More experienced readers (with previous background) may prefer using GitHub. ()

Table 1.1: Example Source Code and Videos

Description	URL
C# Examples (as ZIP)	https://github.com/LoyolaChicagoBooks/introcs-csharp-examples/archive/master.zip
Videos	https://luc.box.com/CSharpVideos

We offer the book in HTML, PDF, and EPUB formats. Kindle users should use EPUB, which can be converted to MOBI format. (We plan to offer it soon through our main site.)

Table 1.2: Alternate Book Formats

Format	URL
Web Site	http://books.cs.luc.edu/introcs-csharp/
Web Site (offline ZIP)	http://books.cs.luc.edu/introcs-csharp/download/html.zip
PDF (US Letter)	http://books.cs.luc.edu/introcs-csharp/download/comp170.pdf
PDF (7x9 Book)	http://books.cs.luc.edu/introcs-csharp/download/comp170book.pdf
EPUB (Experimental)	http://books.cs.luc.edu/introcs-csharp/download/comp170.epub

> **Warning:** The following is for developers only!

The book source code and examples are all maintained on GitHub. We welcome pull requests and will acknowledge any helpful and constructive contributions.

Table 1.3: GitHub

Description	URL
C# Examples	https://github.com/LoyolaChicagoBooks/introcs-csharp-examples
Book Source	https://github.com/LoyolaChicagoBooks/introcs-csharp

1.4 Computer Science, Broadly

We intend this book as an introduction to computer science, with a focus on creating problem solutions in the C# programming language. We should not jump in too quickly. You can get lost in our details and miss an idea of the much larger breadth of computer science.

1.4.1 Information Processing

Computer Science is the study and practice of information processing. This can take many forms. Many forms appear in electronic computers, but information processing takes place in many other contexts, too:

In the early days of electronic computers, the information was largely numerical, calculating mathematical functions.

Later analyzing textual information has become much more important, for instance: What can you tell about the severity of the current flu outbreak by analyzing the phrasing in Google searches?

Images are analyzed: What can a satellite image tell you about the distribution of drought?

Sounds: how do you convert verbal speech accurately into written sentences?

DNA holds information that our bodies process into proteins.

Our brain chemicals and electronic signals process information. There is rich interplay between cognitive scientists and computer scientists modeling problem solving in the brain with *neural nets* on a computer, sometimes to better understand brains and sometimes to better solve problems on an electronic computer.

Economic systems are becoming better understood in terms of the flow of information.

The *computer* doing computations and processing can be a familiar *electronic* computer, but it can be genes or brain chemicals, or a whole society as its economy adapts.

1.4.2 Algorithms

Algorithms are at the heart of traditional problem solving . An *algorithm* is a clearly expressed sequence of steps leading to a result in a finite amount of time.

A recipe for baking a pound cake is an algorithm.

Such an algorithm has useful concepts that we use later in computer programming:

- A named sub-problem: your recipe may include the instruction "Beat 4 eggs." The recipe probably says no more about it, but this is *shorthand*, a name for a simpler sequence of steps, an algorithm like:

```
Beating Any Number of Eggs
--------------------------
1. Get a bowl large enough for the eggs.
2. For each egg:
   a. Crack its shell on the edge of the bowl.
   b. Add the contents of the shell to the bowl.
3. Mix the eggs with a whisk.
4. Continue with step 3 until you cannot
   distinguish the white and yolks.
```

- Parameters: The egg beating instructions work, in general, for any number of eggs. To use these instructions for a *particular* pound cake, you must supply a specific value to use to make the general instructions become clear. The pound cake recipe that uses the egg beating instructions, uses the number **4** as the actual data.

- Repetition: The instructions for cracking an egg are not *written* repeatedly, for every egg. The instruction is stated once, and we are told how long to go on: for each egg in step 2. Step 4 says when to stop repeating step 3.

1.4.3 Data Representation

A recipe represents data by words that get processed by a human reader. Machines have used different representations. One of the earliest adding machines, the Pascaline, http://en.wikipedia.org/wiki/Pascal's_calculator, represented numbers by the angle of rotation of interlocked gears. An abacus uses the positions of groups of sliding beads to represent digits. The Jacquard loom, http://en.wikipedia.org/wiki/Jacquard_weaving, used cards with each row of holes punched in them indicating which warp threads are raised and which lowered when a cross thread is woven in.

In modern electronic computers the most basic bit of data (actually called a *bit*) is held by two-state switches, often in the form of a higher voltage vs. a grounded state. The symbolic representation is often 0 vs. 1. This symbolism comes from the representation of integers in *binary notation*, also called *base 2*: It is a place value system, but where each place in a numeral is a 0 or a 1 and represents a

power of two, so 1101 in binary can be viewed in our decimal system as $(1)2^3 + (1)2^2 + (0)2^1 + (1)2^0 = 8 + 4 + 0 + 1 = 13$.

Computer hardware can only handle a limited number of bits at a time, and memory space is limited, so usually integers are stored in a limited space, like 8, 16, 32 or 64 bits. We illustrate with the smallest, 8 bits, called a *byte*. Since each bit has two possible states, 8 bits have $2^8 = 256$ possible states. Directly considered as binary numerals, they represent 0 through $2^8 - 1 = 255$.

We also want to represent negative numbers, and have about half of the available codes for them. An 8-bit signed integer in *twos complement* notation represents 0 through $2^7 - 1$ just as the unsigned numbers do. These are all the 8-bit codes with a leading 0. A negative number n in the range $-2^7 = -128$ through -1, is represented by what would be the unsigned notation for $n + 2^8$. These will be all the 8-bit codes with a leading 1. For example -3 is represented like unsigned 256 - 3 = 253: 11111101 in binary.

Limited precision approximation of real numbers are stored in something like scientific notation, except in binary, roughly $\pm(m)2^e$, with a sign, mantissa m and exponent e. Both e and m have fixed numbers of bits, so the limited options for the mantissa restricts the *precision* of the numbers, and the limited options for the exponent restricts the *range*. Data on these limits for different sized numeric codes is in *Value Types and Conversions*.

Once you have numbers, all sorts of other kinds of data can be encoded: Characters like on your keyboard each have a numerical code associated with each one. For example the unicode for the letter A is 65. Images are often represented as a sequence of colored pixels. Since the human eye is only sensitive to three specific colors, red, green, and blue, a pixel is represented by a numerical intensity for each of the three colors.

1.4.4 Instruction Representation

Besides the data, instructions need a representation too, and an agent to interpret them. In the earliest electronic computers the two-state switches were relays or later vacuum tubes, and the machine was *manually rewired* when a new set of instructions/program was needed. It was a great advance in the 1940's when the instructions also became symbolic, represented by binary codes that the computer could recognize and act on,

http://en.wikipedia.org/wiki/Von_Neumann_architecture. This code is called *machine language*. With machine language the instructions became a form of data that could be stored in computer memory. We distinguish the *hardware* on which programs are run from the stored programs, the *software*. The *architecture* of the hardware determines the form and capacities of the machine language, so machines with a different hardware architecture are likely to have distinct machine languages.

Biologists have a fair idea of how protein sequence data is encoded in DNA, but they are still working on how the DNA instructions are encoded controlling which proteins should be made when.

In this book we will not be writing instructions shown as sequences of 0's and 1's! Some of the earliest programs were to help programmers work with more human-friendly tools, and an early one was an *assembler*, a program that took easier to understand instructions and automatically translated them into machine language. An example assembler instruction would be like

```
MOV 13, X
```

to move the value 13 to a storage location identified by the name X.

Machine instructions are very elementary, so programming was still tedious, and code could not be reused on a machine with a different architecture.

The next big step past assembler was the advent of *high level* languages, with instructions more like normal mathematical or English expressions. Examples are Fortran (1954) and Cobol (1959). A Fortran statement for calculating a slope like

```
S=(Y-V)/(X-U)
```

might require seven or or more machine code instructions.

To use a Fortran program required three steps: write it (onto punch cards originally), compile it to machine code, and execute the machine code. The compiler would still be architecture specific, but the compiler for an architecture would only need to be written once, and then any number of programs could be compiled and run.

A later variant for executing a high-level language is an *interpreter*. An interpreter translates the high-level language into machine code, and immediately executes it, not storing the machine code for later use, so every time a statement in the code is

executed again, the translation needs to be redone. Interpreters are also machine-specific.

Some later languages like Java and C# use a hybrid approach: A compiler, that can run on any machine, does most of the work by translating the high-level language program into a low-level *virtual machine* language called a *bytecode*. This is not the machine language for any real machine, but the bytecode is simple enough that writing an interpreter for it is very easy. Again the interpreter for the bytecode must be machine-specific. In this approach:

Program source => COMPILER => bytecode => INTERPRETER => execution

1.4.5 Program Development Cycle

The easiest way to check your understanding of small new pieces of C# is to write a highly specified small program that will be sure to test the new ideas. That is totally unlike the real world of programming. Here is a more realistic sequence:

1. Clients have a problem that they want solved.

2. They connect with software developers.

3. The clients discuss the needs of their users.

4. The software developers work with them to make sure they understand the desired deliverables, and work through the design decisions and their trade-offs.

5. Software developers start building and testing and showing off the core pieces of the software, and build on out.

6. The clients may not have a full idea of what they want and the software developers may not have a full idea of what is feasible, and seeing the latest version leads both sides to have a clearer vision. Then the previous process steps are repeated, iteratively refining the product.

7. After a production version is out, there may be later versions and error fixes, again cycling back to the earlier steps.

Note that very important parts of this process are not centered on coding, but on communicating clearly with a possibly non-technical client. Communication skills are critical.

1.4.6 Key Computer Science Areas

Most of the introduction so far has been about key concepts that underlie basic programming. Most of the parts so far about electronic computers could have been written decades ago. Much has emerged since then,

- the Internet

- the development of economical multi-processor machines distributing computation into many parallel parts

- the massive explosion of the amount of information to be stored from diverse parts of life

- the coming *Internet of things*, where sensors are coming to all sorts of previously "dumb" parts of the world, that now can be tracked by GPS and reacted to in real time.

- Computers are now embedded in all sorts of devices: toasters, thermostats,.... Automobiles of today have more computing power embedded in various devices than early mainframe computers.

We conclude with a brief discussion of some of the other organizing principles of computer science.

Communication As the world is criss-crossed with media transmitting gigabytes of data per second, how do we keep the communication reliable and secure?

Coordination With multiple networked entities, how do we enhance cooperation, so more work is done in parallel?

Recollection As the amount of data stored skyrockets, how do we effectively store and efficiently retrieve information?

Evaluation How do we predict the performance and plan the necessary capacity for computer systems? The most spectacular recent public failure in this area was the rollout of the federal Affordable Care website.

Design How do we design better/faster/cheaper/reliable hardware and software systems? What new programming languages will be more expressive, lead to fewer time-consuming errors, or be useful in proving that a major program never makes a mistake? Errors in programs controlling machines delivering radiation for cancer treatment have had errors and led to patient death.

Hardware changes can be evolutionary or revolutionary: Instead of electric circuits can we use light, quantum particles, DNA...?

Computation and Automation What can we compute and automate? Some useful sounding problems have been proven to be unsolvable. What are the limits?

A detailed discussion of these principles and the breadth of importance of computer science can be found at http://denninginstitute.com/pjd/GP/GP-site/welcome.html.

For an alternate general introduction to programming and the context of C# in particular, there is another free online source, Rob Miles' C# Yellow Book, available at http://www.csharpcourse.com. Note that it is written specifically for Microsoft Windows use, using Visual Studio software development environment, which works only on Windows machines, and costs a lot if you are not a student.

The *Lab: Editing, Compiling, and Running with Xamarin Studio* will introduce an alternative to the Microsoft environment: Xamarin Studio and Mono, which are free, open-source software projects that make C# available for multiple platforms: Windows, Mac, or Linux machines. With a substantial fraction of students having their own machine that does *not* run Windows, this flexibility is important.

1.5 Chapter Review Questions

1. What is an algorithm?

2. Are computers the only things that can perform the operations in an algorithm?

3. What are important types of information?

4. What are very distinct important ways that information is stored?

5. What is a *bit*?

6. Look at our brief mention of a Jaquard loom. In a Jacquard loom, how is a single bit of information stored? What two choices does it represent?

7. Does a computer generally execute a C# program directly? Briefly, what transformation happens first?

8. Is coding the only thing important in developing a real-world application? What are other important parts of the process?

9. Is computer science all about writing computer programs? What are some important large areas of concern in computer science?

C# DATA AND OPERATIONS

2.1 A Sample C# Program

As a start let us consider a ridiculously simple problem and a program to solve it. Suppose you paint the walls of rooms in one color and the ceiling in another, and you want to calculate the size of the areas to cover with paint. For simplicity ignore doors. What data do you need to start with? Clearly the dimensions of the room. Suppose we consider modern houses where the height of the room is predictably 8 feet, so the new starting data is just the length and width of the room.

You need to

1. Obtain the length and width from the user.

2. Calculate the wall area and ceiling area.

3. Let the user know the results.

This is a very simple programming pattern: data in, calculate results, output results. In this case the calculations in the middle are very easy.

In the examples that you should have downloaded is a first simple program, painting/painting.cs.

Here is what it looks like when it runs, with the user typing the 20.5 and the 10:

```
Calculation of Room Paint Requirements
Enter room length: 20.5
Enter room width: 10
```

```
The wall area is 488 square feet.
The ceiling area is 205 square feet.
```

This is not very exciting, but it is a simple place to start seeing basic program features. We will refer back to this sample run while discussing the program. Here is the text of the program:

```csharp
1   using System;
2
3   class Painting
4   {
5       static void Main()
6       {
7           double width, length, wallArea, ceilingArea;
8           string widthString, lengthString;
9           double HEIGHT = 8;
10
11          Console.WriteLine ( "Calculation of Room Paint Requirements");
12          Console.Write ( "Enter room length: ");
13          lengthString = Console.ReadLine();
14          length = double.Parse(lengthString);
15          Console.Write( "Enter room width: ");
16          widthString = Console.ReadLine();
17          width = double.Parse(widthString);
18
19          wallArea = 2 * (length + width) * HEIGHT; // ignore doors
20          ceilingArea = length * width;
21
22          Console.WriteLine("The wall area is " + wallArea
23                          + " square feet.");
24          Console.WriteLine("The ceiling area is " + ceilingArea
25                          + " square feet.");
26      }
27  }
```

This section gives an overview of a working program, even if all the explanations do not make total sense yet. This is a first introduction of concepts and syntax that gets fully explained in further sections.

Do not worry if you not totally understand the explanations! Try to get the gist now and the details later.

The different colors are used in modern program editors to emphasize the different uses of the parts of the program.

We give a line by line explanation:

```
using System;
```

The C# environment supplies an enormous number of parts that you can reference. Nobody is familiar with all of them. If you had to make sure you always used names that did not conflict with other names supplied, you would be in trouble. To avoid this C# has *namespaces*. The same name can be used in different namespaces without conflict. The central standard namespace is System. We will always include this first line, using System;.

Lines 2, 10, 18, and 21 are blank. This is merely for the human reader to separate sections visually. The computer ignores them.

```
class Painting
{
```

A basic unit in C# is a *class*. Our code sits inside a class. Each class has a *heading* with class followed by a name. This class is Painting. After the heading comes a *body* delimited by braces. The opening brace { in line 4, is matched by the closing brace } on the last line of the program.

```
static void Main()
{
```

A class is broken up with chunks called *functions* or *methods*. Each has a *heading*. C# allows the currently popular programming paradigm called *object-oriented programming*, where classes generally describe new kinds of objects. This is useful in complicated situations, but we start more simply with the older *procedural* programming. Unfortunately for now, the more common situation is with objects, so a function that does *not* involve such new objects must be marked specially as static.

Functions can be like in math, where they produce a function value for later use. In C# they can also just *do* something (like write to the screen), and not produce a value for later use in the program. To show that no function value is produced, the word void is used.

Every program must start running somewhere. In C# that is at a function with name Main. So our program starts running here. This syntax for this function

needs to start just like here, with `static void Main`.

Even though this is not a mathematical function producing a value, a function in C# must be followed by parentheses `( )`.

After the function heading comes a *body*. Like with a class, a function body is delimited by braces. The opening brace here is matched by the closing brace on the second to last line of the program.

```
double width, length, wallArea, ceilingArea;
```

A program works with data of many different possible types. One type is `double`. A `double` can hold an approximate numerical value, including a possible fractional part.

To refer to data in a program we use names called *variables*. This line says that `width`, `length`, `wallArea`, and `ceilingArea` are all the names for variables that can hold a `double` value. We will assign values to these variables later.

This line is a *declaration* statement. Most statements in C#, like this one, end with `;` - a semicolon.

```
string widthString, lengthString;
```

This is another declaration. This time the type of the variables is `string`, which means a sequence of characters, like a line you might type at the keyboard.

```
double HEIGHT = 8;
```

Here is another declaration for a `double`, looking slightly different. In this case we follow a convention, using all capital letters, to suggest that the value of `HEIGHT` will be constant (unchanging), and we assign its value at the same time with `= 8`. This naming of constants is not strictly necessary, but it makes the program's intention easier to follow.

```
Console.WriteLine ( "Calculation of Room Paint Requirements");
```

`Console` refers to the terminal or console window where text output appears for the program. One of the things you can do with the Console is `WriteLine`, to write a line. The period between `Console` and `WriteLine` indicates `WriteLine` is a named part of the `Console`. This `WriteLine` is a function. Like in math, it can have a parameter in parentheses. While you are used to a

parameter for a function in math being a number, functions in C# are much more general. A function can be defined with any type of parameters. Here the parameter is a string, `"Calculation of Room Paint Requirements"`, delimited by the quotes at either end. Notice that the contents of this string appear at the start of the screen output displayed for this program. The program did **write** this **line**.

```
Console.Write ( "Enter room length: ");
```

This statement is similar to the last one, except that it uses `Write` rather than `WriteLine`. The `WriteLine` function wrote a whole line - see that the output next *after* the `WriteLine` statement started on the next line. Here `Write` does not advance the printing position to the next line after it.

This statement serves as a *prompt*: letting the user know that information is being requested (a room length).

```
lengthString = Console.ReadLine();
```

Here is where the program takes in the information requested from the user. Its action is actually right to left: `Console.ReadLine` is another function available with the `Console`, that reads a line typed in by the user on the keyboard. Here in the sample run, on the same line as the prompt string (because of the previous `Write`, not `WriteLine`), the user types `20.5` and the Enter or Return key.

In the sample run, the value produced by the `Console.ReadLine` function is these four characters `20.5`.

Recall that `lengthString` was declared as a variable to hold a string. The `=` indicates an *assignment statement*. It is an *assignment* of the value on the right of the equal sign to be the current value of the variable on the left of the equal sign. In the sample run, this would mean that the variable `lengthString` would end up holding the value `"20.5"`. Though these characters happen to look like a number, any sequence of characters can be typed. The `Console.ReadLine()` function produces this sequence of characters as a *string* type.

```
length = double.Parse(lengthString);
```

Of course we want to interpret the user's input as a number in order to do our arithmetic. This line makes the conversion between the types.

It is another assignment statement (with the `=`). We are assigning to the variable `length`, which we declared as a `double`. The value assigned comes from

the expression on the right of the =, `double.Parse(lengthString)`. The function `double.Parse`, is just the one we want, it takes a string parameter `lengthString` containing the string from the user input, and the value produced is the corresponding `double` number. In the sample run that assigns to `length` the value 20.5.

```
Console.Write( "Enter room width: " );
widthString = Console.ReadLine();
width = double.Parse(widthString);
```

These lines are analogous to the previous three lines: give a prompt for the user; get the user response; convert it to a `double`, and assign to a variable (`width` in this case). In the sample run the variable `width` is assigned the value 10.

At this point we have all the data we need from the user. The next part is the brief calculations of results:

```
wallArea = 2 * (length + width) * HEIGHT; // ignore doors
ceilingArea = length * width;
```

At the end of the first line is a *comment*. It starts with `//` and ends at the end of the same line. It is ignored by the compiler. It is there for humans, hopefully to add something that helps understanding of the program.

We have two assignment statements. The values to assign are given by arithmetic expressions on the right side of the equal signs. It looks pretty much like regular math, except in math class you may be used to only having one letter names for variables, unlike `length`, `width`, and `HEIGHT`.

The tradeoff for allowing multiple character names is that multiplication must have an explicit operation symbol. The symbol used for multiplication in C# is `*` an asterisk. The `+` and parentheses serve their normal mathematical purpose. In the sample run, the value of `2 * (length + width) * HEIGHT` is

```
2 * (20.5 + 10) * 8
```

which simplifies to 488.

With the sample run, `ceilingArea` would get the value 20.5 * 10, or 205.

```
Console.WriteLine("The wall area is " + wallArea
                 + " square feet.");
```

This is a single statement. Line endings act just like a space in C#. The statement ends with the semicolon on the second line.

Again `Console.WriteLine` will print something to the computer console. This time the string printed is more complicated: It starts off with the literal string `"The wall area is "`, but then we want to print out the calculated result. The `+ wallArea` allows that. The `+` sign after the string is not a mathematical operator here. Coming after a string, it has a special string meaning: It converts the next part `wallArea` to be a string. In the sample run that would be converting the `double` value 488 to be the string `"488"`. The plus sign then "adds" the strings in a manner appropriate for strings, *concatenating* them. That means joining them together, end to end.

The `+ " square feet."` then tacks on the last part to the string. In the sample output you see what is printed:

```
The wall area is 488 square feet.
```

sandwiching the value taken from the variable `wallArea` between two literal string, given in quotes.

```
Console.WriteLine("The ceiling area is " + ceilingArea
                    + " square feet.");
```

This statement behave like the previous one, except with different quoted strings and the value of a different variable. See the sample output.

```
    }
}
```

Finally we have the matching closing braces marking the end of the body of the `Main` function and the end of the body of the `Painting` class.

Of course the display would look different if the user entered different data. Here is what is displayed when the user enters length 15 and width 6.5:

```
Calculation of Room Paint Requirements
Enter room length: 15
Enter room width: 6.5
The wall area is 344 square feet.
The ceiling area is 97.5 square feet.
```

The blank space in the program was there to aid human understanding. In a C#

program *whitespace* is any consecutive combination of spaces, newlines, and tabs. C# treats any amount of whitespace just the same as a single space, except inside quoted strings, where every character is important.

Also the compiler does not require whitespace around special symbols like {};().=*+,. Hence the painting/painting.cs program above would be just as well translated by the compiler if it were written as:

```
using System;class Painting{static void Main(){double width,length
,wallArea,ceilingArea;string widthString,lengthString;double HEIGHT
=8;Console.WriteLine("Calculation of Room Paint Requirements");Console.
Write("Enter room length: ");lengthString=Console.ReadLine();length=
double.Parse(lengthString);Console.Write("Enter room width: ");
widthString=Console.ReadLine();width=double.Parse(widthString);wallArea
=2*(length+width)*HEIGHT;ceilingArea=length*width;Console.WriteLine(
        "The wall area is "                +        wallArea
+" square feet.");Console.WriteLine
      ("The ceiling area is "
              +ceilingArea+
" square feet.");}}
```

Since human understanding is very important, we will emphasize good whitespace conventions, and expect you to use them.

Next we give you an even simpler program to run in the lab. After that we return to how you can get the painting program to run on your computer.

2.2 Lab: Editing, Compiling, and Running with Xamarin Studio

This first lab is aimed at taking you through the end-to-end process of writing and running a basic computer program with the Xamarin Studio environment. As with all things in life, we will learn in this lab that becoming a programmer requires you to learn a number of other things along the way.

In software development/engineering parlance, we typically describe a scenario as a *workflow*, which can be thought of as a series of steps that are possibly repeated. The workflow of programming can loosely be defined as follows:

1. Use a text editor to write your source code (human readable).

2. Compile your code using the Software Development Kit (SDK) into object code.

3. Link your object code to create an executable. (There are other kinds of results to produce, but we will start with the idea of an executable program to keep things simple.) The default is to have an executable program created with compilation, automatically.

4. Run your program. Even for the most seasoned developers, your program may not work entirely right the first time, so you may end up repeating these steps (debugging).

These steps can all be done with different tools. Many find it simpler to have an integrated tool, like Xamarin Studio, that does them all in the same place, and automates the steps that do not need human interaction!

If you are doing this on your own machine, make sure you have Mono and Xamarin Studio installed as in *Development Tools*.

Other tools are available, like the development environment Visual Studio (from Microsoft, focused on Windows).

Understanding the lower level tools that accomplish each step is important, but we defer a discussion to get you going with Xamarin Studio.

2.2.1 Goals

Our primary goal to create and understand an Xamarin Studio setup that you can use to do all of the remaining homework assignments and labs for this course.

2.2.2 Steps

Xamarin Studio files and interactions are organized hierarchically. At a low level are individual C# source code files. One, or possibly more, are used for a particular *project*. Multiple projects are gathered together under a single *solution*. Xamarin Studio deals with one solution at a time, though you can separately create multiple solutions. The simplest thing is to create a *single solution for this course*, and put each of the projects that you create in that one solution. You can keep adding onto previous efforts without having to start over with a new solution each time.

We start by creating a *solution* with a *project* in it. The images are from a Mac. Windows versions should be similar.

1. Open Xamarin Studio, in the appropriate way for an application in your operating system. It should be in the Start menu for Windows. Using Spotlight is quick on a Mac.

2. You get a Welcome screen. Toward the upper left corner is a link for New Solution. Click on it. Alternately you can follow the path through the menus: File -> New -> Solution...
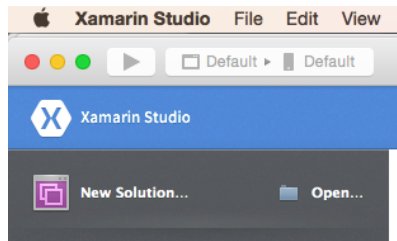
You should get a popup New Project Window, with a banner. "Choose a template for your new project". It may initially show the wrong template choice, like in the left column shown here:
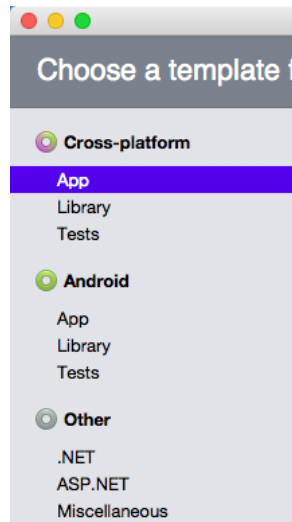
Make sure the template selection gets set to .NET (under Other) as in:

The center part of the screen then shows further choices appropriate to a .NET project. Make sure **Console Project** is selected, and the language shown in the highlighted line should be set to C#.

Then click Next.

3. The banner on the next form is "Configure your new project". The image shows the initial state for the author:

Follow the order below to modify the form. Later parts may not be visible until you do the previous parts. The image shows the final form as filled out by Dr. Harrington.

- Enter **hello** in the Project Name field, for the name of the project.

- If you want to keep all your work together, we suggest that you change the name of the solution from the copied "hello" and enter any name you like: We recommend **work**, which will make sense for all your work for the course.

- On *your* computer the Location field should start off showing a Projects folder under *your* home folder, (not the author's home folder as in the image). You can leave it as is or change it if you like.

- Make sure *Create a project within the solution directory* is checked in the box under the Location.

- You may skip version control for now, though it is very handy.

- After the three fields are filled in, the Create button in the lower right should become active. After checking your entires, click on Create.

You now have created a solution in Xamarin Studio, with one project inside it. Later we can add further *projects* to *this solution*.

4. Look at the Xamarin Studio window that appears. It should have two main sub-windows or "Pads" as Xamarin Studio calls them. A narrow one on the left is the Solution Pad, containing a hierarchical view of the solution. If

you somehow close a pad, you can recover it going to the main Xamarin menu: View => Pads => Solution.

In the Solution Pad you should see your solution name at the top and the hello project under that. Folders have a little triangle shown to their left. You can click on the triangle. A triangle pointing down means the inside of the folder is displayed. A triangle pointing to the right means the contents are not being displayed. Listed under hello are References, Packages and Properties, that we will ignore for now. Below them is the line for the automatically generated sample code file Program.cs. The file should also appear in the Edit Pad to the right.

5. Program.cs should be selected in the Solution Pad, as shown above. Change the selection by clicking on hello. At the right end of the highlighted hello entry you should see an icon with a small gear and a triangle. Click on it to get the context sensitive popup window. When selected, most entries in the Solution Pad should show this icon, allowing you to open its context sensitive menu.

6. Bring up the context menu on the hello project in the Solution Pad. Select Run Item.

7. Here Xamarin Studio combines several steps: saving the file, compiling it into an executable program, and starting running it if compilation succeeded. With the canned file it should succeed! You see a Console window something like

Here you see the output of this simple program. On Windows you can follow the instructions and press the space or Enter key.

Note: This console window must be closed before you can run a program again. If you go to run a program and the Run Item line is grayed out, look for the unclosed terminal window you still have!

---

8. On Microsoft Windows, pressing spacebar or Enter kills the window. **On a Mac only, this makes the window gets two more lines, but still be visible:**

You have to actively close the Mac terminal window by clicking the red window closing button, or using the keyboard, with Command-W. You can either do that now, or replace the previous spacebar/Enter step with this action.

9. This one time it is convenient to get the immediate practice of running this automatically generated program file, `Program.cs`. Hereafter this file is an annoyance. The file name and contents are always the same, and not useful. You would need to redo the whole code for your own program. A general approach for all your projects is to *delete* this file and put in a file of your own:

   • Make sure Program.cs is selected in the Solution Pad. You save a step by closing the Edit Pad for Program.cs, clicking on the X in the Program.cs tab at the top of the Edit Pad.

   • In the Solution Pad open the context sensitive menu for Program.cs, and select Remove.

- You get another popup, with the *wrong* (rightmost) button selected, Remove From Project. Instead select the *left* button, **Delete**, as in the image below. Otherwise the file is left in the hello folder, but it is just not listed as being in the project. Each project is associated with a folder, but not everything in the folder may be cataloged by Xamarin as part of the project.

- If you forgot to close the Edit Pad tab containing Program.cs earlier, you can still do it – just say not to save changes to the file when asked.

10. To get in code that you want, there are several approaches. The one we take now is to start from a completely new empty file: Pop up the context sensitive menu for the hello project. Select the submenu Add... and then New File....

11. In the popup New File Dialog Window, it is likely that the wrong kind of file is highlighted (likely Empty Class).

Click on **Empty File** instead. Change the Name field from EmptyCSharp-File to hello.cs. Always remember the ".cs". Click the New button.

12. This should add hello.cs to the hello project and open an editing window for hello.cs. The file should have no text.

If it does have text, delete this file, too, and go back to the previous step and

be sure to select **Empty File**, not Empty Class.

Much like in most word processors type in (or paste) the following code. This is actually an equivalent *Hello, World!* program to the automatically generated one, but it is a bit shorter. It only introduces the syntax we actually *need* at the beginning, and will be discussing more shortly. The automatically generate line numbers are not a part of the text:

```csharp
1   using System;
2
3   class Hello
4   {
5      static void Main()
6      {
7          Console.WriteLine("Hello World!");
8      }
9   }
```

This program is deliberately simple, so you can type it into the text editor quickly and become familiar with how to create, edit, and save a program. Here is how it looks pasted or typed into the editor pad:

13. You can run the project just as before. You should ge the same result, unless you made a typing error. In that case look for what does not match, fix it, and try again. Remember to close the execution Console Window.

14. Now try a bit of editing: Look at the program to see where output came from. Change what is printed and run it, but don't eliminate the output console window for now (so you can show it off).

15. Now grab the instructor or teaching assistant so they can perform a quick inspection of your work and check it off (including the varied message printed).

Labs need to be completed to receive credit. If you are unable to make class on a lab day, please make sure that you complete the work and demonstrate it by the beginning of the next lab.

At this point, you have accomplished the major objective for this introductory lab: to create a Xamarin Studio project, and enter, compile, and run a C# program.

For further reinforcement

1. Can you make a new program variant print out two *separate* lines? This is possible with an added statement. It is also possible changing one statement, but not adding an additional statement.

2. Download and install Mono Software Development Kit and Xamarin Studio on your home computer or laptop.

3. You can now add further projects to your *current* solution. To add a new project in your solution, in the Solution Pad open the context sensitive menu for the whole solution (top line), select Add, and in the submenu select New project, and you go to the Project Template form. It should remember your last settings, but do check: Later we will sometimes want a Library instead of a Console project.

   Complete the remaining parts in the same way as before, giving a new name for the project.

2.3 Arithmetic

We start with the integers and integer arithmetic, not because arithmetic is exciting, but because the symbolism should be mostly familiar. Of course arithmetic is important in many cases, but C# is often used to manipulate text and other sorts of data.

2.3.1 Csharp

Of course we could write programs to demonstrate arithmetic, but there is a fair amount of overhead with a full program. For just testing little bits, there is another alternative: The Mono system comes with a program *csharp*. Let us try it out.

Open a terminal (Linux or OS X) or *Mono Command Prompt* window in Windows, and enter the command `csharp`. You should see :

```
Mono C# Shell, type "help;" for help

Enter statements below.
csharp>
```

The `csharp>` prompt tells you that the C# interpreter has started and is awaiting input. This allows you to create small bits of C# and test them, interactively, without having to write a full program!

Play along with the examples here, entering what comes after the prompt:

```
csharp> 2 + 3
5
```

The csharp program just has a *read, evaluate, and print loop*: the acronym is *repl*. It evaluated the expression `2 + 3` and printed the result, on a line without a prompt. Csharp can evaluate arbitrary C# expressions. It is very handy for testing as you get used to new syntax.

Subtraction works as you would expect. Blanks are optional around symbols:

```
csharp> 10 - 3
7
```

For the binary arithmetic operators, you are encouraged to add blanks to make the expression more easily readable by humans.

The csharp program is more line-oriented than the C# language. If you press the return key when what you have entered is a complete expression you see the value as a response. If your expression is clearly incomplete you get another > prompt (with no "csharp"), until you have entered enough for a full expression.

```
csharp> 10-
      > 3
7
```

In math class you could enter something like 4(10) for multiplication:

```
csharp> 4(10)
{interactive}(1,2): error CS0119: Expression denotes a 'value',
where a 'method group' was expected
```

Unfortunately the error messages are not always easy to follow: it is hard to guess the intention of the user making a mistake.

The issue here is that the multiplication operator must be *explicit* in C#. Recall that an asterisk is used as a multiplication operator:

```
csharp> 4 * 10
40
```

Enter each of the following expressions into csharp, and think what they will produce (and then check):

```
2*5
2 + 3 * 4
```

If you expected the last answer to be 20, think again: C# uses the normal *precedence* of arithmetic operations: Multiplications divisions, and negations are done before addition and subtraction, unless there are parentheses forcing the order:

```
csharp> -(2+3)*4
-20
```

A sequence of operations with equal precedence also work like in math: left to right in most cases, like for combinations of addition and subtraction:

```
csharp> 10 - 3 + 2
9
```

2.3.2 Division and Remainders

We started with the almost direct translations from math. Division is more complicated. We continue in the csharp program:

```
csharp> 5.0/2.0;
2.5
csharp> 14.0/4.0;
3.5
```

So far so good. Now consider:

```
csharp> 14/4
3
```

What? Some explanation is in order. All data has a *type* in C#. When you write an explicit number without a decimal point, like 2, 17, or -237, it is interpreted as the type of an integer, called `int` for short.

When you include a decimal point, the type is `double`, representing a more general real number. This is true even if the value of the number is an integer like 5.0: the type is still `double`.

Addition, subtraction, and multiplication work as you would expect for `double` values, too:

```
csharp> 0.5 * (2.0 + 4.5)
3.25
```

If one or both of the operands to `/` is a `double`, the result is a `double`, close to the actual quotient. We say close, because C# stores values with only a limited precision, so in fact results are only approximate in general. For example:

```
csharp> 1.0/3
0.333333333333333
```

Small errors are also possible with the `double` type and the other arithmetic operations. See *Type double*.

Note: In C#, the result of the / operator depends on the *type* of the operands, not on the *mathematical value* of the operands.

Division with `int` data is handled completely differently.

If you think about it, you learned several ways to do division. Eventually you learned how to do division resulting in a decimal. In the earliest grades, however, you would say

"14 divided by 4 is 3 with a remainder of 2."

Note the quotient is an integer 3, that matches the C# evaluation of 14/4, so having a way to generate an integer quotient is not actually too strange. The problem here is that the answer from grade school is in *two* parts, the integer quotient 3 and the remainder 2.

C# has a *separate* operation symbol to generate the remainder part. There is no standard single operator character operator for this in regular math, so C# grabs an unused symbol: `%` is the remainder operator. (This is the same as in many other computer languages.)

Try in the csharp shell:

```
csharp> 14 / 4
3
csharp> 14 % 4
2
```

You see that with the combination of the `/` operator and the `%` operator, you get both the quotient and the remainder from our grade school division.

Now predict and then try each of these expression in csharp:

```
23/5
23%5
20%5
6/8
6%8
6.0/8
```

Finding remainders will prove more useful than you might think in the future! Remember the strange `%` operator.

Note: The precedence of `%` is the same as `/` and `*`, and hence higher than addition and subtraction, `+` and `−`.

When you are *done with csharp*, you can enter the special expression

```
quit
```

There are some more details about numeric types in *Value Types and Conversions*.

We have been testing arithmetic expressions, with the word *expression* used pretty much like with normal math. More generally in C# an *expression* is any syntax that evaluates to a single value of some type. We will introduce many more types and operations that can be used in expressions.

Divisible by 17 Exercise

What is a simple expression that lets you see if an int x is divisible by 17?

Mixed Arithmetic Exercise

Think of the result of one of these at a time; write your prediction, and *then* test, and write the correct answer afterward if you were wrong. Then go on to the next.... For the ones you got wrong, can you explain the result after seeing it?

```
2 * 5 + 3
2 + 5 * 3
1.5 * 3
7.0/2.0
7.0/2
7/2.0
4.0 * 3 / 8
4 * 3 / 8
6 * (2.0/3)
6 * (2/3)
3 + 10 % 6
10 % 6 + 3
```

2.4 Variables and Assignment

Each piece of data in a C# program has a *type*. Several types have been introduced: int for integers, double for numbers allowing a fractional part, approximating more general real numbers. There are many other numeric types and also non-numeric types, but we can use int and double for examples now. Data gets stored in computer memory. Specific locations in memory are associated with the type of data stored there and with a name to refer to it.

A program allocates a named storage spot for a particular type of data with a *declaration statement*, like:

```
int width;
```

Each declaration must specify a type for the data to be stored and give a name to refer to it. These names associated with a data storage location are called *variables*.

The declaration statement above sets aside a location to store an `int`, and names the location `width`. Several variables of the same type can be listed together, like:

```
double x, y, z;
```

identifying three storage locations for variables of type `double`.

To be useful, data needs to be stored in these locations. This is done with an *assignment statement*. For example:

```
width = 5;
```

A simple schematic diagram with a name for a location in memory (the box):

Although we are used to reading left to right, an assignment statement works *right to left*. The value on the right side of the equal sign is calculated and then placed in the memory location associated with the variable on the left side of the equal sign, either giving an initial value or *overwriting* any previous value stored there.

Variables can also be initialized as they are declared:

```
int width = 5;
double x = 12.5, y = 27, z = 0.5;
```

or initializations and plain declarations can be mixed:

```
int width = 5, height, area;
height = 7;
```

Stylistically the example above is inconsistent, but it illustrates what is possible. Technically an initialization is not an assignment. We will see some syntax that is legal in initializers, but not in assignment statements.

We could continue with a further assignment statement:

```
area = width * height;
```

Look at this in detail. The assignment statement starts by evaluating the expression on the right-hand side: `width * height`. When variables are used in an expression, their current values are substituted, like in evaluating an expression in math, so the value is the same as

```
5 * 7
```

which finally evaluates to 35. In the last step of the assignment statement, the value 35 is then assigned to the variable on the left, `area`.

> **Warning:** You want *one* spot in memory prepared for *each* variable. This happens with declaration, not assignment: Assignment just changes the value at the current location. Do not *declare* the same variable more than once. You will get an error. More on the fine points around that in *Local Scope*.

We continue introducing *Csharp*: Remember that in csharp you can just give an expression, and csharp responds with a value. That syntax and reaction is special to csharp. In csharp you can also test regular C# statements, like declarations and assignments. The most recent versions of csharp do not require you to end a statement with a semicolon, though we tend to put semicolons after statements in our illustrations (and no semicolon for just an expression). As in a regular program, statements do not give an immediate visible response in csharp. Still in csharp you can display a variable value easily:

```
csharp> int width = 5, height, area;
csharp> height = 7;
csharp> area = width * height;
csharp> area
35
```

In the last line, `area` is an expression, and csharp will give back its value, which is just the current value of the variable.

At this point you should be able to make sense of some more features of csharp. You can start with the csharp special help command:

```
csharp> help
"Static methods:
  Describe (object)       - Describes the object's type
  LoadPackage (package);  - Loads the given Package (like -pkg:FILE)
  LoadAssembly (assembly) - Loads the given assembly (like -r:ASSEMBLY)
  ShowVars ();            - Shows defined local variables.
  ShowUsing ();           - Show active using declarations.
  Prompt                  - The prompt used by the C# shell
  ContinuationPrompt      - The prompt for partial input
  Time(() -> { })         - Times the specified code
  print (obj)             - Shorthand for Console.WriteLine
  quit;                   - You'll never believe it - this quits the repl!
  help;                   - This help text
  TabAtStartCompletes - Whether tab will complete even on empty lines
```

A lot of this is still beyond us but these parts are useful:

```
ShowVars ();               - Shows defined local variables.
quit;                      - You'll never believe it - this quits the repl!
help;                      - This help text
```

We can continue the csharp session above and illustrate ShowVars():

```
csharp> ShowVars();
int width = 5
int height = 7
int area = 35
```

displaying all the variables currently known to csharp, plus their current values.

We refer to "current values". An important distinction between variables in math and variables in C# is that C# values can *change*. Follow this csharp sequence:

```
csharp> int n = 3;
csharp> n
3
csharp> n = 7;
csharp> n
7
```

showing we can change the value of a variable. The most *recent* assignment is remembered (until the next assignment....) We can imagine a schematic diagram:

2.4. Variables and Assignment **43**

n 37

We can carry this csharp session one step further, illustrating a difference between C# and math:

```
csharp> n = n + 1;
csharp> n
8
```

Clearly `n = n + 1` is not a true mathematical equation: It *is* a C# assignment, executing with a specific sequence of steps.

1. First the right hand side expression is evaluated, `n + 1`.

2. This involves looking up the current value of `n`, which we set to 7, so the expression is the same as `7 + 1` which is 8.

3. *After* this evaluation, an assignment is made to the left hand variable, which happens to be `n` again.

4. Then the *new* value of `n` is 8, replacing the old 7.

There are many occasions in which such an operation will be useful.

Assignment syntax does have two strikes against it:

1. It appropriates math's equal sign to mean something quite different.

2. The right to left operation is counter to the English reading direction.

Still this usage is common to many programming languages.

> **Warning:** Remember in an assignment that the sides of the equal sign have totally different meanings. You assign to a variable on the left side *after* evaluating the expression on the right.

We can illustrate a likely mistake in csharp:

```
csharp> 3 = n;
{interactive}(1,2): error CS0131: The left-hand side of an assignment
must be a variable, a property or an indexer
```

Students commonly try to assign left to right. At least in this case you get an error message so you see a mistake. If you mean to assign the value of x to y, and write:

```
x = y;
```

you get the opposite effect, changing x rather than y, with *no* error statement. Be careful!

There is some weirdness in csharp because it adds special syntax for expressions which does not appear in regular programs, but it also wants to allow syntax of regular programs. Some conflict can occur when trying to display an expression, sometimes leading to csharp giving a strange error for apparently no reason. In that case, try putting *parentheses* around the expression, which is always legal for an expression, but would never start a regular statement:

```
csharp> int width = 3;
csharp> int height = 5;
csharp> width * height
{interactive}(1,2): error CS0246: The type or namespace name 'width' could
not be found. Are you missing a using directive or an assembly reference?
csharp> (width * height)
15
```

2.4.1 Literals and Identifiers

Expressions like `27` or `32.5` or `"hello"` are called *literals*, coming from the fact that they *literally* mean exactly what they say. They are distinguished from variables, whose value the compiler *cannot* infer directly from the name alone.

The sequence of characters used to form a variable name (and names for other C# entities later) is called an *identifier*. It identifies a C# variable or other entity.

There are some restrictions on the character sequence that make up an identifier:

- The characters must all be letters, digits, or underscores _, and must start with a letter. In particular, punctuation and blanks are not allowed.

- There are some words that are *keywords* for special use in C#. You may not use these words as your own identifiers. They are easy to recognize in editors like in Xamarin Studio, that know about C# syntax: They are colored differently.

We will only discuss a small fraction of the keywords in this course, but the curious may look at the full list.

C# is case sensitive: The identifiers `last`, `LAST`, and `LaSt` are all different. Be sure to be consistent. The compiler can usually catch these errors, since it is the version used in the *one* declaration that matters.

What is legal is distinct from what is conventional or good practice or recommended. Meaningful names for variables are important for the humans who are looking at programs, understanding them, and revising them. That sometimes means you would like to use a name that is more than one word long, like `price at opening`, but blanks are illegal! One poor option is just leaving out the blanks, like `priceatopening`. Then it may be hard to figure out where words split. Two practical options are

- underscore separated: putting underscores (which are legal) in place of the blanks, like `price_at_opening`.

- using *camel-case*: omitting spaces and using all lowercase, except capitalizing all words after the first, like `priceAtOpening`

Use the choice that fits your taste (or the taste or convention of the people you are working with). We will tend to use camel-case for variable inside programs, while we use underscores in program file names (since different operating systems deal with case differently).

Assignment Exercise

Think what the result would be in csharp:

```
int x = 1;
x = x + 1;
x = x * 3;
x = x * 5;
x
```

Write your prediction. Then test. Can you explain it if you got it wrong?

Another Assignment Exercise

If you had trouble with the last, one try this one, too:

```
int a = 5;
a = a/2;
a = a + 1;
a = a * 2;
a
```

2.5 Syntax Template Typography

When new C# syntax is introduced, the usual approach will be to give both specific examples and general templates. In general templates for C# syntax the typeface indicates the the category of each part:

Typeface	Meaning
Typewriter font	Text to be written *verbatim*
Bold	A place where you can use an arbitrary identifier.
Emphasized	A place where you can use an arbitrary expression (which might be a single variable name).
Normal text	A description of what goes in that position, without giving explicit syntax

An attempt is made with the parts that are not verbatim to be descriptive of the use expected.

As a start we can give some general syntax for declarations and assignment statements:

2.5.1 Declaration Syntax Options

       type variableName ;

or with initialization:

       type variableName = *initialValue* ;

or there can be a list of variables of the same type, for instance a list of three variables:

       type variableName1 , **variableName2** , **variableName3** ;

Some or all of the variables in the list could also have initializers.

Space is allocated for each variable named, according to its type. Where there is an initializer, an initial value is set for the variable.

2.5.2 Assignment Syntax

> **variableName** = *expression* **;**

The *expression* is evaluated before its value is assigned to **variableName**.

2.6 Strings, Part I

Enough with numbers for a while. Strings of characters are another important type in C#.

A string in C# is a sequence of characters. For C# to recognize a literal sequence of characters, like `hello`, as a string, it must be enclosed in quotes `"` to delimit the string: `"hello"`. Special cases are considered later in *String Special Cases*.

2.6.1 String Concatenation

Because everything in C# is typed, C# can give a special meaning to operators depending on the types involved, as we saw with `/`. We can operate on numbers with arithmetic operators, including +. With strings + has a completely different meaning. Look at the example in csharp:

```
csharp> "never" + "ending";
"neverending"
```

The plus operation with strings means *concatenate* the strings: join them together end to end.

C# is even a bit smarter. If you use a + with a string, presumable you are looking to produce a string, so even if the second operand to the + is *not* a string, it is automatically converted to a string representation before concatenating:

```
csharp> int x = 42;
csharp> string result;
csharp> result = "We get " + x;
csharp> result;
"We get 42"
```

You can chain concatenations. We could make a full sentence adding a period:

```
csharp> "We get " + x + ".";
"We get 42."
```

Note to Python programmers: Unfortunately there is no ⋆ multiplication operator for strings in C#.

Four Copies Exercise

In csharp declare and initialize a string variable. Write an expression that evaluates to four copies of the string, so it works no matter what value you gave your string.

Sum String Exercise

In csharp declare and initialize two int's, x and y. Then enter an expression whose value is "x + y is 56", except that 56 is replaced by the sum of x and y, and is not a literal, but calculated from the actual values of variables x and y (which do not need to add up to 56 specifically).

This has a trick to it.

Ints and Strings Added

In csharp enter

```
int x = 2;
int y = 3;
```

Think what the csharp response is to each of these then write one predicted response at a time, *then* test it, and put the right answer beside your answer if you were wrong:

```
x + "??" + y;
x + y + "??";
(x * y + "??");
"??" + x * y;
"??" + x + y;
x + "??" * y;
```

Can you explain the ones you got wrong, after looking at the actual answer? Precedence and operation order is important.

2.7 Writing to the Console

In interactive use of csharp, you can type an expression and immediately see the result of its evaluation. This is fine to test out syntax and maybe do simple calculator calculations. In a regular C# program run from a file like in *A Sample C# Program*, you must explicitly give instructions to print to a console or terminal window. This will be a window like you see when running csharp.

This printing is accomplished through a function with a long name: `Console.WriteLine`. Like with math, you can pass a function a value to work on, by placing it in parentheses after the name of the function. Unlike in high school algebra classes, in C# we have many types of data to supply other than numbers. The simplest way to use the `Console.WriteLine` function is to give it a string. We can demonstrate in csharp. The response is just the line that would be printed in a regular program:

```
csharp> Console.WriteLine("Hello, world!");
Hello, world!
```

What is printed to the screen does *not* have the quotes which we needed to define the literal string `"Hello, world!"` inside the program.

`Console` is a C# class maintained by the system, that interacts with the terminal or console window where text output appears for the program. A function defined in that class is `WriteLine`. To refer to a function like `WriteLine` in a different class, you must indicate the location of the function with the "dot" notation shown: class name, then `.`, then the function. This gives the more elaborate name needed in the program.

The string that gets printed can be the result of evaluating an expression, for instance concatenating:

```
csharp> int total = 5;
csharp> Console.WriteLine("All together: " + total);
All together: 5
```

More elaborate use of `WriteLine` is discussed in *String Format Operation*.

The `Console.WriteLine` function automatically makes the printing position advance to the next line, as when you press the Enter or Return key. A variant, `Console.Write`, prints the parameter exactly, and nothing else. The statement-at-a-time approach in csharp is not good for illustrating the differences.

Printing is better shown off in a real program....

2.8 C# Program Structure

We discuss the most basic syntax satisfied by all C# programs, which are plain text files, with names ending in `.cs`. There will be additions later, but any program you can run will include:

```
using System;

class ClassName
{
    static void Main()
    {
        program statements go here....


    }
}
```

By convention class names are capitalized.

You can see that both the example program painting/painting.cs and the lab program hello/hello.cs follow this pattern. The discussion of these parts through line 6 in *A Sample C# Program* are about all we have to say at this point. For now this

is the boilerplate code. We will make additions as necessary. We choose not to clutter up the basic setup with features that we are not about to use and discuss.

Here is a silly little test illustrating the difference between `Console.WriteLine` and `Console.Write`, in example write_test/write_test.cs:

```csharp
using System;

class WriteTest
{
   static void Main()
   {
      string s = "hello";
      Console.Write(s);
      Console.Write("there");
      Console.WriteLine(s);
      Console.WriteLine( "Another line");
      Console.Write("Starting ");
      Console.WriteLine("yet another line");
   }
}
```

When run, the program prints:

```
hellotherehello
Another line
Starting yet another line
```

Do you see how the output shows the differences between `WriteLine` and `Write`? If we added another printing statement, where would the beginning of the output appear: after the final `e` or under the `S` of Starting?

2.8.1 Indentation Help

Using conventional indentation helps understand a program and find errors, like unmatched braces. When you press return in a C# source file (i.e. a file with name ending in `.cs`), Xamarin Studio makes a guess at the proper indentation of the next line. The exact reaction can be set in the *options*. The simplest approach is to set these options *once* for all new files in a solution, like your work solution:

1. Access the context menu for the whole solution (not one project).

2. Select Options.

3. In the popup Solution Options Window, you will see "Code Formatting" in the left column. If you do not see "C# source code" and then "Text file" right underneath it click on "Code Formatting" to expand the hierarchy.

4. Click on Text file, and then the right side should show options. Adjust them to look like the picture: tab and indent widths 3, first and last check boxes *checked* (Convert tabs to spaces. Remove trailing spaces), and have the middle check box (Allow tabs after non-tabs) *unchecked*.

5. Click OK.

Tabs vs. spaces is not a significant issue inside a consistent environment, like Xamarin Studio, but if there are tab characters in a file, they can be expanded different amounts by different display programs. Then particularly if you mix tab characters and spaces you can get something very strange. If only spaces are used, there is no ambiguity in different displays.

The 3 vs. 4 spaces is not a big deal, but 3 appears to be large enough to see easily, and makes lines with nested indentation have more room.

2.8.2 Compiler Error Help

There are an enormous number of possible *syntactic* errors in your source code that the compiler can detect.

Errors shown *while* **editing**: If you use an editor like Xamarin Studio, some of these errors are even checked while you type and are noted while you are editing. They may be indicated with a red squiggly underline, and possibly a red comment at the right. Sometimes the squiggle is just because you are in the middle of something, but if it is still there after you complete a whole statement, there may well be an issue to look at. Sometimes fixing the issue makes the annotation go away, but sometimes your program may really be fixed, even though the error annotation does not go away *until you formally compile/build the whole program*. This annoying ambiguity leads some people to turn off error annotations, and just let the system note errors after a complete compile cycle.

After compiling and getting an error, sometimes reading the error description carefully will help you understand the problem. Sometimes the error is very cryptic. In those cases it might help to look at the C# .NET error documentation,

> http://msdn.microsoft.com/en-us/library/ms228296.aspx.

Each compiler error you make has a number shown in its description. Many of these error numbers are shown in the left column of the linked page. You can click to get a more complete explanation and examples.

Another important way to learn about the error messages is to leverage your experience: After you have eventually found how to fix your error, *allow the extra time* to use your new knowledge, look *back* at the original error message, and see if the error description text makes more sense now. That should help next time, (and there usually is a next time). Even when the error description still makes no particular sense, you may well get into the same situation again, with the same error number. Then remembering the issue you found in a previous time could help.

Debugging can eat up an enormous amount of time, so it is really worth your effort to understand the errors that you tend to make and the errors' relation to the error messages that you get.

> **Warning:** It is certainly helpful that the compiler finds *syntactic* errors for you, but be sure to remember that compiling does *not* mean the program will "work" and correctly do what you desire. Test your compiled program thoroughly to reduce the chance of *logical* errors remaining, that cause *run-time* errors, or just the wrong results.

2.8.3 Sequential Execution

A function like `Main` involves a sequence of statements to execute. The basic order of execution is *sequential*: first statement, then second, ... This is the same as the *textual order*, so it is easy to follow. Later in *Defining Functions of your Own*, *Decisions*, and *While Loops*, we will see more complicated execution orders that do *not* match textual order. Whatever the order of execution given by the program, it is important to always keep track of the current state of the program: where we are and what are the values of variables.

For now consider a small, artificial example program, update_vars/update_vars.cs, emphasizing the ability to *reassign* variable values.

```csharp
using System;

class UpdateVars
{
   static void Main()
   { // simple sequential code updating two variables
      int x = 3;
      int y = x + 2;
      y = 2 * y;
      x = y - x;
      Console.WriteLine (x + " " + y);
   }
}
```

Can you *predict* the result? Run the program and check. Particularly if you did not guess right, it is important to understand what happens, one step at a time. That means keeping track of what changes to variables are made by each statement.

In the table below, statements are referred to by the numbers labeling the lines in the code above. We can track the state of each variable after each line is executed.

A dash is shown where a variable is not defined. For instance after line 7 is executed, a value is given to x, but y is still undefined. Then y gets a value in line 8. The comment space can be used any time it is helpful. In particular it should be used when something is printed, since this important action does *not* affect the variable list.

Line	x	y	Comment
5	-	-	Start at beginning of Main
7	3	-	initialize x
8	3	5	5=3+2, using the value of x from the previous line
9	3	10	10=2*5 on the right; use the value of y from the previous line
10	7	10	7=10-3 on the right; use the value of x and y from the previous line
11	7	10	print: 7 10

The critical point here is to always use the most recently assigned value of each variable.

The order of execution will always be the order of the lines in our table. In this simple *sequential* code, that *also* follows the textual order of the program.

Following each line of execution of a program in the proper order of execution, carefully, keeping track of the current values of variables, will be called *playing computer*. A table like the one above is an organized way to keep track.

The line numbering is not very exciting in a simple sequential program, but it will become very important when we get to other execution sequences. We start with the simple sequential numbering now for consistency, as we get used to the idea of such a table following execution sequence.

Play Computer Exercise

Here is a similar program, update_vars2/update_vars2.cs:

```
1   using System;
2
3   class UpdateVars2
4   {
5       static void Main()
6       {   // for first Playing Computer Exercise
```

```
7        int a = 5;
8        int b = 2 * a;
9        a = a * b;
10       b = a - b;
11       a = a + b;
12       Console.WriteLine (a + ":" + b);
13     }
14 }
```

Play computer, completing the table

Line	a	b	Comment
5	-	-	Start at beginning of Main
7	5	-	initialize a
8			
9			
10			
11			
12			

Another Play Computer Exercise

A silly one on the same line: update_vars3/update_vars3.cs:

```
1  using System;
2
3  class UpdateVars3
4  {
5     static void Main()
6     {  // another sequentail Playing Computer Exercise
7        string s = "a";
8        string t = s + "n";
9        s = t + s;
10       t = "b" + t;
11       s = t + s;
12       Console.WriteLine (s + " " + t);
13     }
14 }
```

Play computer, completing the table

Line	s	t	Comment
5	-	-	Start at beginning of Main
7			
8			
9			
10			
11			
12			

2.9 Combining Input and Output

2.9.1 Reading from the Keyboard

If you want users to type something at the keyboard, you should let them know
first! The jargon for this is to give them a *prompt*: Instructions written to the
screen, something like

```
Console.Write("Enter your name: ");
```

Then the user should respond. What the user types is automatically shown
(*echoed*) in the terminal or console window. For a program to read what is typed,
another function in the `Console` class is used: `Console.ReadLine`.

Here the data for the function comes from a line typed at the keyboard
by the user, so there is no need for a parameter between the parentheses:
`Console.ReadLine()`. The resulting sequence of characters, typed before
the user presses the Enter (Return) key, form the string *value* of the function. Syn-
tactically in C#, when a function with a value is used, it is an *expression* in the
program, and the expression evaluation is the value produced by the function. This
is the same as in normal use of functions in math.

With any function producing a value, the value is *lost* unless it is *immediately*
used. Often this is done by immediately assigning the value to a variable like in

```
string name;
name = Console.ReadLine();
```

or in the shorter

```
string name = Console.ReadLine();
```

Fine point: Notice that in most operating systems you can edit and correct your line before pressing the Return key. This is handy, but it means that the Return key *must* always be pressed to signal the end of the response. Hence a whole line must be read, and there is *no* function `Console.Read()`. Just for completeness we mention that you can read a raw single keystroke immediately (no editing beforehand). If you want to explore that later, see test_readkey/test_readkey.cs.

2.9.2 Numbers and Strings of Digits

You may well want to have the user supply you with numbers. There is a complication. Suppose you want to get numbers and add them. What happens with this code, in bad_sum/bad_sum.cs?

```
using System;

class BadSum
{
   static void Main()
   {
      string s, t, sum;
      Console.Write ( "Enter an integer: ");
      s = Console.ReadLine();
      Console.Write( "Enter another integer: ");
      t = Console.ReadLine();
      sum = s + t;
      Console.WriteLine("They add up to " + sum);
   }
}
```

Here is a sample run:

```
Enter an integer: 23
Enter another integer: 9
They add up to 239
```

C# has a type for everything and `Console.ReadLine()` gives you a string. Adding strings with + is not the same as adding numbers!

We must explicitly convert the strings to the proper kind of number. There are

functions to do that: `int.Parse` takes a string parameter that should be the characters in an `int`, like "123" or "-25", and produces the corresponding `int` value, like 123 or -25. In good_sum/good_sum.cs, we changed the names to emphasize the type conversions:

```csharp
using System;

class GoodSum
{
   static void Main()
   {
      Console.Write ( "Enter an integer: ");
      string xString = Console.ReadLine();
      int x = int.Parse(xString);
      Console.Write( "Enter another integer: ");
      string yString = Console.ReadLine();
      int y = int.Parse(yString);
      int sum = x + y;
      Console.WriteLine("They add up to " + sum);
   }
}
```

Notice that the values calculated by `int.Parse` for the strings `xString` and `yString` are immediately remembered in assignment statements. Be careful of the distinction here. The `int.Parse` function does not magically *change* the variable `xString` into an `int`: the string `xString` is unchanged, but the corresponding `int` value is calculated, and gets assigned to an `int` variable, `x`.

Note that this would not work if the string represents the wrong kind of number, but there is an alternative:

```
csharp> string s = "34.5";
csharp> int.Parse(s);
System.FormatException: Input string was not in the correct format ....
csharp> double.Parse(s);
34.5
```

We omitted the long tail of the error message. There is no decimal point in an `int`. You see the fix with the corresponding function that returns a double.

2.9.3 Example Projects and the Source Repository

We have started to refer to whole programs that we have written. You will want to have your own copies to test and modify for related work.

All of our examples are set up in a Xamarin Studio solution in our zip file that you can download.

The zip file and the folder it unzips to have the long name introcs-csharp-examples-master. We suggest you *rename the folder* simply `examples` to match the name of the Xamarin Studio solution it contains.

There are various way to access our files.

1. One way is to look at individual files from your download under our examples directory.

2. If you open the examples solution in Xamarin Studio, you can select files from the Solutions pad. (Instructions are in the next section.)

3. In the notes we refer to individual code file names that are hyperlinked. They link to the *latest version* in our online source repository. You get a display of color-coded web page with numbered lines. If you want to adapt a chunk, you can select it, and copy. If you want to copy all of a large file, your editing shortcuts for Select All do *not* work: You get a bunch of extra html. An alternative is to click the **Raw** button in the web page to the left above the source code. That brings up a plain text page with just the code. You can either download it or select all of it, and you *only* get the code.

We have one main convention in naming our projects: Most projects are examples of full, functional programs to run. Others are intended to be copied by you as *stubs* for your solutions, for you to elaborate. These project folders all end with "_stub", like `string_manip_stub`. Even the stubs can be compiled immediately, though they may not accomplish anything.

A further convention is using "chunk" comments inside example source files: To keep the book and the source code in sync, our Sphinx building routine directly uses excerpts from the exact source code that is in the examples download. We have to mark the limits of the excerpts that we want for the book. Our convention is to have a comment referring to the beginning or the end of an excerpt "chunk". Hence a comment including "chunk" in a source file is *not* intended as commentary on the code, but merely a marker for automatically regenerating a revision of the book.

2.9.4 Running our Xamarin Studio Examples Solution

If you are just starting Xamarin Studio, and you have *not* run our solution before:

1. On the Welcome screen select the button Open Solution or File.

2. You get an open-file dialog. Navigate to our example solution. (It must be unzipped already! We assume you renamed the folder *examples*.)

3. Select `examples/examples.sln`.

The next time you come to the Welcome screen, our examples should be listed in the Recent Projects, and you can click to open it directly.

2.9.5 Copying and Modifying Our Example Xamarin Studio Projects

We strongly encourage you *not* to modify our examples in place, if you want to keep the changes, because we will make additions and modifications to our source download, and we would not want you to overwrite any of your modified files after downloading a later version of the examples.

If you do want to alter our code, we suggest you copy it to a project in your solution ("work", discussed in the first lab in the *Steps*).

1. Open your solution.

2. Create a new project, maybe with the same name as the one we had. If it was a "_stub" project, remove the "_stub" from your project's name.

3. In the Solution Pad open the menu on the new project, select, Add, and then in the further submenu, select Add Files....

4. This brings up an operating system open-file dialog. Switch folders into our example projects. Select the files you want to copy. (It makes things easier if you put the examples folder right beside your work folder.)

5. A further dialog window pops up, with the choice **Copy** selected. Click to approve copy (as opposed to move or link).

6. Now the desired files should appear in your project, along with the unfortunate default Program.cs. If you have not already deleted Program.cs, as described in *Steps*, do it now.

7. If you intended to copy everything for a project, test by running the project. Even our stub projects should compile, though a stub project may not do anything when you run it until you add your own code to it. To make successful incremental additions, it is always good to start from something that compiles!

When creating modifications of previous examples, like the exercise below, you can often save time by copying in the related example, particularly avoiding retyping the standard boiler plate code at the top. However, when you are first learning and getting used to new syntax, typing reinforces learning. Perhaps after looking over the related example, you are encouraged to write your version from scratch, to get used to all the parts of the code. Later, when you can produce such text automatically, feel free to switch to just copying from a place that you had it before.

Interview Exercise/Example

Write a program that prompts the user for a name (like Elliot) and a time (like 10AM) and prints out a sentence like:

```
Elliot has an interview at 10AM.
```

If you are having a hard time and want a further example, or want to compare to your work, see our solution, interview/interview.cs.

Exercise for Addition

Write a version, `add3.cs`, that prompts the user for three numbers, *not necessarily integers*, and lists all three, and their sum, in similar format to good_sum/good_sum.cs.

Exercise for Quotients

Write a program, `quotient.cs`, that prompts the user for two integers, and then prints them out in a sentence with an integer division problem like

```
The quotient of 14 and 3 is 4 with a remainder of 2.
```

Review *Division and Remainders* if you forget the integer division or remainder operator.

2.10 String Special Cases

There are some special cases for creating literal strings. For instance you might want quotes as characters inside your string. In this case you need special symbolism using a character *escape code*, starting with \ backslash. Then the character after the backslash has a special meaning.

For instance a quote character after a backslash, \", does not mean the end of a string literal. It means a quote character is literally used *in* the string: `"He said, \"Hello!\", over and over."`

We can illustrate with csharp, first with a simple string:

```
csharp> Console.WriteLine("Hello world!");
Hello world!
csharp> Console.WriteLine("He said, \"Hello!\", over and over.");
He said, "Hello!", over and over.
```

There are many other special cases of escape code. The main ones you are likely to use are:

Escape code	Meaning
\"	" (quote)
\'	' (single quote in char literal)
\\	\ (backslash)
\n	newline

Hence if you really want a backslash character in a literal, you need to write two of them.

The newline character indicates further text will appear on the next line down when *printed* with the `Console.WriteLine` function.

Example:

```
csharp> Console.WriteLine("Windows path: c:\\Users\\aharrin");
Windows path: c:\Users\aharrin
csharp> Console.WriteLine("a\nbc\n\ndef")
a
bc

def
```

Literal strings that are simply delimited by quotes " must start and end on the same line. There is also a notation for *@-quoting*, with an at-sign @ before the first quote. In an @-quoted string, all characters are treated verbatim, including all backslashes. Also the string may go on for several lines, and all newlines are included literally. (The csharp program does not recognize multi-line @-quoted strings.) This fragment in a program would produce the same output as the statements in the csharp example above:

```
        Console.WriteLine(@"Windows path: c:\Users\aharrin");
        Console.WriteLine(@"a
bc

def");
```

The only thing this example does not show off well is the amount of left margin indentation. That is significant in a multiline @-quoted string. A whole simple program with this code is in example at_sign_strings/at_sign_strings.cs.

Caution: When you give csharp an expression evaluating to a string at the prompt, you get back a verbatim string with *quotes added around it*, but no @ to remind you that it is verbatim:

```
csharp> "Windows path: c:\\Users\\aharrin"
"Windows path: c:\Users\aharrin"
csharp> "a\nbc\n\ndef"
"a
bc

def"
```

2.10.1 Multiline String Exercise

1. Write a statement that initializes a string s with a **single** string literal that, when printed, shows something on one line then three empty lines, and then a final line with text.

2. Declare the same string with a different string literal expression, that produces the same string. (Just one of your literals should start with @.)

2.11 Substitutions in Console.WriteLine

2.11.1 Output With +

An elaboration of a "Hello, World" program, could greet the user, after obtaining the user's name. If the user enters the name Elliot, the program could print

Hello, Elliot!

This is a very simple input-process-output program (in fact with almost no "process"). Think how would you code it.

You need to obtain a name, remember it and use it in your output. A solution is in the next section.

2.11.2 String Format Operation

A common convention is fill-in-the blanks. For instance,

Hello, _____!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. C# has a similar construction, better called fill-in-the-braces, that can be used with Console.WriteLine.

Instead of inserting user input with the + operation as in hello_you1/hello_you1.cs:

```csharp
using System;

class HelloYou1
{
    static void Main ()
    {
        Console.WriteLine ("What is your name?");
        string name = Console.ReadLine ();
        Console.WriteLine ("Hello, " + name + "!");
    }

}
```

look at a variation, hello_you2/hello_you2.cs, shown below. Both programs look exactly the same to the user:

```csharp
using System;

class HelloYou
{
   static void Main ()
   {
      Console.WriteLine ("What is your name?");
      string name = Console.ReadLine ();
      Console.WriteLine ("Hello, {0}!", name);
   }

}
```

All the new syntax is in the line:

```csharp
Console.WriteLine ("Hello, {0}!", name);
```

`Console.WriteLine` actually can take parameters *after* an initial string, but only when the string is in the form of a *format string*, with expression(s) in braces where substitutions are to be made, (like in fill-in-the-blanks). Here the format string is `"Hello, {0}!"`.

The remaining parameters, after the initial string, give the values to be substituted. To know *which* further parameter to substitute, the parameters after the initial string are implicitly numbered, *starting from 0*. Starting with 0 is consistent with other numbering sequences in C#. So here, where there is just one value to substitute (`name`), it gets the index 0, and where it is substituted, the braces get 0 inside, to indicate that parameter with index 0 is to be substituted.

Everything in the initial string that is *outside* the braces is just *repeated verbatim*. In particular, if the only parameter is a string with no braces, it is printed completely verbatim (reducing to the situations where we have used `Console.WriteLine` before).

A more elaborate silly examples that you could test in csharp would be:

```csharp
string first = "Peter";
string last = "Piper";
string what = "pick";
Console.WriteLine("{0} {1}, {0} {1}, {2}.", first, last, what);
```

It would print:

```
Peter Piper, Peter Piper, pick.
```

where parameter 0 is `first` (value `"Peter"`), parameter 1 is `last` (value `"Piper"`), and parameter 2 is `what` (value `"pick"`).

Make sure you see why the given output is exactly what is printed.

Or try in csharp:

```
int x = 7;
int y = 5;
Console.WriteLine("{0} plus {1} is {2}; {0} times {1} is {3}.", x, y, x+y, x*y);
```

and see it print:

```
7 plus 5 is 12; 7 times 5 is 35.
```

Note the following features of the parameters after the first string:

- These parameters can be any expression, and the expressions get evaluated before printing.

- These parameters to be substituted can be of any type.

- These parameters are automatically converted to a string form, just as in the use of the string + operation.

In fact the simple use of format strings shown so far can be completed replaced by long expressions with +, if that is your taste. We later discusses fancier formatting in *Tables*, that *cannot* be duplicated with a simple string + operation. We will use the simple numbered substitutions for now just to get used to the idea of substitution.

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final *formatted* string. The rule is to double the braces: `"{{"` and `"}}"`. The fragment

```
int a = 2, b = 3;
Console.WriteLine("The set is {{{0}, {1}}}.", a, b);
```

produces

```
The set is {2, 3}.
```

Format Reading Exercise

What is printed?

```
Console.WriteLine("{0}{1}{1}{2}", "Mi", "ssi", "ppi");
```

Check yourself.

Exercise for Format

Write a program, `quotient_format.cs`, that behaves like *Exercise for Quotients*, but generate the sentence using `Console.WriteLine` with a format string and no + operator.

Madlib Exercise

Write a program, `my_mad_lib.cs`, that prompts the user for words that fit specified gramatical patterns (a noun, a verb, a color, a city....) and plug them into a multiline format string so they fit grammatically, and print the usually silly result. If you are not used to mad libs, try running (not looking at the source code) the example project mad_lib, and then try it again with different data. If this exercise seems like too big of a challenge yet, see our example source code, mad_lib/mad_lib.cs, and then *start over* on your own.

2.11.3 Overloading

The `WriteLine` function can take parameters in different ways:

- It can take a single parameter of an type (and print its string representation).

- It can take a string parameter followed by any number of parameters used to substitute into the initial format string.

- It can take no parameters, and just advance to the next line (not used yet in this book).

Though each of these uses has the same name, `Console.WriteLine`, they are technically all different functions: A function is not just recognized by its name, but by its *signature*, which includes the name **and** the number and types of parameters. The technical term for using the same name with different signatures for different functions is *overloading* the function (or method).

This only makes practical sense for a group of closely related functions, where the use of the same name is more helpful than confusing.

2.12 Value Types and Conversions

2.12.1 Type int

A variable is associated with a space in memory. This space has a fixed size associated with the type of data. The `int` and `double` types are examples of *value types*, which means that this memory space holds an encoding of the complete data for the value of the variable. The fixed space means that an `int` cannot be a totally arbitrary integer of an enormous size. In fact an `int` variable can only hold an integer in a specific range. See *Data Representation* for the general format of the underlying encoding in bits.

An `int` is held in a memory space of 32 bits, so it can have at most 2^{32} values, and the encoding is chosen so about half are positive and half are negative: An `int` has maximum value $2^{31} - 1 = 2147483647$ and a minimum value of $-2^{31} = -2147483648$. The extreme values are also named constants in C#, `int.MaxValue` and `int.MinValue`.

In particular this means `int` arithmetic does not always work. What is worse, it fails *silently*:

```
csharp> int i = int.MaxValue;
csharp> i;
2147483647
csharp> i + 5;
-2147483644
```

Add two positive numbers and get a negative number! Getting such a wrong answer is called *overflow*. Be very careful if you are going to be using big numbers!

Note: with addition, overflow will give the wrong sign, but the sign may not give such a clue if another operation overflows, like multiplication.

2.12.2 Type long

Most everyday uses of integers fit in this range of an `int`, and many modern computers are designed to operate on an `int` very efficiently, but sometimes you need a larger range. Type *long* uses twice as much space.

The same kind of silent overflow errors happen with `long` arithmetic, but only with much larger numbers.

When we get to *Arrays*, you will see that a program may store an enormous number of integers, and then the total space may be an issue. If some numbers fit in a `long`, but not an `int`, `long` must be used, taking us twice the space of an array of `int` elements. If all the integers have even more limited ranges, they might be stored in the smaller space of a `short` or a `byte`. We will not further discuss or use types `short` or `byte` in this book. Here we will only use the integral types `int` and `long`.

2.12.3 Type double

A `double` is also a value type, stored in a fixed sized space. There are even more issues with `double` storage than with an `int`: Not only do you need to worry about the total magnitude of the number, you also need to choose a *precision*: There are an infinite number of real values, just between 0 and 1. Clearly we cannot encode for all of them! As a result a `double` has a limited number of digits of accuracy. There is also an older type `float` that takes up half of the space of a `double`, and has a smaller range and less accuracy. This at least gives a reason for the name `double`: double the storage space of a `float`.

To avoid a ridiculously large number of trailing 0's, a big double literal can be expressed using a variant of scientific notation:

$$1.79769313486232\text{E}+308 \text{ means } 1.7976931348623157(10^{308})$$

C# does not have the typography for raised exponents. Instead literal values can use the E to mean "times 10 to the power", and the E is followed by and exponent integer that can be positive or negative. The whole double literal may not contain

any embedded blanks. Internally these numbers are stored with powers of 2, not 10: See *Data Representation*.

Arithmetic with the `double` type does not overflow silently as with the integral types. We show behavior that could be important if you do scientific computing with enormous numbers: There are values for Infinity and Not a Number, abbreviated NaN. See them used in csharp:

```
csharp> double x = double.MaxValue;
csharp> x;
1.79769313486232E+308
csharp> double y = 10 * x;
csharp> y;
Infinity
csharp> y  + 1000;
Infinity
csharp> y  - 1000;
Infinity
csharp> 1000/y;
0
csharp> double z = 10 - y;
csharp> z;
-Infinity
csharp> double sum = y + z;
csharp> sum;
NaN
csharp> sum/1000;
NaN
```

Once a result gets too big, it gets listed as infinity. As you can see, there is some arithmetic allowed with a finite number and infinity! Still some operations are not legal. Once a result turns into `NaN`, no arithmetic operations change further results away from `NaN`, so there is a lasting record of a big error!

Note that Infinity, -Infinity and NaN are just representations when displayed as strings. The numerical constants are `Double.PositiveInfinity`, `Double.NegativeInfinity`, and `Double.NaN`.

> **Warning:** There is no such neat system for showing off small inaccuracies in `double` arithmetic accumulating due to limited precision. These inaccuracies *still* happen silently.

2.12.4 Numeric Types and Limits

The listing below shows how the storage size in bits translates into the limits for various numerical types. We will not discuss or use `short`, `byte` or `float` further.

long 64 bits; range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

int 32 bits; range -2,147,483,648 to 2,147,483,647

short 16 bits; range -32,768 to 32,767

byte 8 bits; range 0 to 255 (no negative values)

double 64 bits; maximum magnitude: $1.7976931348623157 (10^{308})$; about 15 digits of accuracy

float 32 bits; maximum magnitude: $3.402823 (10^{38})$; about 7 digits of accuracy

decimal 128 bits; maximum value: 79228162514264337593543950335; 28 digits of accuracy; can exactly represents decimal values for financial operations; briefly discussed in *optional Decimal Type*.

char See *char as integer*.

2.12.5 Casting

While the mathematical ideas of 42 and 42.0 are the same, C# has specific types. There are various places where numerical types get converted automatically by C# or explicitly by the programmer. A major issue is whether the new type can accurately represent the original value.

Going from `int` to `double` has no issue: Any `int` can be exactly represented as a `double`. Code like the following is fine:

```
csharp> int i = 33;
csharp> double d = i;
csharp> double x;
csharp> x = 11;
csharp> double z = i + 2.5;
csharp> ShowVars();
int i = 33
double d = 33
```

```
double x = 11
double z = 35.5
```

The `double` variable `d` is initialized with the value of an `int` variable. The `double` variable `x` is assigned a value using an `int` literal. The `double` variable `z` is initialized with the value of a sum involving both an `int` variable and a `double` literal. As we have discussed before in *Arithmetic*, the `int` is converted to a `double` before the addition operation is done.

The other direction for conversion is more problematic:

```
csharp> double d= 2.7;
csharp> int i;
csharp> i = d;
{interactive}(1,4): error CS0266: Cannot implicitly convert type
    'double' to 'int'.
An explicit conversion exists (are you missing a cast?)
```

The `int i` cannot accurately hold the value 2.7. Since the compiler does this checking, looking only at types, not values, this even fails if the the `double` happens to have an integer value:

```
csharp> double d = 2.0;
csharp> int i = d;
{interactive}(1,4): error CS0266: Cannot implicitly convert type
    'double' to 'int'.
An explicit conversion exists (are you missing a cast?)
```

If you really want to possibly lose precision and convert a `double` to an `int` result, you *can* do it, but you must be explicit, using a *cast* as the csharp error messages suggest.

```
csharp> double d= 2.7;
csharp> int i;
csharp> i = (int)d;
csharp> i;
2
```

The desired result type name in parentheses `(int)` is a *cast*, telling the compiler you really intend the conversion. Look what is lost! The cast does not *round* to the nearest integer, it *truncates* toward 0, dropping the fractional part, .7 here.

Rounding is possible, but if you really want the `int` type, it takes two steps,

---

because the function `Math.Round` does round to a mathematical integer, but leaves the type as `double`! To round `d` to an `int` result we could use:

```
csharp> i = (int)Math.Round(d);
csharp> i;
3
```

You can also use an explicit cast from int to double. This is generally not needed, because of the automatic conversions, but there is one place where it is important: if you want `double` division but have `int` parts. Here is a quick artificial test:

```
csharp> int sum = 14;
csharp> int n = 4;
csharp> double avg = sum/n;
csharp> avg;
3
```

Oops, integer division. Instead, continue with:

```
csharp> avg = (double)sum/n;
csharp> avg;
3.5
```

We get the right decimal answer.

This is a bit more subtle than it may appear: The cast to double, `(double)` is an operation in C# and so it has a *precedence* like all operations. Casting happens to have precedence higher than any arithmetic operation, so the expression is equivalent to:

```
avg = ((double)sum)/n;
```

On the other hand, if we switch the order the other way with parentheses around the division:

```
csharp> avg = (double)(sum/n);
csharp> avg;
3
```

then working *one* step at a time, `(sum/n)` is *integer* division, with result 3. It is the 3 that is then cast to a double (too late)!

See the appendix *Precedence of Operators*, listing all C# operations discussed in this book.

2.12.6 Type char

The type for an individual character is `char`. A `char` literal value is a *single* character enclosed in *single* quotes, like `'a'` or `'$'`. The literal for a single quote character itself and the literal for a newline use *escape codes*, like in *String Special Cases*: The literals are `'\''` and `'\n'` respectively.

Be careful to distinguish a `char` literal like `'A'` from a string literal `"A"`.

Char as integer: Though the `char` type has character literals and prints as a character, internally a `char` is a *type of integer*, stored in 16 bits, with the correspondence between numeric codes and characters given by the *Unicode* standard. Unicode allows special symbol characters and alphabets of many languages. We will stick to the standard American keyboard for these characters.

Besides different alphabets, Unicode also has characters for all sorts of symbols: emoticons, chess pieces, advanced math.... See http://www.unicode.org/charts. All the symbols can be represented as escape codes in C#, starting with `\u` followed by 4 hexadecimal digits. For example `\u262F` produces a yin-yang symbol.

We mention the `char` type being numeric mostly because of errors that you can make that would otherwise be hard to figure out. This code does not concatenate the `char` symbols:

```
csharp> Console.WriteLine('A' + '-');
110
```

What? We mentioned that modern computers are set up to easily work with the `int` type. In arithmetic with *smaller* integral types the operands are first automatically converted to type `int`. An `int` sum is an `int`, and that is what is printed.

You can look at the numeric values inside a `char` with a cast!

```
csharp> (int)'A';
65
csharp> (int)'-';
45
```

So the earlier 110 is correct: 65 + 45 = 110.

For completeness: It is also possible to cast from small `int` back to `char`. This

---

may be useful for dealing with the alphabet in sequence (or simple classical cryptographic codes):

```
csharp> 'A' + 1;
66
csharp> (char)('A' + 1);
'B'
```

The capital letter one place after A is B.

2.12.7 Type Boolean or bool

There is one more very important value type, that we introduce here for completeness, though we will not use it until *Decisions*. Logical conditions are either true or false. The type with just these two values is *Boolean*, or *bool* for short. The type is named after George Boole, who invented what we now call *Boolean algebra*. Though it seemed like a useless mathematical curiosity when Boole invented it, a century later Boolean algebra turned out to be at the heart of the implementation of computer hardware.

Note: The Boolean literals are `true` and `false`, with *no* quotes around them.

With quotes they would be of type string, not Boolean!

2.12.8 Overflow to Positive Exercise

We gave an example above in *Type int*, adding two positive `int` values and clearly having an error, since the result was negative. Declare and initialize two positive `int` variables `x` and `y`. Experiment with the initializations so

1. Their product is too big to fit in an `int` AND

2. The wrong overflow result for `x*y` is *positive*, not negative.

2.13 Learning to Solve Problems

This section might have been placed earlier, but from reading all the way to here, you should realize that you will have a *lot* of data and concepts to deal with.

The manner in which you deal with all the data and ideas is very important for effective learning. It might be rather different than what you needed if you were in a situation where *rote* recall is the main important thing.

Different learning styles mean different things are useful to different people. Consider what is mentioned here and try out some approaches.

The idea of this course is *not* to regurgitate the book, but to learn to solve problems (generally involving producing a computer program). In this highly connected and wired world you have access to all sorts of data. The data is not an end in itself, the question is *doing* the right things with the tools out there to solve a new creative problem.

In this course there is a lot of data tied into syntax and library function names and .... It can seem overwhelming. It need not be. Take a breath.

First basic language syntax: When learning any new language, there is a lot to take in. We introduce C# in chunks. For a while there will always be the new current topic coming. You do NOT need to memorize *everything* immediately!

- Some things that you use rarely, you may never memorize, like, "What is the exact maximum magnitude of a double?" At *some* point that might be useful. Can you find it? It happens to be in *Numeric Types and Limits*. It is also in online .Net documentation that you can Google or bookmark.

- Some things you will use all the time, but of course they start off as new and maybe strange. Knowing where to go to check is still useful but not sufficient. For much-used material that you do not find yourself absorbing immediately, consider writing down a summary of the current topic. Both thinking of a summary and writing help reinforce things and get you to remember faster. Also, if you have the current things of interest summarized in one place, they are easy to look up!

- If you need some syntax to solve a simple early problem, first try to remember the syntax, then check. With frequently used material and with this sort of repetition, most everyone will remember most everything shortly. If there are a few things that just do not stick, keep them in your list. Then go on

to new material. The list of what you need to check on will keep changing as you get more experience and get to more topics. If you keep some of the old lists, you will be amazed how much stuff that you sweated over, is later ho-hum or automatic.

- In the earliest exercises the general steps that you need should be pretty apparent, and you can just concentrate on translating simple ideas into C# syntax (mostly from the material most recently introduced). In this case the focus is mostly on syntax.

Memorizing syntax is not going to directly get you to solve real problems. In any domain: programming, construction, organizing political action, ..., you need to analyze the problem and figure out a sequence of steps, knowing what *powers and resources you have*.

For example with political action: if you know demonstrations are possible in front of City Hall, you can make a high-level plan to have one, but then you have to attend to details: Do you need city permission? Who do you call? ... You do not have to have all that in your head when coming up with the idea of the demonstration, but you better know how to find the information allowing you to follow through to make it happen.

With programming, syntax details are like the details above: not the first thing to think of, and maybe not things that you have memorized. What *is* important to break down a problem and plan a solution, is to know the basic *capacities* you have in programming. As you get into larger projects and have more experience, "basic capacities" will be bigger and bigger ideas. For now, as beginners, based on the sections of the book so far, it is important to know:

- You can get information from a user and return information via keyboard and screen.

- You can remember and recall and use information using variables.

- You can deal directly with various kinds of data: numbers and strings at this point.

- There are basic operations you can do with the data (arithmetic, concatenating string, converting between data types).

- At a slightly higher level, you might already have the idea of basic recurring patterns, like solving a straightforward problem with **input-processing-output**.

- You will see shortly that you have more tools: decision, repetition, more built-in ways to deal with data (like more string operations shortly), creating your own data types....

At slightly more detailed level, *after* thinking of overall plans:

- There are multiple kinds of number types. What is appropriate for your use?

- There are various ways of formatting and presenting data to output. What shall you use?

Finally, you actually need to translate specific instructions into C# (or whatever language). Of course if you remember the syntax, then this level of step is pretty automatic. Even if you do *not* remember, you have something very specific to look up! If you are keeping track of your sources of detailed information, this is hopefully only one further step.

Contrast this last-step translation with the earlier creative organizational process: If you do not have *in your head* an idea of the basic tools available, how are you going to plan? How are you going to even know how to start looking something up?

So far basic ideas for planning a solution has been discussed, and you can see that you do not need to think of everything at once or have everything equally prominent in your brain.

Also, when you are coding, you do not need to to have all the details of syntax in your head, even for the *one* instruction that you are dealing with at the moment. You want to have the main idea, and you want to get it written down, but once it is written down, you can make multiple passes, examining and modifying what you have. For example, Dr. Harrington does a lot of Python programming, where semicolons are not needed. He can get the main ideas down in C# without the required semicolons. He *could* wait for the compiler to stop him on every one that is missed, and maybe have the compiler misinterpret further parts, and give bogus error messages. *More effective* is having a list of things to concentrate on in later rounds of manual checking. For example, checking for semicolons: Scan the statements; look at the ends; add semicolons where missing. You can go through a large program very quickly and efficiently doing this and have one less thing to obsess about when first writing.

This list of things-to-check-separately should come from experience. Keep track of the errors you make. Some people even keep an error log. What errors keep

occurring? Make entries in things-to-check-separately, so you will make scans checking for the specific things that you frequently slip up on.

This things-to-check-separately list, too, will evolve. Revise it occasionally. If Dr. Harrington does enough concentrated C#, *maybe* he will find that entering semicolons becomes automatic, and he can take the separate round of semicolon checking off his list....

What to do *after* you finish an exercise is important, too. The natural thing psychologically, particularly if you had a struggle, is to think, "Whew, outta here!!!!"

On something that came automatically or flowed smoothly, that is not a big deal - you will probably get it just as fast the next time. If you had a hard time and only eventually got to success, you may be doing yourself a disservice with "Whew, outta here!!!"

We have already mentioned how not everything is equally important, and some things are more important to keep in your head than others. The same idea applies to all the steps in solving a possibly long problem. Some parts were easy; some were hard; there may have been many steps. If all of that goes into your brain in one continuous stream of stuff that you remember at the same level, then you are going to leave important nuggets mixed in with an awful lot of unimportant and basically useless information, and have it all fade into oblivion, or be next to impossible to cycle through looking for the nuggets. Why do the problem anyway if you are just going to bury important information further down in your brain?

What is important? The most obvious thing you will need at a higher level of recall is what *just messed you up*, what you missed until doing this problem: After finishing the actual problem, *actively* follow up and ask yourself:

- What did I get in the end that I was missing initially? What was the connection I made?

- Does this example fit in to some larger idea/abstraction/generalization in a way that I did not see before?

- How am I going to look at this so I can make a similar connection in a similar (or maybe only partly similar) problem?

- Is there a kernel here that I can think of as a new tool in my bag of tricks?

Your answers to these questions are the most important things to take away from your recent hard work. The extra consideration puts them more in the "priority" part of your brain, so you can really learn from your effort. When you need the

important ideas next, you do not need to play through all the details of the stuff you did to solve the exact earlier problem.

Keep coming back to this section and check up on your process: It is really important.

2.14 Lab: Division Sentences

2.14.1 Overview

In this lab, we're going to begin to look at what makes computers *do their thing* so to speak.

It is rather insightful to look at how Wikipedia summarizes the computer:

> A computer is a programmable machine designed to sequentially and automatically carry out a sequence of arithmetic or logical operations. The particular sequence of operations can be changed readily, allowing the computer to solve more than one kind of problem.

In other words, a computer is a calculator–and much more. Furthermore, the definition of a computer goes on to include access to storage and peripherals, such as consoles (graphical displays), printers, and the network. We already got a glimpse of this access when we explored `Console.WriteLine` in the first lab exercise.

We have discussed all the syntax and concepts needed in recent sections on *Arithmetic*, *Variables and Assignment*, *Combining Input and Output*, and *Casting*. Also you can make things easier for yourself using *Substitutions in Console.WriteLine* to format output.

Before writing your final program, you might like to review some of the parts, testing in the *Csharp* program, so you get immediate feedback for the calculations.

2.14.2 Requirements

We want to develop a program that can do the following:

- Prompt the user for input of two integers, which we will call *numerator* and *denominator*. For clarity, we are only looking at integers, because this assignment is about rational numbers. A rational number can always be expressed as a quotient of two integers.

- Calculate the floating point division result (e.g. 10/4 = 2.5).

- Calculate the quotient and the remainder (e.g. 10/4 = 2 with a remainder of 2 = 2 2/4).

Your final program should work as in this sample run, and use the same labeled format:

```
Please enter the numerator? 14
Please enter the denominator? 4
Integer division result = 3 with a remainder 2
Floating point division result = 3.5
The result as a mixed fraction is 3 2/4.
```

For this lab the example format `3 2/4` is sufficient. It would look better as `3 1/2`, but a general efficient way to reduce fractions to lowest terms is not covered until the section on the algorithm *Greatest Common Divisor*.

To do the part requiring a decimal quotient you are going to need to have a `double` value, though your original data was of type `int`. You could use the approach in *Casting*, with an explicit cast. Another approach mentioned in that section was to do the cast implicitly in a `double` declaration with initialization from an `int`. If we already had `int` variables, `numerator` and `denominator`, that were previously assigned their values, we could use:

```
double numeratorDouble = numerator; // implicit cast
double quotientDouble = numeratorDouble/denominator;
...
```

Remember: at least one operand in a quotient must be `double` to get a `double` result.

To help you get started with your program code, we provided this simple *stub* in the example file do_the_math_stub/do_the_math.cs. You are encouraged to copy this into your own project as reviewed after the lab in *Xamarin Studio Reminders and Fixes*.

The body of `Main` presently contains only *comments*, skipped by the compiler. We illustrate two forms (being inconsistent for your information only):

- // to the end of the *same* line

- /* to */ through any number of lines.

Save the stub in a project of your own and replace the comments with your code to complete it:

```
using System;

class DoTheMath {   // Lab stub
   static void Main() {
      /* Prompt the user for the numerator using
         Console.Write().

         Convert this text into int numerator using
         int.Parse().

         Do the same for the denominator.

         Calculate quotient and remainder (as integers)
         Use Console.WriteLine() to display the labels
         as illustrated in the sample output in the lab.

         Do the same but using floating point division
         and not doing the remainder calculation.

         Create the sentence with the mixed fraction.
         Be careful of the places there are *not* spaces.
      */
   }
}
```

Be sure to run it and test it thoroughly. Show your output to a TA.

2.14.3 Xamarin Studio Reminders and Fixes

Be careful to open your Xamarin Studio solution and add a new **C# Console project** to it, and add your new file directly into the project (through the Solution pad). There are two main places to mess up here. We emphasize them and mention fixes if you make the easy mistakes:

1. It is easy to select Empty Project instead of C# and Console Project. If

you do that, a correct program will compile successfully, but it will run in limbo, with no console attached to it, and all `Console.ReadLine()` calls return `null`, which is likely to make the program have a run-time error. One way to fix it:

- If you discovered this while running your program, there is no good access to the running process. (You lack a console!) In this case you need to close your solution, ending the running process, and open the solution again.

- Double click on the project in the Solution Pad (if that does anything, or right-click it and select Options). An elaborate Project Options dialog window appears.

- In the left pane under Run, select General. In the right pane, two check boxes should appear. Make sure you have the first checked: Run on external console. That should check the second one automatically. Close the window and you should be set.

  Be careful, it is possible to uncomment the second checkbox, which makes your execution console close instantly at the end of your program, so you miss any last thing printed. Recheck if necessary.

2. Another common error is to proceed like with most text processors, and open the top File menu, and choose to open and edit a new file for your program. *You cannot* run* **this program from Xamarin Studio.** The file you edit must show in the solution pad in Xamarain Studio, as a source file in your project. If you have a separate project set up, but without this file or any other showing in the Solutions pad, an attempt to run the project with say no `Main` method (in fact no program at all). The fix:

   - You will shortly need to navigate in an operating system open file dialog to where you put the file created from the File menu. If you do not remember where that was, a good trick is to click in the edit window of the file and then go to the File menu and select Save As. The dialog should show where the file currently is. Cancel the dialog.

   - Right click on the project in the Solution pad, and choose Add and then Add Files.... Browse to where the file is and select it; click Open. Unless you have some reason to keep a copy in the original place, select Move, and Ok. Now the orphaned file is moved into your project. You should see it list under the project in the Solution pad. You can

proceed to edit and run it.

3. If you lose the display of the Solution pad somehow, you can go to the View menu, select Pads, and then select Solution.

2.15 Chapter Review Questions

1. Most of C# arithmetic is just like normal math. The exceptions are most important: What are they?

2. What are the consequences of numerical value types each being stored in a fixed amount of memory space?

3. What is the order of operations if several of the same level, are chained together like x + y + z? This can matter in C# (including in a question below).

4. Which of these individual two-line fragments could fit into a legal C# program? Explain:

```
w = 3;
w = 4;

x = 3;
x = x+1;

y = 3;
y = "hello";

z = 3.5
z = 3;

q = 3;
q + 1 = q;
```

5. For the legal pairs above, what could the type of the variable have been declared? You can check them in csharp, giving the declaration first.

6. What are the final values of x and y after this fragment?

```
int x = 3
int y = x + 2;
x = x + y;
y = x + y;
```

Test in csharp after you have decided.

7. Which of these expressions are legal in C#? Think of the results. Explain.

```
"a" + "b"
"a" + 'b'
"a" + 2
2 + "a"
"a" + 2 * 3;
"a" + 2 + 3
2 + 3 + "a"
2 + 3 * "a"
```

Think first; try in csharp; reconsider if necessary.

8. Write a single `WriteLine` statement that would produce output on two separate lines, not one. Accomplish the same thing a second time with fundamentally different syntax.

9. What is printed?

```
Console.WriteLine("{1} {0} {2} {1} {0}", 'B', 2, "or not");
```

10. Which of these casts is necessary, and which could be left out (and be legal and mean the same thing)? Before testing, think what the values of the variables will be for the first two or three:

```
int x= (int)5.8;
double y = (double)6;
char c = (char)('a' + 1)
int z = (int)'a' + 1;
```

DEFINING FUNCTIONS OF YOUR OWN

3.1 A First Function Definition

If you know it is the birthday of a friend, Emily, you might tell those gathered with you to sing "Happy Birthday to Emily".

We can make C# display the song. *Read*, and run if you like, the example program birthday1/birthday1.cs:

```csharp
using System;

class Birthday1
{
   static void Main ()
   {
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday, dear Emily.");
      Console.WriteLine ("Happy Birthday to you!");
   }
}
```

Here the song is just a part of the `Main` method that is in every program.

Note that we are using a function already provided to us, `Console.WriteLine`. We can use it over and over, wherever we like. We can alter its behavior by including a different parameter. Now we look further at writing and using your own functions.

If we want this song to be just part of a larger program, and be able to refer to it repeatedly and easily, we might like to package it separately. You would probably not repeat the whole song to let others know what to sing. You would give a request to sing via a descriptive name like "Happy Birthday to Emily".

In C# we can also give a name like `HappyBirthdayEmily`, and associate the name with whole song by using a new *function definition*, also called a *method*. We will see many variations on method definitions. Later we will see definitions that are attached to a particular object. For now the simpler cases do not involve creating a type of object, but there is an extra word, `static`, needed to distinguish a function definition *not* attached to on object. We will also shortly look at functions more like the functions from math class, that produce or *return* a value. In this simple case we will not deal with returning a value. This also requires a special word in the heading: `void`. A `void` function will just be a shorthand name for something to do, a procedure to follow, in this case printing out the Happy Birthday song for Emily. (Note that the `Main` method for a program is also `static void`. This *does* your whole program and is not attached to an object.)

Read for now:

```csharp
using System;

class Birthday2
{
   static void HappyBirthdayEmily()
   {
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday, dear Emily.");
      Console.WriteLine ("Happy Birthday to you!");
   }

   static void Main()
   {
      HappyBirthdayEmily();
      Console.WriteLine ("Hip hip hooray!");
      HappyBirthdayEmily();
   }
}
```

There are several parts of the syntax for a function definition to notice:

Line 5: The *heading* starts with `static void`, the name of the function, and then parentheses.

A more general syntax for functions that just *do* something is

> `static void` **FunctionName** `()`
> `{`
>
> > statements in the function body...
>
> `}`

Recall the conventions in *Syntax Template Typography*.

Lines 6-11: The remaining lines form the function *body*. They are enclosed in braces. By convention the lines inside the braces are indented by a consistent amount. Three spaces is a common indentation.

The whole definition does just that: *defines* the meaning of the name `HappyBirthdayEmily`, but it does not do anything else yet - for example, the definition itself does not yet make anything be printed. This is our first example of altering the order of execution of statements from the normal sequential order. This is important: the statements in the function *definition* are *not* executed as C# first passes over the lines. The only part of a program that is automatically executed is `Main`. Hence `Main` better refer to the newly defined function....

Look at the first statement inside Main, line 15:

```
HappyBirthdayEmily();
```

Note that the `static void` of the function definition is missing, but we still have the function name and parentheses. When `Main` is running, C# goes back and looks up the definition, and only then, executes the code inside the function definition. The term for this action is a *function call* or function *invocation*. In this simple situation the format is

> *FunctionName* `()`

While the convention for variable identifiers is to start with a lowercase letter, the convention for function names is to start with a capital letter. Hence `HappyBirthdayEmily`, not `happyBirthdayEmily`.

Can you predict what the program will do? Note the two function calls to `HappyBirthdayEmily`. To see, load and run birthday2/birthday2.cs.

The *execution* sequence for the program is different from the *textual* sequence. Execution always starts in Main:

1. Line 13: Main is where execution starts, and initially proceeds sequentially.

2. Line 15: the function is called while this location is remembered.

3. Lines 5-11: Jump! The code of the function is executed for the first time, printing out the song.

4. End of line 15: Back from the function call; continue on.

5. Line 16: Just to mix things up, print out a "Hip, hip, hooray".

6. Line 17: the function is called again while this location is remembered.

7. Lines 5-11: The function is executed again, printing out the song again.

8. End of line 17: Back from the function call, but at this point there is nothing more in `Main`, and execution stops.

Functions alter execution order in several ways: by statements not being executed as the definition is first read, and then when the function is called during execution, jumping to the function code, and back at the the end of the function execution.

Understanding the jumping around in the code with function calls is crucial. Be sure you follow the sequence detailed above. In particular, be sure to distinguish function **definition** from function **call**.

If it also happens to be Andre's birthday, we might define a function `HappyBirthdayAndre`, too. Think how to do that before going on ....

3.2 Multiple Function Definitions

Here is example program birthday3/birthday3.cs where we add a function `HappyBirthdayAndre`, and call them both. Guess what happens, and then load and try it:

```csharp
using System;

class Birthday3
{
    static void Main()
```

```
   {
      HappyBirthdayEmily();
      HappyBirthdayAndre();
   }

   static void HappyBirthdayEmily()
   {
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday, dear Emily.");
      Console.WriteLine ("Happy Birthday to you!");
   }

   static void HappyBirthdayAndre()
   {
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday, dear Andre.");
      Console.WriteLine ("Happy Birthday to you!");
   }
}
```

Again, definitions are remembered and execution starts in `Main`. The order in which the function definitions are given does not matter to C#. It is a human choice. For variety I show `Main` first. This means a human reading in order gets an overview of what is happening by looking at Main, but does not know the details until reading the definitions of the birthday functions.

Detailed order of execution:

1. Line 5: Start on `Main`

2. Line 7. This location is remembered as execution jumps to `HappyBirthdayEmily`

3. Lines 11-17 are executed and Emily is sung to.

4. Return to the end of Line 7: Back from `HappyBirthdayEmily` function call

5. Line 8: Now `HappyBirthdayAndre` is called as this location is remembered.

6. Lines 19-25: Sing to Andre

7. Return to the end of line 8: Back from `HappyBirthdayAndre` function call, done with `Main`; at the end of the program

The calls to the birthday functions *happen* to be in the same order as their definitions, but that is arbitrary. If the two lines of the body of `Main` were swapped, the order of operations would change, but if the order of the whole function definitions were changed, it would make no difference in execution.

Functions that you write can also call other functions you write. In this case Main calls each of the birthday functions.

> **Warning:** A common compiler error is caused by failing to match the braces that wrap a function body. A new function heading can only exist outside all other function declarations and inside a class. If you have too few or extra `'}'` you are likely to find a perfectly fine looking function heading with an error, for instance, about not allowing `static` here.... Check your earlier lack or excess of braces!

Xamarin Studio, like other modern code editors, can show you matching delimiters. If you place your cursor immediately after a delimiter { } ( ) [ ], the matching one should become highlighted.

3.2.1 Poem Function Exercise

Write a program, `poem.cs`, that defines a function that prints a *short* poem or song verse. Give a meaningful name to the function. Have the program call the function three times, so the poem or verse is repeated three times.

3.3 Function Parameters

As a young child, you probably heard Happy Birthday sung to a couple of people, and then you could sing to a new person, say Maria, without needing to hear the whole special version with Maria's name in it word for word. You had the power of *abstraction*. With examples like the versions for Emily and Andre, you could figure out what change to make it so the song could be sung to Maria!

Unfortunately, C# is not that smart. It needs explicit rules. If you needed to explain *explicitly* to someone how Happy Birthday worked in general, rather than

just by example, you might say something like this:

First you have to be *given* a person's name. Then you sing the song with the person's name inserted at the end of the third line.

C# works something like that, but with its own syntax. The term "person's name" serves as a stand-in for the actual data that will be used, "Emily", "Andre", or "Maria". This is just like the association with a variable name in C#. "person's name" is not a legal C# identifier, so we will use just person as this stand-in. It will be a variable in the program, so it needs a type in C#. The names are strings, so the type of person is string.

In between the parentheses of the function definition heading, we insert the variable name person, preceded by its type, string. Then in the body of the definition of the function, person is used in place of the real data for any specific person's name. Read and then run example program birthday4/birthday4.cs:

```csharp
using System;

class Birthday4
{
    static void HappyBirthday(string person)
    {
        Console.WriteLine ("Happy Birthday to you!");
        Console.WriteLine ("Happy Birthday to you!");
        Console.WriteLine ("Happy Birthday, dear " + person + ".");
        Console.WriteLine ("Happy Birthday to you!");
    }

    static void Main()
    {
        HappyBirthday("Emily");
        HappyBirthday("Andre");
    }

}
```

In the definition heading for HappyBirthday, person is referred to as a *parameter*, or a *formal parameter*. This variable name is a *placeholder* for the real name of the person being sung to. In the definition we give instructions for singing Happy Birthday *without* knowing the exact name of the person who might be sung to.

`Main` now has two calls to the same function, but between the parentheses, where there was the **placeholder** `person` in the definition, now we have the **actual people** being sung to. The value between the parentheses here in the function call is referred to as an *argument* or *actual parameter* of the function call. The argument supplies the actual data to be used in the function execution. When the call is made, C# does this by associating the **formal** parameter name `person` with the **actual** parameter data, as in an assignment statement. In the first call, this actual data is `"Emily"`. We say the actual parameter value is *passed* to the function for execution.

The execution in greater detail:

1. Lines 13: Execution starts in Main.

2. Line 15: Call to `HappyBirthday`, with actual parameter `"Emily"`.

3. Line 5: `"Emily"` is passed to the function, so `person = "Emily"`.

4. Lines 7-10: The song is printed, with `"Emily"` used as the value of `person` in line 9: printing

   ```
   Happy Birthday, dear Emily.
   ```

5. End of line 15 after returning from the function call

6. Line 16: Call to `HappyBirthday`, this time with actual parameter `"Andre"`

7. Line 5: `"Andre"` is passed to the function, so `person = "Andre"`.

8. Lines 7-10: The song is printed, with `"Andre"` used as the value of `person` in line 9: printing

   ```
   Happy Birthday, dear Andre.
   ```

9. End of line 16 after returning from the function call, and the program is over.

The beauty of this system is that the same function definition can be used for a call with a different actual parameter variable, and then have a different effect. The value of the variable person is used in the third line of `HappyBirthday`, to put in whatever actual parameter value was given.

This is the power of *abstraction*. It is one application of the most important principal in programming. Rather than have a number of separately coded parts with

only slight variations, see where it is appropriate to combine them using a function whose parameters refer to the parts that are different in different situations. Then the code is written to be simultaneously appropriate for the separate specific situations, with the substitutions of the right parameter values.

Note: Be sure you completely understand birthday4/birthday4.cs and the sequence of execution! It illustrates extremely important ideas that many people miss the first time! It is essential to understand the difference between

1. *Defining* a function (lines 5-11) with the heading including *formal* parameter name and type, where the code is merely instructions to be remembered, not acted on immediately.

2. *Calling* a function with an *actual* parameter value to be substituted for the formal parameter, (with *no* type included!) and have the function code actually *run* when the instruction containing the call is run. Also note that the function can be called multiple times with different expressions as the actual parameter (line 15 and again in line 16).

We can combine function parameters with user input, and have the program be able to print Happy Birthday for anyone. Check out the `Main` method and run birthday_who/birthday_who.cs:

```csharp
using System;

class Birthday_Who
{
   static void HappyBirthday(string person)
   {
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday to you!");
      Console.WriteLine ("Happy Birthday, dear " + person + ".");
      Console.WriteLine ("Happy Birthday to you!");
   }

   static void Main()
   {
      string userName;
      Console.WriteLine("Who would you like to sing Happy Birthday to?");
      userName = Console.ReadLine();
      HappyBirthday(userName);
```

```
19        }
20
21  }
```

This last version illustrates several important ideas:

1. There are more than one way to get information into a function:

   (a) Have a value passed in through a parameter (from line 18 to line 5).

   (b) Prompt the user, and obtain data from the keyboard (lines 16-17).

2. It is a good idea to separate the *internal* processing of data from the *external* input from the user by the use of distinct functions. Here the user interaction is in `Main`, and the data is manipulated in `HappyBirthday`.

3. In the first examples of actual parameters, we used literal values. In general an actual parameter can be an expression. The expression is evaluated before it is passed in the function call. One of the simplest expressions is a plain variable name, which is evaluated by replacing it with its associated value. Note this important situation in the example: We have the value of `userName` in `Main` becoming the value of `person` in `HappyBirthday`. We used different names to illustrate the important fact:

   Note: Only the *value* of the actual parameter is passed, not any variable name, so there is *no need* to have a match between a variable name used in an actual parameter and the formal parameter name.

3.3.1 Birthday Function Exercise

Make your own further change to birthday4/birthday4.cs and save it in your own project as `birthday_many.cs`: Add a function call (but *not* another function *definition*), so Maria gets a verse, in addition to Emily and Andre. Also print a blank line between verses. (There are two ways to handle the blank lines: You may *either* do this by adding a print line to the function definition, *or* by adding a print line between all calls to the function. Recall that if you give Console.WriteLine an empty parameter list, it just goes to the next line.)

3.4 Multiple Function Parameters

A function can have more than one parameter in a parameter list. The list entries are separated by commas. Each formal parameter name is preceded by its type. The example program addition1/addition1.cs uses a function, SumProblem, with two parameters to make it easy to display many sum problems. Read and follow the code, and then run:

```csharp
using System;

class Addition2
{
   static string SumProblemString(int x, int y)
   {
      int sum = x + y;
      string sentence = "The sum of " + x + " and " + y + " is " + sum + ".";
      return sentence;
   }

   static void Main()
   {
      Console.WriteLine(SumProblemString(2, 3));
      Console.WriteLine(SumProblemString(12345, 53579));
      Console.Write("Enter an integer: ");
      int a = int.Parse(Console.ReadLine());
      Console.Write("Enter another integer: ");
      int b = int.Parse(Console.ReadLine());
      Console.WriteLine(SumProblemString(a, b));
   }
}
```

The actual parameters in the function call are evaluated left to right, and then these values are associated with the formal parameter names in the function definition, also left to right. For example a function call with actual parameters, F(actual1, actual2, actual3), calling a function F with definition heading:

```csharp
static void F(int formal1, int formal2, int formal3)
```

acts approximately as if the first lines executed inside the called function F were

---

```
formal1 = actual1;
formal2 = actual2;
formal3 = actual3;
```

Functions provide extremely important functionality to programs, allowing tasks to be defined once and performed repeatedly with different data. It is essential to see the difference between the **formal** parameters used to describe what is done inside the function definition (like x and y in the definition of SumProblem) and the **actual** parameters (like 2 and 3 or 12345 and 53579) which *substitute* for the formal parameters when the function is actually executed. Main uses three different sets of actual parameters in the three calls to SumProblem.

Warning: It is easy to confuse the heading in a function *definition* and a *call* to actually execute that function. Be careful. In particular, do *not* list the types of parameters in a call's *actual* parameter list. The actual parameters are expressions involving terms that are *already defined*, not just being declared.

3.4.1 Quotient Function Exercise

Modify quotient_format.cs from *Exercise for Format* and save it as quotient_prob.cs. You should create a function QuotientProblem with int parameters. Like in the earlier versions, it should print a full sentence with inputs, integer quotient, and remainder. Main should test the QuotientProblem function on several sets of literal values, and also test the function with input from the user.

3.5 Returned Function Values

You probably have used mathematical functions in algebra class, but they all had calculated values associated with them. For instance if you defined

$$F(x)=x^2$$

then it follows that F(3) is $3^2 = 9$, and F(3)+F(4) is $3^2 + 4^2 = 9 + 16 = 25$.

Function calls in expressions get replaced during evaluation by the value of the function.

The corresponding definition and examples in C# would be the following, taken from example program `return1.cs`. *Read and run*:

```
using System;

class Return1
{
   static int F(int x)
   {
      return x*x;
   }

   static void Main()
   {
      Console.WriteLine(F(3));
      Console.WriteLine(F(3) + F(4));
   }
}
```

The new C# syntax is the *return statement*, with the word `return` followed by an expression. Functions that return values can be used in expressions, just like in math class. When an expression with a function call is evaluated, the function call is effectively replaced temporarily by its returned value. Inside the C# function, the value to be returned is given by the expression in the `return` statement.

Since the function returns data, and all data in C# is typed, there must be a type given for the value returned. Note that the function heading does not start with `static void`. In place of `void` is `int`. The `void` in earlier function headings meant nothing was returned. The `int` here means that a value *is* returned and its type is `int`.

After the function `F` finishes executing from inside

```
Console.WriteLine(F(3));
```

it is as if the statement temporarily became

```
Console.WriteLine(9);
```

and similarly when executing

```
Console.WriteLine(F(3) + F(4));
```

3.5. Returned Function Values **101**

the interpreter first evaluates F(3) and effectively replaces the call by the returned result, 9, as if the statement temporarily became

```
Console.WriteLine(9 + F(4));
```

and then the interpreter evaluates F(4) and effectively replaces the call by the returned result, 16, as if the statement temporarily became

```
Console.WriteLine(9 + 16);
```

resulting finally in 25 being calculated and printed.

C# functions can return any type of data, not just numbers, and there can be any number of statements executed before the return statement. Read, follow, and run the example program `return2.cs`:

```csharp
using System;

class Return2
{
    static string LastFirst(string firstName, string lastName)
    {
        string separator = ", ";
        string result = lastName + separator + firstName;
        return result;
    }

    static void Main()
    {
        Console.WriteLine(LastFirst("Benjamin", "Franklin"));
        Console.WriteLine(LastFirst("Andrew", "Harrington"));
    }
}
```

Many have a hard time following the flow of execution with functions. Even more is involved when there are return values. Make sure you completely follow the details of the execution:

1. Lines 12: Start at Main

2. Line 14: call the function, remembering where to return

3. Line 5: pass the parameters: `firstName = "Benjamin"`; `lastName = "Franklin"`

4. Line 7: Assign the variable `separator` the value `", "`

5. Line 8: Assign the variable `result` the value of `lastName + separator + firstName` which is `"Franklin" + ", " + "Benjamin"`, which evaluates to `"Franklin, Benjamin"`

6. Line 9: Return `"Franklin, Benjamin"`

7. Line 14: Use the value returned from the function call so the line effectively becomes `Console.WriteLine("Franklin, Benjamin");`, so print it.

8. Line 15: call the function with the new actual parameters, remembering where to return

9. Line 5: pass the parameters: `firstName = "Andrew"; lastName = "Harrington"`

10. Lines 7-9: ... calculate and return `"Harrington, Andrew"`

11. Line 15: Use the value returned by the function and print `"Harrington, Andrew"`

Compare return2/return2.cs and addition1/addition1.cs, from the previous section. Both use functions. Both print, but where the printing *is done* differs. The function `SumProblem` prints directly inside the function and returns nothing. On the other hand `LastFirst` does not print anything but returns a string. The caller gets to decide what to do with the returned string, and above it is printed in `Main`.

In general functions should do a single thing. You can easily combine a sequence of functions, and you have more flexibility in the combinations if each does just one unified thing. The function SumProblem in addition1/addition1.cs does two thing: It creates a sentence, and prints it. If that is all you have, you are out of luck if you want to do something different with the sentence string. A better approach is to have a function that just creates the sentence, and returns it for whatever further use you want. After returning that value, printing is one possibility, done in addition2/addition2.cs:

```csharp
using System;

class Addition2
{
    static string SumProblemString(int x, int y)
    {
```

```
      int sum = x + y;
      string sentence = "The sum of " + x + " and " + y + " is " + sum + ".",
      return sentence;
   }

   static void Main()
   {
      Console.WriteLine(SumProblemString(2, 3));
      Console.WriteLine(SumProblemString(12345, 53579));
      Console.Write("Enter an integer: ");
      int a = int.Parse(Console.ReadLine());
      Console.Write("Enter another integer: ");
      int b = int.Parse(Console.ReadLine());
      Console.WriteLine(SumProblemString(a, b));
   }
}
```

This example constructs the sentence using the string + operator. Generating a string with substitutions using a format string in Console.Write is neater, but we are forced to directly print the string, and not remember it for later arbitrary use.

It is common to want to construct and immediately print a string, so having Console.Write is definitely handy when we want it., However it is an example of combining two separate steps! Sometimes (like here) we just want to have the resulting string, and do something else with it. We introduce the C# library function string.Format, which does just what we want: The parameters have the same form as for Console.Write, but the formatted string is *returned*.

Here is a revised version of the function SumProblemString, from example addition2a/addition2a.cs:

```
static string SumProblemString(int x, int y) // with string.Format
{
   int sum = x + y;
   return string.Format("The sum of {0} and {1} is {2}.", x, y, sum);
}
```

The only caveat with string.Format is that there is *no* special function corresponding to Console.WriteLine, with an automatic terminating newline. You can generate a newline with string.Format: Remember the escape code "\n". Put it at the end to go on to a new line.

In class recommendation: Improve example painting/painting.cs with a function used for repeated similar operations. Copy it to a file `painting_input.cs` in your own project and modify it.

3.5.1 Interview String Return Exercise/Example

Write a program by that accomplishes the same thing as *Interview Exercise/Example*, but introduce a function `InterviewSentence` that takes name and time strings as parameters and returns the interview sentence string. For practice use `string.Format` in the function. With this setup you can manage input from the user and output to the screen entirely in `Main`, while using `InterviewSentence` to generate the sentence that you want to *later* print.

(Here we are having you work on getting used to function syntax while keeping the body of your new function very simple. Combining that with longer, more realistic function bodies is coming!)

If you want a further example on this idea of returning something first and then using the result, or if you want to compare your work to ours, see our solution, interview2/interview2.cs.

3.5.2 Quotient String Return Exercise

Create `quotient_return.cs` by modifying `quotient_prob.cs` in *Quotient Function Exercise* so that the program accomplishes the same thing, but everywhere:

- Change the QuotientProblem function into one called `QuotientString` that merely *returns* the string rather than printing the string directly.

- Have `Main` print the result of each call to the `QuotientString` function.

Use `string.Format` to create the sentence that you return.

3.6 Two Roles: Writer and Consumer of Functions

The remainder of this section covers finer points about functions that you might skip on a first reading.

We are only doing tiny examples so far to get the basic idea of functions. In much larger programs, functions are useful to manage complexity, splitting things up into logically related, modest sized pieces. Programmers are both writers of functions and consumers of the other functions called inside their functions. It is useful to keep those two roles separate:

The user of an already written function needs to know:

1. the name of the function

2. the order and meaning of parameters

3. what is returned or produced by the function

How this is accomplished is not relevant at this point. For instance, you use the work of the C# development team, calling functions that are built into the language. You need know the three facts about the functions you call. You do not need to know exactly *how* the function accomplishes its purpose.

On the other hand when you *write* a function you need to figure out exactly how to accomplish your goal, name relevant variables, and write your code, which brings us to the next section.

The jargon for these parts are the *interface* (for the consumer) and the *implementation* (for the programmer, who must be sure to satisfy the public interface).

3.7 Local Scope

For the logic of writing functions, it is important that the writer of a function knows the names of variables inside the function. On the other hand, if you are only using a function, maybe written by someone unknown to you, you should not care what names are given to values used internally in the implementation of the function you are calling. C# enforces this idea with *local scope* rules:

Variable names initialized and used inside one function are *invisible* to other functions. Such variables are called *local* variables. For example, an elaboration of the earlier program return2/return2.cs might have its `lastFirst` function with its local variable `separator`, but it might also have another function that defines a `separator` variable, maybe with a different value like `"\n"`. They would not conflict. They would be independent. This avoids lots of errors!

For example, the following code in the example program bad_scope/bad_scope.cs causes a compilation error if the last line is uncommented. Read it, uncomment the line, and try to run it, and see:

```csharp
using System;

class BadScope
{
   public static void Main()
   {
      int x = 3;
      F();
   }

   static void F()
   {  // F doesn't know about the x defined in Main
      //Console.WriteLine(x); //ERROR if uncommented
   }
}
```

The compilation error that Mono gives is pretty clear:

> The name 'x' does not exist in the current context.

The the error occurs in the function `F`. The *context* there just includes the local variables already declared in `F`. And `x` is declared in `Main`, not in `F`, so it *does not exist* inside `F`. We will fix this error below.

If you do want local data from one function to go to another, define the called function so it includes parameters! Read and compare and try the program good_scope/good_scope.cs:

```csharp
using System;

class GoodScope
{
```

```
   static void Main()
   {
      int x = 3;
      F(x);
   }
   static void F(int x)
   {
      Console.WriteLine(x);
   }

}
```

With parameter passing, the parameter name `x` in the function `F` does not need to match the name of the actual parameter in the calling function `Main`. (Just the `int` value is passed.) The definition of `F` could just as well have been:

```
static void F(int whatever)
{
   Console.WriteLine(whatever);
}
```

> **Warning:**   It is a very common error to declare a variable in one function and try to use it by name in a different function. If you get an error about a variable not existing, look to see where it was declared (or if you remembered to declare it at all)!

In general the *scope* of a variable is the places in the program where its value can be referenced and used. The scope of a local variable is just inside the function where it is declared.

3.8 Static Variables

You may define *static variables* (variables defined with the word `static` inside the class, but *outside* of any function definition). These variables are visible inside all of your functions in the class. Instead of local scope, static variables have *class scope*. It is good programming practice generally to avoid defining static variables and instead to put your variables inside functions and explicitly pass them as parameters where needed. There are exceptions. For now a good reason

for using static variables is constants: A *constant* is a name that you give a fixed data value to. If you have a static definition of a constant, then you can then use the name of the fixed data value in expressions in any function in the class. A simple example program is constant/constant.cs:

```csharp
using System;

class UseConstant
{
    static double PI = 3.14159265358979; // constant, value not reset

    static double CircleArea(double radius)
    {
        return PI*radius*radius;
    }

    static double Circumference(double radius)
    {
        return 2*PI*radius;
    }

    static void Main()
    {
        Console.WriteLine("circle area with radius 5: " + CircleArea(5));
        Console.WriteLine("circumference with radius 5:" + Circumference(5));
    }
}
```

See that PI is used in two functions without being declared locally.

By convention, names for constants are all capital letters (and underscores joining multiple words).

3.9 Not using Return Values

Some functions return a value, and get used as an expression in a larger calling statement. The calling statement uses the value returned. Usually the only effect of such a value-returning function is from the value returned.

Some functions are void, and get used as a whole instruction in your code: With-

out returning a value, the only way to be useful is to do something that leaves some lasting *side effect*: make some change to the system that persists after the termination of the function and its local variables disappear. The only such effect that we have seen so far is to print something that remains on the console screen. Later we will talk about other persistent changes to values in objects, locations in files, ...

Usually there is this division in the behavior of functions, returning a value or not:

1. `void`: do something as a whole instruction, with a side effect in the larger system.

2. Return a value to use in a larger calling statement

It is legal to do *both*: accomplish something with a side effect in the system, *and* return a value. Sometimes you care about both, and sometimes you use the function only for its side effect. We will see examples of that later, like in *Sets*.

This later advanced use will mean that the compiler needs to *permit* the programmer to ignore a returned value, and use a function returning a value as a whole statement.

> **Warning:** This means that the compiler cannot catch a common logical error: forgetting to immediately use a returned value that your program logic really needs.

For example with this definition:

```csharp
static int CalcResult(int param)
{
   int result;
   // ....
   result = ....;
   return result;
}
```

you might try to use `CalcResult` in this bad code, intending to use the `result` from CalcResult:

```csharp
static void BadUseResult(int x)
{
   int result = 0;
   CalcResult(x);
```

```
    Console.WriteLine(result);
}
```

In fact you would always print 0, ignoring the `result` calculated in `CalcResult`. The reason is the *Local Scope* rules: The local variable name `result` disappears when the `CalcResult` function returns. It is not used in the calling function, `BadUseResult`, and the separately declared `result` of `BadUseResult` retains its original 0 value.

Here we set up the worst situation: where there is a logical error, but not an error shown by the compiler. More commonly a student leaves out the `int result = 0;` line, incorrectly relying on the declaration of `result` in `CalcResult`. At least in that situation a compiler error brings attention to the problem: The last line would try to use the variable `result` without it being declared.

You can use the result from `CalcResult`, like with any other value-returning function, with either

```
int value = CalcResult(x);   //store the returned value in an assignment!
Console.WriteLine(value);    //  and then use the remembered value
```

or

```
Console.WriteLine(CalcResult(x)); // immediately use the returned value
```

This second version works as long as you do *not* need the returned value later, in another place, since you do not remember it past that one statement!

3.10 Library Classes

In *Returned Function Values*, the suggestion was made to look at the Painter class and split out repeated ideas into functions, leading to a function to prompt the user and return a double value. The same section included the example program addition2/addition2.cs. In that case there were repeated prompts for integers. Clearly another common situation is to prompt for a string. We can create functions to do all these things and more, and embed them into a class specially written for a new interactive program.

A neater thing is to put them as a class in a separate library that can be used directly for multiple programs. We can create functions `PromptLine`, `PromptInt`,

and `PromptDouble`, and put them in their own class, `UIF` (for User Input First
version) in project ui's file uif.cs. We explain the namespace line after the code:

```
using System;

namespace IntroCS
{
/// User Input First version (bombs in Parse with mistyping)
   public class UIF
   {
      /// After displaying the prompt, return a line from the keyboard.
      public static string PromptLine(string prompt)
      {
         Console.Write(prompt);
         return Console.ReadLine();
      }

      /// After displaying the prompt,
      /// return an integer entered from the keyboard.
      public static int PromptInt(string prompt)
      {
         return int.Parse(PromptLine(prompt));
      }

      /// After displaying the prompt,
      /// return a double entered from the keyboard.
      public static double PromptDouble(string prompt)
      {
         return double.Parse(PromptLine(prompt));
      }

      /// After displaying the prompt,
      /// return  true if 'y' is entered from the keyboard
      /// and false otherwise.
      public static bool Agree(string prompt)
      {
         return "y" == PromptLine(prompt);
      }
   }
}
```

We have been `using System` in every program. `System` is a *namespace* that

collects a particular group of class names, making them available to the program, and distinguishes them form any classes in a different namespace that might have the same class names.

Once we start writing and using multiple classes at once, it is a good idea for us to specify our own namespace. We will consistently use `IntroCS` in our multi-file examples in this book.

Specifying a namespace makes it possible for all other classes in the same namespace to reference *public* parts of the current class, and vice-versa.

Public classes and functions start their heading with `public`.

The code included in a namespace is enclosed in braces, so the general syntax is

```
namespace name
{
        class definition(s)...
}
```

We will keep user input library classes like this one, uif.cs, in a project ui.

Notice that the functions we want accessible in `UIF` are all marked `public`, so that any class can use them.

We can write a modified example addition program, addition3/addition3.cs, as an example of using `UIF`:

```csharp
using System;
namespace IntroCS
{
   class Addition3 // using UIF
   {
      /// Return a sentence stating the sum of x and y.
      static string SumProblemString(int x, int y)
      {
         int sum = x + y;
         string sentence = "The sum of " + x + " and " + y +
                           " is " + sum + ".";
         return sentence;
      }

      public static void Main()
      {
```

```
        Console.WriteLine(SumProblemString(2, 3));
        Console.WriteLine(SumProblemString(12345, 53579));
        int a = UIF.PromptInt("Enter an integer: ");
        int b = UIF.PromptInt("Enter another integer: ");
        Console.WriteLine(SumProblemString(a, b));
    }
  }
}
```

To allow access to UIF, we have added the IntroCS namespace for the class. To reference the static functions in the different class UIF, we put UIF. (with the dot) at the start of each reference to a static function in the class UIF.

> **Warning:** In Xamarin Studio, if you use a file from a library project (without just copying the present version of that file into the current project), be sure that the current project includes a *reference* to the library project. If you expand the references in the Xamarin Studio project addition3, by clicking on the References line in the solution pad, you should see the project ui.

Shortly you will see the optional section for making your own *Library Projects in Xamarin Studio (Optional)*.

Though we have not discussed all the C# syntax needed yet, there is also an improved class UI in the ui project that we discuss later. It includes all the function names in UIF, and keeps your program from bombing out if the user enters an illegal format for a number.

3.10.1 Function Documentation

In keeping with *Two Roles: Writer and Consumer of Functions*, in future you will be a *consumer* of the library classes. It is particularly important to document library classes with the interface information users will need. Documentation could be written in a separate document, but much developer history has shown that such documentation does not tend to either get written in the first place, or not updated well to stay consistent with updates in the code. Inconsistent documentation is useless. Documentation is much more likely to be seen and maintained by the implementers if it sits right with the code, like our comments before the class and function headings.

You will note that instead of the usual line comment syntax `//`, we have added an extra `/`, making `///`. That will also start a comment. (The third `/` is technically just a part of the comment.) There is a special reason for the notation: Though it is convenient for the *implementer* of code to have the documentation right with the code, a *user* of the functions only needs the interface information found in good documentation. The `///` lines before heading are specially recognized by *separate* automatic documentation generating programs.

There are many documentation generating programs and conventions. For now we will just use plain text in the `///` lines. This is recognized by the Xamarin Studio system. If you open our `examples` solution, in Xamarin Studio, and edit window for addition3/addition3.cs, you can place your mouse over `UIF` and a popup window shows the `UIF` class heading documentation.

If you move the mouse over `PromptInt`, you should see the popup label showing the function signature and the function documentation. If you change the two `///` lines in `uif.cs` above the `PromptInt` heading to start with just `//`. you should no longer be able to see the documentation part of the popup for `PromptInt` in the `addition3.cs` edit window. (Be sure to change back to `///`.)

There are more elaborate documentation conventions that can be used for Xamarin Studio and other documentation generation programs, not discussed here.

This documentation also works inside a single program file. If you have a long program with lots of functions defined, this can also be helpful when calling one of your own functions. You can avoid jumping around to be reminded of the signature and use of your functions.

3.10.2 Library Projects in Xamarin Studio (Optional)

Xamarin Studio has a multi-step process for creating a library project and for separately referencing it in other projects. The *advantage* of this approach is when you want to change the implementation but not the interface to library functions, you just do it once, in the library project. Other projects reference that project.

Some students find the Xamarin Studion overhead of setting up and referencing library projects onerous. As a practical matter with files that you want to reuse but are not likely to change, you can just copy the source file into the new project, and avoid the Xamarin Studio library setup overhead. Many of our already created

example projects use a library version of UIF and several other utility files. You can do the same with your solutions, following the instructions below, or you can just copy in the needed utility files for each project.

Hence the rest of the section here is *optional*:

Try adding a reference yourself. Follow these instructions:

1. In your own Xamarin Studio solution, start to add a project, but *instead* of leaving Console Project selected in the dialog window, select **Library Project**.

2. Then add the project name ui, and continue like when starting previous projects.

3. Copy in the `.cs` files from our ui project, uif.cs and ui.cs. Now you have your library project.

4. Create another regular Console project, addition3, in your *same* solution, and copy in our addition3/addition3.cs, so that is the only file.

> **Warning:** Xamarin Studio remembers the last kind of project you created. That is fine when you are creating a sequence of Console projects. However, if you have just explicitly chosen to create a library project, the default for your next project will also be library, and really mess up your next Console project. Fix such an error after the fact as in the first entry in *Xamarin Studio Reminders and Fixes*.

5. In the Solutions pad, in your addition3 project, click on the References entry just inside the project. You should see that the project is automatically set up to reference System.

6. Open the local menu for the References, and select Edit References.

7. Click the Projects tab in the window that pops up. This limits the length of the list that you search.

8. Possibly after scrolling down, find the recently made ui project and check the box beside it.

9. Click OK in the bottom right corner of the window. Now look at the References again. You should see ui listed!

10. Run your addition3 project.

You only need to add a library project once, but every further project that needs it, must have a *reference* to the library project added. You might try another for yourself with the next exercise!

Again this approach allows you to change the implementation of your library class in just one copy in one project, which can be referenced from many places. If you copy the file into different projects, and then decide the code needs to be updated, you are stuck *finding* and editing *all* the copies! Not good. Our library files uif.cs, ui.cs, and later fio.cs, should not be moving targets, so copying should not cause a problem. This may simplify your life, but the tradeoff is not getting used to using library references, which are useful in the larger scheme of things.

3.10.3 Quotient UI Exercise

Create `quotient_u_i.cs` by modifying `quotient_return.cs` in *Quotient String Return Exercise* so that the program accomplishes the same thing, but use the UIF class for all user input.

3.11 Static Function Summary

This chapter has introduced static functions: those used in procedural programming as opposed to *Instance Methods* used to implement object-oriented programming.

References in square brackets link to fully discussions of summary items below.

3.11.1 Function definition

1. The general syntax for defining a static function is

   `static` **returnTypeOrVoid FunctionName** (formal parameter list)
   {
        statements in the function body...
   }

2. The *formal parameter list* can be empty or contain one or more comma separated *formal parameter* entries. [*Function Parameters*] Each formal parameter entry has the form

   type parameterName

3. If the function is going to be called from outside its class, the heading needs to start with `public` before the `static`. [*Library Classes*]

4. If **returnTypeOrVoid** in the heading is not `void`, there must be a *return statement* in the function body. A return statement has the form

   `return` *expression* `;`

   where the expression should be of the same type as in **returnTypeOrVoid**. Execution of the function terminates immediately when a return statement is reached. [*Returned Function Values*]

5. Execution of a program starts at a function with a heading including

   `static void Main`

   Thus far we have only discussed having an empty parameter list in the heading of the definition of `Main`, and we defer discussion of *Parameters to Main* until we have introduced *One Dimensional Arrays*.

6. There are various conventions for putting documentation just above the headings of function definitions. The official format, specified by C# and recognized by Xamarin Studio, involves putting the function interface description on consecutive lines starting with `///`. [*Function Documentation*]

3.11.2 Function Calls

1. A function call takes the form

   FunctionName (actual parameter list)

   A function call makes the function definition be *executed*.

2. The actual parameter list is a comma separated list of the *same* length as the formal parameter list. Each entry is an expression. The entries in an actual parameter list do *not* include type declarations.

Effectively, the function execution starts by assigning to each formal parameter variable the corresponding value from evaluating the actual parameter expression. In particular, that means the actual parameter values must be allowed in an assignment statement for a variable of the formal parameter's type! [*Multiple Function Parameters*]

3. If the function has return type `void`, it can only be used syntactically as an entire statement (with a semicolon added). After the function call completes, execution continues with the next statement.

4. If there is a non-void return type, then the function call is syntactically an expression in the statement where is appears. The execution of such a function must reach a return statement. The value of the function-call expression is the value of the expression in this return statement. [*Returned Function Values*]

5. A function with a return value can also legally be used as a whole statement. In this case the return value is lost. Though legal, this is often an error! [*Not using Return Values*]

3.11.3 Scope

1. A variable declared inside a function definition is called a *local variable*. This declaration may be in either the formal parameter list or in the body of the function. [*Local Scope*]

2. A local variable comes into existence after the function is called, and ceases to exist after that function call terminates. A local variable is invisible to the rest of the program. Its *scope* is just within that function. Its lifetime is just through a single function call. Its *value* may be transferred outside of the function scope by standard means, principally:

   • If it is the expression in a return statement, its value is sent back to the caller.

   • It can be passed as an actual parameter to a further function called within its scope.

   [*Local Scope*]

3.11.4 Static Variables

1. There may be a declaration prefaced by the word `static` that appears *inside* a class and *outside* of any function definition in the class. Static variables are visible within each function of the class, and may be used by the functions. [*Static Variables*]

2. A common use of a static variable is to give a name to a constant value used in multiple functions in the class. [*Static Variables*]

3.12 Chapter Review Questions

1. Write the function definition heading for a static function called `Q1` which has two `int` parameters, `x` and `y`, and returns a `double`.

2. The function above must have what kind of a statement in its body?

3. Each of these lines has a call to the function above, `Q1`. Which are legal? Explain:

```
double d = Q1(2, 5);

int x = Q1(2, 5);

double y = Q1(2) + 5.5;

Console.WriteLine(Q1("2", "5"));

Console.WriteLine(Q1(2.5, 5.5));

Q1(10, 20);
```

4. Suppose `Q1` does nothing except produce the value to return, like most functions returning a `double`. Which line in the previous problem is legal, but has *no* effect?

5. Write the function definition heading for a static function called `Q4` which has one `string` parameter, `s`, and returns nothing.

6. Which of these lines with a call to the function above, `Q4`, is legal? Explain:

```
Q4("hi");

string t = Q4("hi");

Console.WriteLine(Q4("hi"));

Q4("hi" + "ho");

Q4("hi", "ho");

Q4(2);
```

7. Can you have more than one function/method in the same class definition with the same name?

8. What is a function/method signature? Can you have more than one function/method declared in the same class definition with the same signature?

9. In each part, is this a legal program? If so, what is printed? If not, why not?

   Each version uses the same code, except for different versions of `Main`. Here is the common code with the body of `Main` omitted:

```
using System;
class Local1
{
   static int Q(int a)   // 1
   {                      // 2
      int x = 3;          // 3
      x = x + a;          // 4
      return x;           // 5
   }                      // 6

   static void Main()
   {
      // see each version
   }
}
```

   (a) Insert:

```
      static void Main()
      {
```

```
        Q(5);
        Console.WriteLine(x);
    }
```

(b) Insert instead:

```
    static void Main()
    {
        int x= 1;
        Q(5);
        Console.WriteLine(x);
    }
```

(c) Insert instead:

```
    static void Main()                    // 7
    {                                     // 8
        int x = 1, y = 2;                 // 9
        y = Q(5);                         // 10
        Console.WriteLine(x + " " + y);   // 11
    }                                     // 12
```

10. In the previous problem consider the common code with part c. Note the line numbers as comments.

    (a) In what line(s) is Q being defined?

    (b) In what line(s) is Q called?

    (c) What is the return type of Q?

    (d) What is a formal parameter to Q?

    (e) What is used as an actual parameter to Q?

    (f) What is the scope of the x in line 3?

    (g) What is the scope of the x in line 9?

BASIC STRING OPERATIONS

4.1 String Indexing

Strings are composed of characters. In literals be careful of the different kinds of quotes: single for individual characters for type char and double for strings of 0 or more characters. For example, 'u' (single quotes) is a char type literal, while "u" is a string literal, referencing a string object. While "you" is a legal string literal, 'you' generates a compiler error (too many characters for a char literal).

Many of the operations on strings depend upon referring to the positions of characters in the string. A position is given by a numerical *index* number. In C#, positions are counted *starting at 0*, not 1. The indices of the characters in the string "coding" are labeled:

Index	0	1	2	3	4	5
Character	c	o	d	i	n	g

There are 6 characters in "coding", while the last index is 5.

> **Warning:** Because the indices start at 0, not 1, the index of the last character is one less that the length of the string. This is a common source of errors!

You can easily create an expression that refers to an individual character inside a string. Use square braces around the index of the character:

```
csharp> string s = "coding";
csharp> s[2];
```

```
'd'
csharp> s[0];
'c'
csharp> s[5];
'g'
csharp> string greeting = "Bonjour";
csharp> greeting[1];
'o'
```

Note from the single quotes that the result is a char in each case.

C# does not allow the typography for normal mathematical subscripts, like s_2. There is a correspondence with index notation, so s[2] is sometimes spoken as "s sub 2". The indices are sometimes referred to as *subscripts*.

In this introduction, we have used literal integers for the subscripts. The most common situation in practice is to have a variable or a more complicated expression as the subscript. An expression inside square braces is always evaluated to find the resulting index:

```
csharp> string s = "coding";
csharp> int n = 3;
csharp> s[n-1];
'd'
```

When we get to loops, we will find this is useful.

4.1.1 Indexing Exercise

What is printed by this fragment?

```
string str = "fragment";
int k = 3;
Console.WriteLine(str[1]);
Console.WriteLine(str[k]);
Console.WriteLine(str[2*k - 2]);
```

Write an expression that would give you the n in str, above.

Play with csharp: declare other strings and int variables, and make up string indexing expressions for which you predict the value and then test.

4.2 Some Instance Methods and the Length Property

Strings are a special type in C#. We have used string literals as parameters to functions and we have used the special concatenation operator +. Thus far we have not emphasized the use of objects, or even noted what is an object. In fact strings are objects. Like other objects, strings have a general notation for functions that are specially tied to the particular type of object. These functions are called *instance methods*. They always act on an object of the particular class, but a reference to the object is not placed inside the parameter list, but *before* the method name and a dot as in:

```
csharp> string s = "hello";
csharp> s.ToUpper();
"HELLO"
```

ToUpper (the method converting to upper case) does a particular action that makes sense with strings. It takes s (the string object reference before the dot in this example) and returns a *new* string in upper case, based on s. Since this action depends only on the string itself, no further parameters are necessary, and the parentheses after the method name are empty. The general method syntax is

> *object-reference* . **methodName** (*further-parameters*)

More string methods are listed below, some with further parameters.

Data can also be associated with object *properties*. A property of a string is its Length (an int). References to property values use dot notation but do not have a parameter list in parentheses at the end:

```
csharp> string s = "Hello";
csharp> s.Length;
5
csharp> "".Length;
0
```

Be careful: Though 5 is the length of s in the example above, the last character in s is s[4]. Using s[5] would generate an IndexOutOfRangeException.

String objects have associated string methods which can be used to manipulate string values. There are an enormous number of string methods, but here are just

---

a few of the most common ones to get you started. The string object to which the method is being applied is referred to as **this** string in the descriptions. After the methods, the length property is also listed. In the heading *this* object is not shown explicitly, so be careful when applying these methods and the length property: In actual use in your programs they must be preceded by a reference to a string, followed by a dot, as shown in all the examples. The reference to *this* string can be a variable name, a literal, or any expression evaluating to a string.

4.2.1 Summary of String Length and Some Instance Methods

int IndexOf(string target) Returns the index of the beginning of the first occurrence of the string `target` in **this** string object. Returns -1 if `target` not found. Examples:

```csharp
csharp> string greeting = "Bonjour", part = "jo";
csharp> greeting.IndexOf(part);
3
csharp> greeting.IndexOf("jot");
-1
```

string Substring(int start) Returns the substring of **this** string object starting from index `start` through to the end of the string object. Example:

```csharp
csharp> string name = "Sheryl Crow";
csharp> name.Substring(7);
"Crow"
```

string Substring(int start, int len) Returns the substring of **this** string object starting from index `start`, including a total of `len` characters. Example:

```csharp
csharp> string name = "Sheryl Crow";
csharp> name.Substring(3,5);
"ryl C"
```

Java programmers: Note the second parameter is *not* the same as in Java.

string ToUpper() Return a string like **this** string, except all in upper case. Example:

```
csharp> "Hi Jane!".ToUpper();
"HI JANE!"
```

string ToLower() Return a string like **this** string, except all in lower case. Example:

```
csharp> "Hi Jane!".ToLower();
"hi jane!"
```

int Length Property referring to the length of **this** string object. Example:

```
csharp> string greeting = "Bonjour";
csharp> greeting.Length;  //no parentheses
7
```

> **Warning:** All of these methods that return a string return a *new* string. No string method alters the original string. Strings are *immutable*: They are objects that cannot be changed after they are first produced. This is a common source of errors.

```
csharp> string s = "Hello";
csharp> s.ToUpper()
"HELLO"
csharp> s
"Hello"
csharp> s = s.ToUpper();
csharp> s
"HELLO"
```

See that you need an explicit assignment if you *want* the variable associated with the original string to change.

Further string methods are introduced in *More String Methods*.

Time to reflect, thinking back to *Learning to Solve Problems*. Without forcing all the code details on yourself, how can you concisely say what powers you have with strings so far? Remember that kernel.

With strings you can: Index characters, find a part; extract a part; convert case; determine length. These may not be evocative phrases for you. Find your own.

When we get to loops, we will find this is useful.

4.2. Some Instance Methods and the Length Property   **127**

Here is a brief example of a function using several of these methods,

parenthesized/parenthesized.cs:

```
 1   using System;
 2
 3   class Parenthesized
 4   {
 5      static void Main()
 6      {
 7         string s = "What (really) happened?";
 8         Console.WriteLine ("Original: " + s);
 9         Console.WriteLine ("In parentheses: " + InsideParen(s));
10      }
11
12      /// Assume s contains parentheses.
13      /// Return the substring between the first parentheses.
14      static string InsideParen(string s)
15      {
16         int first = s.IndexOf('(') + 1;
17         int past = s.IndexOf(')');
18         int len = past - first;
19         return s.Substring(first, len);
20      }
21   }
```

It is a silly assumption, but until we get to *Decisions*, we will have to assume there
is a parenthesized expression in the parameter string.

String Methods Exercise

1. What is printed by this fragment?

```
string w = "quickly";
Console.WriteLine(w.Length);
Console.WriteLine(w[w.Length-2]);
Console.WriteLine(w.Substring(3, 2));
Console.WriteLine(w.Substring(2));
Console.WriteLine(w.IndexOf("ti"));
Console.WriteLine(w.IndexOf("ick"));
int k = w.IndexOf("c");
```

```
   Console.WriteLine("{0} {1} {2} {3}",
      k, w[k], w[k-3], w.Substring(k));
```

2. What is printed by this fragment?

```
   string s = "HELLO!", t = s.ToLower();
   Console.WriteLine(s+t);
```

Play with csharp: Declare other strings and make up string expressions with these methods for which you predict the value and then test.

4.3 A Creative Problem Solution

Thus far the exercises and examples suggested have been of a very simple form, where the idea of the steps should have been pretty clear, and the main issue was just translating syntax into C#, one instruction at a time.

We still have a lot of syntax to concentrate on, but still, early on, we wanted to get in some real thought of problem solving. To get very interesting you need a number of options that might be combined in a variety of ways. The short list of string methods just introduced is likely give us enough to think about....

Here is a basic string manipulation problem: given a string, like, `"It was the best of times."`, find and replace a specified part of it by another string. For instance replace `"best"` by `"worst"`. In this example we would get the result: `"It was the worst of times."`.

It is very important to give concrete examples to illustrate the idea desired. Our human brains may be very quick to see a solution like this in a very concrete case, but what about making it general?

First this seems like a basic logical operation worthy of a function or method, so we need a heading. (Confession: there are methods in the class string for replacement, but this is a good learning exercise, so we are starting over on our own.) Since we cannot change the string class, we will write a static function to generate the new string.

For simplicity at the moment we will only change the first occurrence, and for now we will assume the replacement makes sense. The following heading (with documentation) should work:

```
/// Return s with the first occurence of target
/// replaced by replacement.
static string replaceFirst(string s, string target,
                           string replacement)
```

As soon as we have the calling interface, it is good to be thinking of the tests it should pass. Here is a Main program written to test the function in different ways and display the results:

```
static void Main ()
{
   string str1 = "It was the best of times.";
   string str2 = "Of times it was the best.";
   Console.WriteLine("str1=" + str1);
   Console.WriteLine("str2=" + str2);
   Console.WriteLine();
   // to embed a quote inside a string constant, precede it by backslash(\).
   Console.WriteLine("Let us do some \"cutting and pasting\" of strings!");
   string str3 = replaceFirst(str1, "best", "worst");
   Console.WriteLine("str3 = str1 with best => worst: " + str3);
   string str4 = replaceFirst(str2, "best", "worst");
   Console.WriteLine("str2 with best => worst: " + str4);
   string str5 = replaceFirst(str3, "worst", "best");
   Console.WriteLine("str3 with worst => best: " + str5);
}
```

Writing tests *first* is a good idea to focus you on what really needs to be accomplished, and then running tests later is a snap!

The human brain and eyes are fabulous in the way they process many things in parallel and use tools you have accumulated over a lifetime. In particular this substitution idea should seem pretty reasonable, and given any *specific concrete* example, you are likely to be able to solve it instantly, with very little conscious effort. Once it becomes a programming problem, with parameters stated in general, with just placeholder names like s and target, and given the limited set of approaches you have in a programming language, then the complexion of this problem changes completely. Many students guess the general problem will be nearly as simple as the concrete examples they do in their heads, and then get very discouraged when the answer does not flow out of them. In fact it takes practice and experience, and it is easier to handle if you acknowledge that up front!

So let's start in with the practice, and gain some experience. With s, target,

and `replacement` all being general, this problem could easily be too much to contemplate at once, so let us replace *concrete* examples by generality gradually. The idea is to get to the end. Rather than trying to jump a chasm, we can take small steps and go around.

A basic idea is to make small incremental changes, test at each stage, and gradually see more of the tests (that you have already written) be satisfied. Also, if you make a mistake and screw up something that worked before, you can generally focus on the small addition to see where the mistakes were. [1]

This also avoids you needing to keep too much in your head at once.

We do have code written already: The test code. Start by writing something that will trivially satisfy the first concrete test. The body of the function can be just:

```
return "It was the worst of times";
```

This is a tiny, easy, silly looking step, but it does accomplish two things: It makes sure we can produce output in the proper string form, and the test code runs, passing the first test.

Now we gradually get more complicated. We will continue to assume `target` and `replacement` are as in the original example, and `target` is in the same place in `s`, but suppose we imagine each of the other characters in `s` may be something different:

```
"??????????best??????????"
```

Now we have to start thinking about what we have to work with. We have a string, and we have string methods. Have a look at the ideas of each method (exact syntax not important at the moment). Clearly we are going to have to deal with parts of strings, and the methods to deal with parts involve indices, so let us add to our visual model:

```
Index: 012345678901234567890234
    s: ??????????best??????????
```

Continue in class.... The example program stub is string_manip_stub/string_manip.cs.

---

[1] We will not go far into the history of software engineering practice here, but these incremental problem solving methods were first widely introduced as a part of *extreme programming*. That name gives you an idea of the newness at the time.

In general, when given a project with "stub" in it, you should copy the files into a project of your own and make modifications. Though the original version should compile and run, it does not do much without your additions. In stubs where you need to complete a function with a return value, you will often see a dummy choice for the return statement, just so the stub compiles. Where the return type is string `"Not implemented"` is a handy temporary choice.

When you have that function version, test it. You will need to rename our incremental variations so the current version has the name used in Main.

What might further advances toward full generality be, in small steps? We pinned `best` at a specific location. We could remove that assumption. The location will still be important, but we do not know it ahead of time....

A further advance would be a version that is complete in all ways, except we still assume `target` is in `s`, but beyond that, do not assume what the three parameters are.

Finally we should allow `s` to not contain `target` (though this requires the central idea of the next chapter).

The testing regime in `Main` is clear to understand and write, but pretty primitive. You have to look at a lot of output every time you test. We will come up with better testing schemes later.

4.4 Lab: String Operations

4.4.1 Goals for this lab:

1. Explore some of the properties of the pre-defined String class.

2. Write conditional statements.

3. Think about problem solving.

This lab depends on the introductory material earlier in this chapter, particularly keep handy *Summary of String Length and Some Instance Methods*. Be mindful of the processes developed in class filling in *A Creative Problem Solution*.

Parts 2 and 4 also depend on *Decisions* through *More Conditional Expressions*.

Design, compile and run a single C# program to accomplish all of the following tasks. Add one part at a time and test before trying the next one. The program can just include a Main method, or it is neater to split things into separate methods (all static void, with names like ShowLength, SentenceType, LastFirst1, LastFirst), and have Main call all the ones you have written so far (or for testing purposes, just the one you are working on, with the other function calls commented out). **Use the UIF class for user input.** You can copy the file into your project.

Alternately practice using an Xamarin Studio library reference: Create a new project in a solution in which you already have added the ui library project. Make the ui project be a reference for the lab project.

Make sure your program has `namespace IntroCS;` to match the ui project. Beware of modifying the sample Program.cs generated by Xamarin Studio - it will use the project name for a namespace. We are never going to use that.

1. Read a string from the keyboard and print the length of the string, with a label.

2. Read a sentence (string) from a line of input, and print whether it represents a *declarative* sentence (i.e. ending in a period), *interrogatory* sentence (ending in a question mark), or an *exclamation* (ending in exclamation point) or is not a sentence (anything else).

   This may be the first time you write a conditional statement. (This needs the next chapter.) It makes sense to only make small changes at once and build up to final code. First you might just code it to check if a sentence is declarative or not. Then remember you can test further cases with `else if (...)`.

3. Read a whole name from a line of input. Assume first and last names are separated by a space (no middle name). Print last name first followed by a comma and a space, followed by the first name. For example, if the input is `"Marcel Proust"`, the output is `"Proust, Marcel"`.

4. Improve the previous part, so it also allows a single name without spaces, like "Socrates", and prints the original without change. If there are two parts of the name, it should work as in the original version. (This needs the next chapter.)

Run the program (with parts 1, 2 and 4 active) from a terminal window and show your TA when you are done. You should run it twice to show off both paths through part 4. Alternately have the main program just call part 4 twice!

4.5 Chapter Review Questions

1. What is printed by this fragment?

```csharp
string s = "question";
Console.WriteLine(s.Length);
Console.WriteLine(s[2]);
Console.WriteLine(s.Substring(2, 3));
Console.WriteLine(s.Substring(3));
Console.WriteLine(s.IndexOf("ti"));
Console.WriteLine(s.IndexOf("to"));
int j = s.IndexOf("u"), k = s.IndexOf("o");
Console.WriteLine("{0} {1} {2}", j, k, s.Substring(j, k-j));
```

2. What is printed by this fragment?

```csharp
string s = "Word";
s.ToUpper();
Console.WriteLine(s);
```

3. What is printed by this fragment?

```csharp
string a = "hi", b = a.ToUpper();
Console.WriteLine(a+b);
```

4. Are strings mutable or immutable: which?

5. What is the distinction syntactically between the use of a method and a property?

6. Suppose we have a string s. Is this expression legal, or what should it be?

```csharp
s.Length()
```

DECISIONS

5.1 Conditions I

Thus far, within a given function, instructions have been executed sequentially, in the same order as written. Of course that is often appropriate! On the other hands if you are planning your instruction sequence, you can get to a place where you say, "Hm, that depends....", and a choice must be made. The simplest choices are two-way: do one thing if a condition is true, and another (possibly nothing) if the condition is not true.

More syntax for conditions will be introduced later, but for now consider simple arithmetic comparisons that directly translate from math into C#. First start csharp an enter:

```
int x = 11;
```

Now think of which of these expressions below are true and which false, and then enter each one into your csharp session to test:

```
2 < 5
3 > 7
x > 10
2*x < x
```

You see the C# values, either `true` or `false` (with no quotes!). These are the only possible *Boolean* values (named after 19th century mathematician George Boole). You can also use the abbreviation for the type `bool`. It is the type of the results of true-false conditions or tests.

The simplest place to use conditions is in a decision made with an `if` statement.

We will consider *More Conditional Expressions* later, but this is a quick start with the easiest ones.

5.2 Simple `if` Statements

Run the example program, suitcase/suitcase.cs. Try it at least twice, with inputs: 30 and then 55. As you can see, you get an extra result, depending on the input. The main code is:

```
double weight =
    UIF.PromptDouble("How many pounds does your suitcase weigh? ");
if (weight > 50) {
    Console.WriteLine("There is a $25 charge for luggage that heavy.");
}
Console.WriteLine("Thank you for your business.");
```

The lines labeled 3-5 are an `if` statement. It reads pretty much like English. If it is true that the weight is greater than 50, then print the statement about an extra charge. If it is not true that the weight is greater than 50, then skip the part right after the condition about printing the extra luggage charge. In any event, when you have finished with the `if` statement (whether it actually does anything or not), go on to the next statement. In this case that is the statement printing "Thank you". An `if` statement only breaks the normal sequential order *inside* the `if` statement itself.

The general C# syntax for a simple `if` statement is

> if (*condition*)
>   statement

The condition is an expression that is true or false, of *Type Boolean or bool*.

Often you want multiple statements executed when the condition is true. We have used braces before. We have not said what they do technically, syntactically: braces around a group of statements technically makes a single *compound statement*. So the pattern commonly written is:

> if (*condition*) {
>   one or more statements

```
    }
```

If the condition is true, then do the statement(s) in braces. If the condition is not true, then skip the statements in braces. The indentation pattern is also illustrated. Recall the compiler does not care about the amount of whitespace, but humans do. In general indent the statements inside a compound statement. Later in:ref:*missing-braces* we will see that there is good reason to use this format with braces *even* if there is just one statement inside the braces.

Another fragment as an example:

```
if (balance < 0) {
    transfer = -balance;
    // transfer enough from the backup account:
    backupAccount = backupAccount - transfer;
    balance = balance + transfer;
}
```

The assumption in the example above is that if an account goes negative, it is brought back to 0 by transferring money from a backup account in *several* steps.

In the examples above the choice is between doing something (if the condition is true) or nothing (if the condition is false). Often there is a choice of two possibilities, only one of which will be done, depending on the truth of a condition....

5.2.1 Simple If Exercise

Think of two different inputs you could give that would make the execution of the code fragment proceed differently. What would happen in each case? (Assume we have access to the class UIF.)

1. Consider:

```
string v = UIF.PromptLine("Enter a word: ");
if (v.Length > 3) {
    v = v + v;
}
Console.WriteLine("Now we have " + v);
```

2. Consider:

```
    int x = UIF.PromptInt("Enter a integer: ");
    Console.Write("The magnitude of " + x + " is ");
    if (x < 0) {
        x = -x;
    }
    Console.WriteLine(x);
```

5.3 `if-else` Statements

Run the example program, clothes/clothes.cs. Try it at least twice, with inputs 50 and then 80. As you can see, you get different results, depending on the input. The main code of clothes/clothes.cs is:

```
1  double temperature = UIF.PromptDouble("What is the temperature? ");
2  if (temperature > 70) {
3      Console.WriteLine("Wear shorts.");
4  }
5  else {
6      Console.WriteLine("Wear long pants.");
7  }
8  Console.WriteLine("Get some exercise outside.");
```

The lines labeled 2-7 are an `if-else` statement. Again it is close to English, though you might say "otherwise" instead of "else" (but else is shorter!). There are two indented statements in braces: One, like in the simple `if` statement, comes right after the `if` condition and is executed when the condition is true. In the `if-else` form this is followed by an `else` (lined up under the `if` by convention), followed by another indented statement enclosed in braces that is only executed when the original condition is *false*. In an `if-else` statement exactly one of two possible parts in braces is executed.

A final line is also shown that is not indented, about getting exercise. The `if` and `else` clauses each only embed a *single* (possibly compound) statement as option, so the last statement is not part of the `if-else` statement. It is beyond the `if-else` statement; it is just a part of the normal sequential flow of statements. We repeat:

Note: Inside an `if-else` there is a choice made of which clause to execute

and which clause to skip, but the whole `if-else` construction is a *single* larger statement, which exists in the normal *sequential* flow of execution.

The compiler does not require the indentation of the if-true-statement and the if-false-statement, but it is a standard style convention.

The general C# `if-else` syntax is

```
if ( condition ) {
      statement(s) for if-true
}
else {
      statement(s) for if-false
}
```

The statements chosen based on the condition can be any kind of statement. This is the suggested form, but as with the plain `if` statement, the if-true compound statement or the if-false compound statement can be replace by a single statement without braces, except in one otherwise ambiguous situation later, *Match Wrong if With else*.

5.3.1 Scope With Compound Statements

The section *Local Scope* referred to function bodies, which happen to be enclosed in braces, making the function body a *compound statement*. In fact variables declared inside *any* compound statement have their scope restricted to *inside* that compound statement.

As a result the following code makes no sense:

```csharp
static int BadBlockScope(int x)
{
   if ( x < 100) {
      int val = x + 2;
   }
   else {
      int val = x - 2:
   }
   return val;
}
```

The `if-else` statement is legal, but useless, because whichever compound statement gets executed, `val` ceases being defined after the closing brace of its compound statement, so the `val` in the return statement has not been declared or given a value. The code would generate a compiler error.

If we want `val` be used inside the braces and to make sense past the end of the compound statement, it cannot be declared inside the braces. Instead it must be declared before the compound statements that are parts of the `if-else` statement. A local variable in a function declared before a nested compound statement is still visible (in scope) *inside* that compound statement. The following would work:

```csharp
static int OkScope(int x)
{
   int val;
   if (x < 100) {
      val = x + 2;
   }
   else {
      val = x - 2;
   }
   return val;
}
```

There is even more subtlety here than meets the eye: An `if-else` statement can generally be rewritten as two simple if statements (though it is less efficient and less clear). The two `if` statements would use opposite conditions, as in this variation:

```csharp
static int OkScope2(int x)
{
   // without the = 0: Unassigned local variable error
   int val = 0;
   if (x < 100) {
      val = x + 2;
   }
   if (x >= 100) {
      val = x - 2;
   }
   return val;
}
```

Note that in this variation we added an initialization for `val` to be 0, though the

value of the initialization is never used: `val` is guaranteed to be assigned a value in one of the `if` statements before its value is used in the return statement.

Open Xamarin Studio with the examples solution, and open ok_if_scope/ok_if_scope.cs in the edit window. The last function, `OkScope2`, is the one shown above. Now *remove* the logically unnecessary `= 0` initialization for `val` so the line is just `int val;`. As the comment says, an error should appear (at least after you try to compile the program). The error will say that there is an uninitialized local variable! Why?

For safety the C# compiler has some basic analysis to check that every local variable gets given a value before its value is used. In the `OkScope` function there is no *one* place where `val` gets an initial value, but the compiler is smart enough to see that one of the branches of any if-else statement is always taken, and `val` gets a value in each, so there is no problem.

The compiler analysis is not complete: It does not actually evaluate any expressions. This is good enough to catch many initialization errors that coders make, but it is not sufficient in general: We can see this from the altered `OkScope2`.

The original code shows the fix: Give a dummy initialization that is never used in execution, but keeps the compiler happy.

Although this extra initialization is annoying, the extra step is rarely needed. Meanwhile it is very easy to forget to give a value to a local variable before use! Having the error caught quickly by the compiler is very handy, offsetting the extra work when the compiler gives this error unnecessarily.

5.3.2 If-else Exercise

Think of two different inputs you could give that would make the execution of the code fragment proceed differently. What would happen in each case? (Assume we have access to the class UIF.)

1. Consider:

```
string v = UIF.PromptLine("Enter a word: ");
if (v.Length > 3) {
    v = v + v;
    Console.WriteLine("Now we have " + v);
}
```

```
    else {
        Console.WriteLine("We still have " + v);
    }
```

2. Consider:

```
    int x = UIF.PromptInt("Enter a integer: ");
    Console.Write("The magnitude of " + x + " is ");
    if (x < 0) {
        Console.WriteLine(-x);
    }
    else {
        Console.WriteLine(x);
    }
```

5.4 More Conditional Expressions

All the usual arithmetic comparisons may be made in C#, but many do not use standard mathematical symbolism, mostly for lack of proper keys on a standard keyboard. (If you are looking at the following table in the html version, you need to be online and may need an up-to-date browser to see these mathematical symbols be displayed correctly, as well as the mathematical expressions later in the text. The pdf version of the book automatically shows all the right math symbolism online or offline.

Meaning	Math Symbol	C# Symbols
Less than	$<$	$<$
Greater than	$>$	$>$
Less than or equal	$\leq$	$<=$
Greater than or equal	$\geq$	$>=$
Equals	$=$	$==$
Not equal	$\neq$	$!=$

There should not be space between the two-symbol C# substitutes.

Notice that the obvious choice for *equals*, a single equal sign, is *not* used to check for equality. An annoying second equal sign is required. This is because the single equal sign is already used for *assignment* in C#, so it is not available for tests.

> **Warning:** It is a common error to use only one equal sign when you mean to *test* for equality, and not make an assignment!

Tests for equality do not make an assignment. Tests for equality can have an arbitrary expression on the left, not just a variable.

All these tests work for numbers, and characters. Strings can also be compared, most often for equality (==) or inequality (!=), though they also have a defined order, so you can use <, for instance.

Predict the results and try each line in csharp:

```
int x = 5;
x;
x == 5;
x == 6;
x;
x != 6;
x = 6;
6 == x;
6 != x;
"hi" == "h" + "i";
"HI" != "hi";
string s = "Hello";
string t = "HELLO";
s == t;
s.ToUpper() == t;
```

An equality check does not make an assignment. Strings equality tests are case sensitive.

Try this: Following up on the discussion of the *inexactness* of float arithmetic, confirm that C# does not consider .1 + .2 to be equal to .3: Write a simple condition into csharp to test.

Pay with Overtime Example

Given a person's work hours for the week and regular hourly wage, calculate the total pay for the week, taking into account overtime. Hours worked over 40 are

overtime, paid at 1.5 times the normal rate. This is a natural place for a function enclosing the calculation.

Read the setup for the function:

```
/// Return the total weekly wages for a worker working
/// totalHours with a given regular hourlyWage.
/// Include overtime for hours over 40.
static double CalcWeeklyWages(double totalHours, double hourlyWage)
```

The problem clearly indicates two cases: when no more than 40 hours are worked or when more than 40 hours are worked. In case more than 40 hours are worked, it is convenient to introduce a variable overtimeHours. You are encouraged to think about a solution before going on and examining mine.

You can try running the complete example program, wages1/wages1.cs, also shown below.

```
1   using System;
2   namespace IntroCS
3   {
4     class Wages
5     {  //heading chunk
6        /// Return the total weekly wages for a worker working
7        /// totalHours with a given regular hourlyWage.
8        /// Include overtime for hours over 40.
9        static double CalcWeeklyWages(double totalHours, double hourlyWage)
10       {
11          double totalWages;
12          if (totalHours <= 40) {
13             totalWages = hourlyWage*totalHours;
14          }
15          else {
16             double overtime = totalHours - 40;
17             totalWages = hourlyWage*40 + (1.5*hourlyWage)*overtime;
18          }
19          return totalWages;
20       }
21
22       static void Main()
23       {
24          double hours = UIF.PromptDouble("Enter hours worked: ");
25          double wage = UIF.PromptDouble("Enter dollars paid per hour: ");
```

```
26              double total = CalcWeeklyWages(hours, wage);  //before chunk2
27              Console.WriteLine(
28                 "Wages for {0} hours at ${1:F2} per hour are ${2:F2}.",
29                 hours, wage, total);
30          }                                                    //after chunk2
31      }
32 }
```

This program also introduces new notation for displaying decimal numbers:

```
Console.WriteLine(
   "Wages for {0} hours at ${1:F2} per hour are ${2:F2}.",
   hours, wage, total);
```

In the format string are `{1:F2}` and `{2:F2}`: Inside the braces, after the parameter index, you see a new part, `:F2`. The part after the colon gives additional formatting information. In this case: Display with the decimal point fixed (hence the **F**) so **2** places beyond the decimal point are shown. Also the result is *rounded*. This is appropriate for money with dollars and cents. You can replace the 2 to display a different number of digits after the decimal point. More formatting syntax is introduced in *Tables*.

Because `if-else` statements alter the flow or execution, and altering this flow causes lots of problems for many students, this is a good place to play computer to illustrate clearly what is happening. We will just concentrate on a call to `CalcWeeklyWages`, where the `if-else` statement is.

Recall that to play computer, we keep track of the state of variables while following execution carefully, statement by statement. The big point here is that this order of execution is *not* textual order! If we are just following a call to `CalcWeeklyWages`, we start at line 9, and we will assume for example that it is called with `totalHour` as 50 and `hourlyWage` as 14.00:

Line	total-Wages	over-time	Comment
9	-	-	Assume passed totalHours=50; hourlyWage=14
12			50 <= 40: false; SKIP if clause
16		10	50-40=10 at start of else clause
17	770		14*40 + (1.5*14)*10 = 770
19			return 770 to caller

We *skipped* the executable line 13 in the if-true clause.

Instead suppose the function is called with `totalHour` as 30 and `hourlyWage` as 14.00:

Line	total-Wages	over-time	Comment
9	-	-	Assume passed totalHours=30; hourlyWage=14
12			30 <= 40: true; continue straight
13	420		14*30 = 420; SKIP else clause
19			return 420 to caller

We skipped the executable lines 16-17 in the if-false clause after `else`.

Below is an equivalent alternative version of `CalcWeeklyWages`, used in wages2/wages2.cs. It uses just one general calculation formula and sets the parameters for the formula in the `if` statement. There are generally a number of ways you might solve the same problem!

```
static double CalcWeeklyWages(double totalHours, double hourlyWage)
{
    double regularHours, overtime;
    if (totalHours <= 40) {
        regularHours = totalHours;
        overtime = 0;
    }
    else {
        regularHours = 40;
        overtime = totalHours - 40;
    }
    return hourlyWage*regularHours + (1.5*hourlyWage)*overtime;
}
```

5.4.1 Wages Play Computer Exercise

We played computer on the original version of `CalcWeeklyWages`. Now follow the second version above, from wages2/wages2.cs. Try it with the same two sets of formal parameter values.

Line	regu-larHours	over-time	Comment
1	-	-	Assume passed totalHours=50; hourlyWage=14

Line	regu-larHours	over-time	Comment
1	-	-	Assume passed totalHours=30; hourlyWage=14

5.4.2 Graduate Exercise

Write a program, `graduate.cs`, that prompts students for how many credits they have. Print whether of not they have enough credits for graduation. (At Loyola University Chicago 120 credits are needed for graduation.)

5.4.3 Roundoff Exercise

In csharp declare and initialize non-zero `double` variables x and y. Experiment so, according to C# (and csharp): `x+y == x`. In other words, while y is not 0, adding it to x does not change x. (Hints: Note the approximate number of digits of accuracy of a `double`, and remember the power of 10 notation with `E` for `double` literals. See *Type double* and the section on limits after it.)

5.5 Multiple Tests and `if-else` Statements

Often you want to distinguish between more than two distinct cases, but conditions only have two possible results, `true` or `false`, so the only direct choice is

between two options. As anyone who has played "20 Questions" knows, you can distinguish more cases with further questions. If there are more than two choices, a single test may only reduce the possibilities, but further tests can reduce the possibilities further and further. Since most any kind of statement can be placed in the sub-statements in an `if-else` statement, one choice is a further `if` or `if-else` statement. For instance consider a function to convert a numerical grade to a letter grade, 'A', 'B', 'C', 'D' or 'F', where the cutoffs for 'A', 'B', 'C', and 'D' are 90, 80, 70, and 60 respectively. One way to write the function would be to test for one grade at a time, and resolve all the remaining possibilities inside the next `else` clause. If we do this consistent with our indentation conventions so far:

```csharp
static char letterGrade(double score)
{
   char letter;
   if (score >= 90) {
      letter = 'A';
   }
   else {    // grade must be B, C, D or F
      if (score >= 80) {
         letter = 'B';
      }
      else { // grade must be C, D or F
         if (score >= 70) {
            letter = 'C';
         }
         else {    // grade must D or F
            if (score >= 60) {
               letter = 'D';
            }
            else {
               letter = 'F';
            }
         }   //end else D or F
      }      // end of else C, D, or F
   }         // end of else B, C, D or F
   return letter;
}
```

This repeatedly increasing indentation with an `if` statement in the `else` clause can be annoying and distracting. Here is a preferred alternative in this situation, that avoids all this further indentation: Combine each `else` and following `if`

onto the same line, and note that the `if` part after each else is just a *single* (possibly very complicated) statement. This allows the elimination of some of the braces:

```csharp
/// Return letter grade for score.
static char letterGrade(double score)
{
   char letter;
   if (score >= 90) {
      letter = 'A';
   }
   else if (score >= 80) { // grade must be B, C, D or F
      letter = 'B';
   }
   else if (score >= 70) { // grade must be C, D or F
      letter = 'C';
   }
   else if (score >= 60) { // grade must D or F
      letter = 'D';
   }
   else {
      letter = 'F';
   }
   return letter;
}
```

A program testing the letterGrade function is in example program grade1/grade1.cs.

See *Grade Exercise*.

As in a basic `if-else` statement, in the general format,

```
if ( condition1 )  {
      statement-block-run-if-condition1-is-true;
}
else if ( condition2 )  {
      statement-block-run-if-condition2-is-the-first-true;
}
else if ( condition3 )  {
      statement-block-run-if-condition3-is-the-first-true;
}
```

5.5. Multiple Tests and `if-else` Statements   **149**

```
    // ...
    else {  // no condition!
          statement-block-run-if-no condition-is-true;
    }
```

exactly one of the statement blocks gets executed: If some condition is true, the first block following a true condition is executed. If no condition is true, the `else` block is executed.

Here is a variation. Consider this fragment *without* a final `else`:

```
if (weight > 120) {
    Console.WriteLine("Sorry, we can not take a suitcase that heavy.");
}
else if (weight > 50) {
    Console.WriteLine("There is a $25 charge for luggage that heavy.");
}
```

This statement only prints one of two lines if there is a problem with the weight of the suitcase. Nothing is printed if there is not a problem.

If the final `else` clause is omitted from the general `if ... else if ...` pattern above, at most one block after a condition is executed: That is the block after the first true condition. If all the conditions are false, none of the statement blocks will be executed.

It is also possible to embed `if-else` statements inside other `if` or `if-else` statements in more complicated patterns.

5.5.1 Sign Exercise

Write a program `sign.cs` to ask the user for a number. Print out which category the number is in: `"positive"`, `"negative"`, or `"zero"`.

5.5.2 Grade Exercise

Copy [grade1/grade1.cs] to `grade2.cs` in your own project. Modify `grade2.cs` so it has an equivalent version of the letterGrade function that tests

in the opposite order, first for F, then D, C, ... Hint: How many tests do you need to do? [1]

Be sure to run your new version and test with different inputs that test all the different paths through the program.

Be careful for edge cases: Test the grades on the "edge" of a change in the result.

5.5.3 Wages Exercise

Modify the wages1/wages1.cs or the wages2/wages2.cs example to create a program `wages3.cs` that assumes people are paid double time for hours over 60. Hence they get paid for at most 20 hours overtime at 1.5 times the normal rate. For example, a person working 65 hours with a regular wage of $10 per hour would work at $10 per hour for 40 hours, at 1.5 * $10 for 20 hours of overtime, and 2 * $10 for 5 hours of double time, for a total of

$$10*40 + 1.5*10*20 + 2*10*5 = \$800.$$

You may find wages2/wages2.cs easier to adapt than wages1/wages1.cs.

Caution: Be sure to thoroughly test your *final* program version. It is easy to add new features that work by themselves, but break a part that worked before! In particular in a program with decisions, make sure you test with enough different data to check all lines of your program.

5.6 If-statement Pitfalls

5.6.1 Dangerous Semicolon

Regular statements must end with a semicolon. It turns out that the semicolon is all you need to have a legal statement:

```
;
```

We will see places that it is useful, but meanwhile it can cause errors: You may be hard pressed to remember to put semicolons at the end of all your statements,

---

[1] 4 tests to distinguish the 5 cases, as in the previous version

and in response you may get compulsive about adding them at the end of statement lines. Be careful NOT to put one at the end of a method heading or an `if` condition:

```
if ( x < 0); // WRONG PROBABLY!
    Console.WriteLine(x);
```

This code is deadly, since it compiles and is almost surely *not* what you mean.

Remember indentation and newlines are only significant for humans. The two lines above are equivalent to:

```
if ( x < 0)
    ;   // Do nothing as statement when the condition is true
Console.WriteLine(x); // past if statement - do it always
```

(Whenever you do need an empty statement, you are encouraged to put the semicolon all by itself on a line, as above.)

If you always put an open brace *directly* after the condition in an `if` statement, you will not make this error:

```
if ( x < 0) {
    Console.WriteLine(x);
}
```

Then even if you were to add a semicolon:

```
if ( x < 0) { ;
    Console.WriteLine(x);
}
```

it would be a waste of a keystroke, but it would just be the first (empty) statement inside the block, and the writing would still follow: The extra semicolon would have no effect.

The corresponding error at the end of a method heading will at least generate a compiler error, though it may appear cryptic:

```
static void badSemicolon(int x);
{
    x = x + 2;
    // ...
```

This is another easy one to make and *miss* - just one innocent semicolon.

5.6.2 Match Wrong `if` With `else`

If you do not consistently put the substatements for the true and false choices inside braces, you can run into problems from the fact that the else part of an if statement is *optional*. Even if you use braces consistently, you may well need to read code that does not place braces around single statements. If C# understood indentation as in the recommended formatting style (or as required in Python), the following would be OK:

```
if (x > 0)
    if (y > 0)
        Console.WriteLine("positive x and y");
else
    Console.WriteLine("x not positive, untested y");
```

Unfortunately placing the `else` under the first `if` is not enough to make them go together (remember the C# compiler ignores extra whitespace). The following is equivalent to the compiler, with the else apparently going with the second if:

```
if (x > 0)
    if (y > 0)
        Console.WriteLine("positive x and y");
    else
        Console.WriteLine("x not positive, untested y");
```

The compiler is consistent with the latter visual pattern: an `else` goes with the most *recent* `if` that could still take an `else`. Hence if x is 3 and y is -2, the `else` part is executed and the statement printed is incorrect: in this code the else clause is only executed when x is positive and y (*is* tested and) is not positive.

If you put braces everywhere to reinforce your indentation, as we suggest, or if you only add the following one set of braces around the inner if statement:

```
if (x > 0) {
    if (y > 0)
        Console.WriteLine("positive x and y");
}
else
    Console.WriteLine("x not positive, untested y");
```

then the braces enclosing the inner `if` statement make it impossible for the inner `if` to continue on to an optional `else` part. The `else` must go with the first `if`. Now when the `else` part is reached, the statement printed will be true: `x` is not positive, and the test of `y` was skipped.

5.6.3 Missing Braces

Another place you can fool yourself with nice indenting style is something like this. Suppose we start with a perfectly reasonable

```
if (x > 0)
    Console.WriteLine("x is: positive");
```

We may decide to avoid the braces, since there *is* just one statement that we want as the if-true part, but if we later decide that we want this on two lines and change it to

```
if (x > 0)
    Console.WriteLine("x is:");
    Console.WriteLine("  positive");
```

We are not going to get the behavior we want. The word "positive" will *always* be printed.

If we had first taken a bit more effort originally to write

```
if (x > 0) {
    Console.WriteLine("x is: positive");
}
```

then we could have split successfully into

```
if (x > 0) {
    Console.WriteLine("x is:");
    Console.WriteLine("  positive");
}
```

This way we do not have to keep worrying about this question when we revise: "Have I switched to multiple lines after the `if` and need to introduce braces?"

The last two of the pitfalls mentioned in this section are fixed by consistent use of braces in the sub-statements of `if` statements. They fix the `;` after if-condition

problem only if the open brace comes right after the condition, but you still get a nasty error if you put in a semicolon between the condition and opening brace.

5.7 Compound Boolean Expressions

To be eligible to graduate from Loyola University Chicago, you must have 120 credits *and* a GPA of at least 2.0. C# does not use the word *and*. Instead it uses `&&` (inherited from the C language). Then the requirement translates directly into C# as a *compound condition*:

```
credits >= 120 && GPA >=2.0
```

This is true if both `credits >= 120` is true and `GPA >= 2.0` is true. A short example function using this would be:

```csharp
static void checkGraduation(int credits, double GPA)
{
    if (credits >= 120 && GPA >=2.0) {
        Console.WriteLine("You are eligible to graduate!");
    }
    else {
        Console.WriteLine("You are not eligible to graduate.");
    }
}
```

The new C# syntax for the operator `&&`:

> *condition1* `&&` *condition2*

The compound condition is true if both of the component conditions are true. It is false if at least one of the conditions is false.

Suppose we want a C# condition that is true in the same situations as the mathematical expression: low < val < high. Unfortunately the math is not a C# expression. The C# operator < is binary. In C# the statement above is equivalent to

> (low < val) < high

comparing a Boolean result to high, and causing a compiler error. There is a C# version. Be sure to use this pattern:

```
low < val && val < high
```

Now suppose we want the opposite condition: that val is *not* strictly between low and high. There are several approaches. One is that val would be less than or equal to low *or* greater than or equal to high. C# translate *or* into ||, so a C# expression would be:

val <= low || val >= high

The new C# syntax for the operator ||:

condition1 || *condition2*

The compound condition is true if at least one of the component conditions are true. It is false if both conditions are false.

Another logical way to express the opposite of the condition low < val < high is that it is *not* the case that low < val && val << high. C# translates *not* as !. Another way to state the condition would be

```
!(low < val && val < high)
```

The parentheses are needed because the ! operator has higher precedence than <.

A way to remember this strange *not* operator is to think of the use of ! in the not-equal operator: !=

The new C# syntax for the operator !:

! *condition*

This whole expression is true when *condition* is false, and false when *condition* is true.

Because of the precedence of !, you are often going to write:

! (*condition*)

Remember when such a condition is used in an if statement, *outer* parentheses are also needed:

if (!(*condition*)) {

We now have a lot of operators! Most of those in appendix *Precedence of Operators* have now been considered. There you can see that ! has the high precedence

---

of unary arithmetic operators. The operators `&&` and `||` are almost at the bottom of the operators considered in this book, just above the assignment operators, and below the relational operators, with `&&` above `||`. You are encourages to use parentheses to make sure.

Compound Overkill: Look back to the code converting a score to a letter grade in *Multiple Tests and if-else Statements*. The condition before assigning the B grade could have been:

```
(score >= 80 && score < 90)
```

That would have totally nailed the condition, but it is overly verbose in the `if .. else if ...` code where it appeared: Since you only get to consider a B as a grade if the grade was *not* already set to A, the second part of the compound condition above is redundant.

There are a couple more wrinkles with compound Boolean expressions introduced later in *Short-Circuiting && and ||*.

5.7.1 Congress Exercise

A person is eligible to be a US Senator who is at least 30 years old and has been a US citizen for at least 9 years. Write a version of a program `congress.cs` to obtain age and length of citizenship from the user and print out if a person is eligible to be a Senator or not. A person is eligible to be a US Representative who is at least 25 years old and has been a US citizen for at least 7 years. Elaborate your program `congress.cs` so it obtains age and length of citizenship and prints whether a person is eligible to be a US Representative only, or is eligible for both offices, or is eligible for neither.

This exercise could be done by making an exhaustive treatment of all possible combinations of age and citizenship. Try to avoid that. (Note the paragraph just before this exercise.)

Caution: be sure to do exhaustive testing. It is easy to write code that is correct for *some* inputs, but not all.

5.7.2 Implication Exercise

We have introduced C# Boolean operators for AND, OR, and NOT. There are other Boolean operators important in logic, that are not directly given as a C# operator. One example is "implies", also expressed in a logical if-then statement: If I am expecting rain, then I am carrying an umbrella. Otherwise put: "I am expecting rain" *implies* "I am carrying an umbrella". The first part is a Boolean expression called the *hypothesis*, and the second part is called the *conclusion*. In general, when A and B are Boolean expressions, "A implies B" is also a Boolean expression.

Just as the truth of a compound Boolean expression like "A and B" depends on the truth value of the two parts, so with *implies*: If you are using good logic, and you start with a true assertion, you should only be able to conclude something else true, so it is true that "true implies true". If you start with garbage you can use that false statement in a logical argument and end up with something either false or true: "false implies false" and "false implies true" are both true. The only thing that should not work is to start with something true and conclude something false. If that were the case, logical arguments would be useless, so "true implies false" is false. There is no C# operator for "implies", but you can check all four cases of Boolean values for A and B to see that "A implies B" is true exactly when "not A or B" is true. We can express this in C# as `!A || B`.

So here is a silly little exercise illustrating both implication and using the C# Boolean operators: Ask the user whether "I am expecting rain" is true. (We have the UI function Agree.) Then check with the user whether "I am carrying an umbrella." Then conclude and print out whether the implication "If I am expecting rain, then I am carrying an umbrella." is true or not in this situation.

5.8 Nested `if` Statements

In `if-else` statements the substatements (the if-true and if-false clauses) are quite arbitrary statements. They can be more `if` or `if-else` statements. In *Multiple Tests and if-else Statements* we have just illustrated placing an `if-else` statement as the `else` clause, and repeating this pattern, to repeatedly test for one more case, stopping when the first true condition if reached. To choose one case from multiple cases, each condition separates one case terminal case from all the remaining untested cases.

Consider a different situation: Steven Covey suggested that people classify possible actions on two axes: urgent vs. not urgent and important vs. not important, leading to four possible combinations. We could ask a person to classify an activity this way, and them give a process comment, something like from Covey's book:

- Important and urgent: Be sure to schedule this promptly!

- Important and not urgent: Make sure that this is included regularly in your plans! Do not let urgent but unimportant things interfere!

- Not important and urgent: Can you skip this, or is it really worth letting this displace important things you need to do?

- Not important and not urgent: Is there anything more worthwhile for you to do now?

Assume we have Boolean variables `important` and `urgent`. There are four separate combinations, and we could handle this with a chain of compound conditions checking for one at a time:

```
if (important && urgent) {
   Console.WriteLine("Be sure ...");
}
else if (important && !urgent) {
   Console.WriteLine("Make sure ...");
}
else if (!important && urgent) {
   Console.WriteLine("Can you...");
}
else {
   Console.WriteLine("Is there ...");
}
```

Compound test conditions are not necessary if we keep track of partial answers, nesting `if` statements, thinking about the two aspects separately:

```
if (important) {
   if (urgent) {
      Console.WriteLine("Be sure ...");
   }
   else {
      Console.WriteLine("Make sure ...");
```

```
    }
}
else {
    if (urgent) {
        Console.WriteLine("Can you...");
    }
    else {
        Console.WriteLine("Is there ...");
    }
}
```

The outer `if-else` determines whether the action is important, so the inner conditions only need to deal with urgency. Also note that in executing this version there are never more than two short conditions evaluated. In the first version, you may have to go through all three conditions. Both approaches work. Which is clearer to you?

5.8.1 Cartesian Plane Location Exercise/Example

Points in the Cartesian plane are given by an x and a y coordinate. Seven parts of the Cartesian plane are labeled in the figure below.

So we match each point with only one name, refer to the point where both x and y are 0 as the origin, and only use the terms x axis or y axis when the point is *not* the origin.

Write a program prompting the user for integer x and y values, and print out the part of the plane as named in the figure. Separate the input and output from the naming logic: Have a function with x and y coordinates as parameters that returns the name of the part of the Cartesian plane.

There are several possible approaches to this function:

- Since each part is associated with a condition on both x and y, you could write a 7 clause chain of if else if else .... with each condition being a *compound* Boolean expression checking for another specific case. This gives some more practice with compound boolean expressions.

- The previous version is conceptually straightforward, but you end up checking the sign of x and y many more times than you need to. Alternately you can make each condition check only the sign of *one* coordinate, and *nest* `if-else` statements checking one coordinate inside an `if-else` statement checking the other coordinate. This is more complicated than the Covey importance and urgency example, since each sign has three possibilities (+, 0, -) rather than two. *Our* solutions to this approach are in example cartesian/cartesian.cs. There are actually two alternative solutions functions there. The first version uses many `if-else` statements, but since each clause executes a `return` statement that stops any further execution, no `else` clauses are actually needed, as shown in `PartOfPlane2`.

5.9 Chapter Review Questions

1. Which of these are Boolean expressions? Assume the variables are of type `int`:

```
true
"false"
x = 3
n < 10
count == 22
x <= 2 || x > 10
x == 2 || 3
1 < y < 10
```

2. What are the values of these expressions? Be able to explain:

```
2 < 3 && 4 < 5
2 < 3 && 4 < 3
2 < 3 || 4 < 5
2 < 3 || 4 < 3
3 < 2 || 4 < 3
2 < 3 || 4 < 5 && 4 < 3
```

3. Correct the last two entries in the first problem, supposing the user meant "x could be either 2 or 3" and then "y is strictly between 1 and 10".

4. Add parentheses in `2 < 3 || 4 < 5 && 4 < 3` to get a different result.

5. Suppose you have four possible distinct situations in your algorithm, each requiring a totally different response in your code, and exactly one of the situations is sure to occur. Have many times must you have `if` followed by a condition?

6. Suppose you have four possible distinct situations in your algorithm, each requiring a totally different response in your code, and at most one of the situations will occur, so possibly nothing will happen that needs a response at all. Have many times must you have `if` followed by a condition?

7. Assume `IsBig(x)` returns a Boolean value. Remove the redundant part of this statement:

```
if (IsBig(x) == true)
    x = 3;
```

8. Write an equivalent (and much shorter!) statement with no `if`:

```
if (x > 7)
    return true;
else
    return false;
```

9. Write an equivalent (and much shorter!) statement with no `if`:

```
if (x > 7)
    isSmall = false;
else
    isSmall = true;
```

10. Assume `x` and `y` are local `int` variables. Code fragments are separated by a blank line below. Pairs of the fragments are logically equivalent, but not necessarily with a directly adjacent fragment. Match the pairs. Be sure you understand when different pairs would behave differently. Caution: there is some pretty awful code here, that we would *hope* you would never write, but you might need to correct/read! Think of pitfalls. In each equivalent pair, which code fragment is more professional?

```
if (x > 7) {      //a
    x = 5;
}
y = 1;

if (x > 7) {      //b
    x = 5;
    y = 1;
}

if (x > 7)        //c
    x = 5;
    y = 1;

if (x > 7) {      //d
    x = 5;
}
else {
    y = 1;
}

if (x > 7)        //e
    x = 5;
else if (x <= 7) {
    y = 1;
}

if (x > 7) {      //f
    y = 1;
}
if (x > 7) {
    x = 5;
}
```

11. Same situation as the last problem, and same caution, except this time as-
    sume the fragments appear in a function that returns an int. In each pair
    of equivalent fragments, which is your preference?

```
y = 1;            //a
if (x > 7) {
    return x;
}
```

```
    if (x > 7) {     //b
        return x;
    }
    y = 1;

    if (x > 7) {     //c
        return x;
    }
    else {
        y = 1;
    }

    if (x > 7) {     //d
        return x;
        y = 1;
    }

    if (x > 7) {     //e
        y = 1;
        return x;
    }
    y = 1;

    if (x > 7) {     //f
        return x;
    }

    if (x > 7);      //g
        return x;

    return x;        //h
```

12. Same situation as the last problem, and same caution:

```
    if (x > 5)          //a
        if (x > 7)
            return x;
    else
        y = 1;

    if (x > 5)   {      //b
```

```
        if (x > 7)
            return x;
    }
    else {
        y = 1;
    }

    if (x > 7)          //c
        return x;
    if (x <= 5)
        y = 1;

    if (x > 7)          //d
        return x;
    if (x > 5)
        y = 1;
```

WHILE LOOPS

6.1 While-Statements

We have seen that the sequential flow of a program can be altered with function calls and decisions. The last important pattern is *repetition* or *loops*. There are several varieties. The simplest place to start is with `while` loops.

A C# `while` loop behaves quite similarly to common English usage. If you hear

> While your tea is too hot, add a chip of ice.

Presumably you would test your tea. If it were too hot, you would add a little ice. If you test again and it is still too hot, you would add ice again. *As long as* you tested and found it was true that your tea was too hot, you would go back and add more ice. C# has a similar syntax:

```
while ( condition )
      statement
```

As with an `if` statement we will generally assume a compound statement, after the condition, so the syntax will actually be:

```
while ( condition ) {
      statement(s)
}
```

Setting up the English example as pseudocode in a similar format would be:

```
while ( your tea is too hot ) {
```

> add a chip of ice
> }

To make things concrete and numerical, suppose the following: The tea starts at 115 degrees Fahrenheit. You want it at 112 degrees. A chip of ice turns out to lower the temperature one degree each time. You test the temperature each time, and also print out the temperature before reducing the temperature. In C# you could write and run the code below, saved in example program cool/cool.cs:

```
1  int temperature = 115;
2  while (temperature > 112) { // first while loop code
3      Console.WriteLine(temperature);
4      temperature = temperature - 1;
5  }
6  Console.WriteLine("The tea is cool enough.");
```

We added a final line after the `while` loop to remind you that execution follows sequentially after a loop completes.

It is extremely important to totally understand how the flow of execution works with loops. One way to follow it closely is to make a table with a line for each instruction executed, keeping track of all the variables, playing computer. as with `if` statements, the executed lines that you show in your table will not be in textual order, as in *Sequential Execution*. While `if` statements merely altered execution order by skipping some lines, loops allow the same line in the text of your program to be executed repeatedly, and show up in multiple places in your table.

If you play computer and follow the path of execution, you could generate the following table. Remember, that each time you reach the end of the block after the `while` heading, execution returns to the `while` heading for another test:

Line	temperature	Comment
1	115	
2		115 > 112 is true, do loop
3		prints 115
4	114	115 - 1 is 114, loop back
2		114 > 112 is true, do loop
3		prints 114
4	113	114 - 1 is 113, loop back
2		113 > 112 is true, do loop
3		prints 113
4	112	113 - 1 is 112, loop back
2		112 > 112 is false, skip loop
6		prints that the tea is cool

Each time the end of the loop body block is reached, execution *goes back* to the `while` loop heading for another test. When the test is finally false, execution jumps past the indented body of the `while` loop to the next sequential statement.

Note: Unless a program is purely sequential, the numbers under the **Line** column are *not* just in textual, sequential order. The order of the numbers is the order of *execution*. Each line number in the "playing computer" table is the line *number label* for the next particular line *getting executed*. Since in decisions, loops, and function calls, lines may be reordered or repeated, the corresponding line numbers may be skipped, repeated, or otherwise out of numerical order.

The biggest trick with a loop is to make the same code do the next thing you want each time through. That generally involves the use of variables that are modified for each successive time through the loop. Here is a general pattern:

> initialization
>
> `while` (*continuationCondition*) {
>
> > do main action to be repeated
> >
> > prepare variables for the next time through the loop
>
> }

The simple first example follows this pattern directly. Note that the variables needed for the test of the condition must be set up *both* in the initialization *and* inside the loop (often at the very end). Without a change inside the loop, the loop would run forever!

How to manage all of this in general is a big deal for beginning students. We will see a number of common patterns in lots of practice. We will use the term *successive modification loop* for loops following the pattern above.

Test yourself: Follow the code. Figure out what is printed. If it helps, get detailed and play computer:

```
1  int i = 4;
2  while (i < 9) {
3      Console.WriteLine(i);
4      i = i + 2;
5  }
```

Check yourself by running the example program test_while1/test_while1.cs.

Note: In C#, `while` is not used *quite* like in English. In English you could mean to stop *as soon as* the condition you want to test becomes false. In C# the test is *only* made when execution for the loop starts (or starts again), *not* in the middle of the loop.

Predict what will happen with this slight variation on the previous example, switching the order in the loop body. Follow it carefully, one step at a time.

```
1  int i = 4; //variation on TestWhile1.cs
2  while (i < 9) {
3      i = i + 2;
4      Console.WriteLine(i);
5  }
```

Check yourself by running the example program test_while2/test_while2.cs.

The line sequence is important. The variable `i` is increased before it is printed, so the first number printed is 6. Another common error is to assume that 10 will *not* be printed, since 10 is *past* 9, but the test that may stop the loop is *not* made in the middle of the loop. Once the body of the loop is started, it continues to the end, even when `i` becomes 10.

Line	i	Comment
1	4	
2		4 < 9 is true, do loop
3	6	4+2=6
4		print 6
2		6 < 9 is true, do loop
3	8	6+2= 8
4		print 8
2		8 < 9 is true, do loop
3	10	8+2=10 *No test here*
4		print 10
2		10 < 9 is false, skip loop

You should be able to generate a table like the one above, following the execution of one statement at a time. You are playing through the role of the computer in detail. As code gets more complicated, particularly with loops, this "playing computer" is an important skill.

Problem: Write a program with a `while` loop to print:

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

Analysis: We have seen that we can produce a regular sequence of numbers in a loop. The "Blastoff!" part does not fit the pattern, so it is logically a *separate* part after the loop. We need a name for the number that decreases. It can be `time`. Remember the general rubric for a `while` loop:

      initialization
      `while` (*continuationCondition*) {
          do main action to be repeated
          prepare variables for the next time through the loop

```
      }
```

You can consider each part separately. Where to start is partly a matter of taste.

The main thing to do is print the time over and over. The initial value of the time is 10. We are going to want to keep printing until the time is down to 1, so we *continue* while the time is at least 1, meaning the continuationCondition can be `time >= 1`, or we could use `time > 0`, since `time` is an integer here.

Finally we need to get ready to print a different time in the next pass through the loop. Since each successive time is one less than the previous one, the preparation for the next value of time is: `time = time - 1`.

Putting that all together, and remembering the one thing we noted to do after the loop, we get blastoff/blastoff.cs:

```csharp
using System;

class Blastoff
{
   static void Main()
   {
      int time = 10;
      while (time > 0) {
         Console.WriteLine(time);
         time = time - 1;
      }
      Console.WriteLine("Blastoff!");
   }
}
```

Look back and see how we fit the general rubric. There are a bunch of things to think about with a while loop, so it helps to go one step at a time, thinking of the rubric and the specific needs of the current problem.

There are many different (and more exciting) patterns of change coming for loops, but the simple examples so far get us started.

Loop Planning Rubric

Looking ahead to more complicated and interesting problems, here is a more complete list of questions to ask yourself when designing a function with a `while`

loop (and most will apply to later forms of loops, too):

- What data is involved? Make sure you give good variable names.

- What needs to be initialized and how? This certainly includes any variable tested in the condition.

- What is the condition that will allow the loop to *continue*? It may be easier to think of the condition that will *stop* the loop. That is fine - but remember to *negate* it (with !) to turn it into a proper *continuation* condition.

- Distinguish: What is the code that should only be executed once? What action do I want to repeat?

- How do I write the repeating action so I can modify it for the next time through the loop to work with new data?

- What code is needed to do modifications to make the same code work the next time through the loop?

- Have I thought of variables needed in the middle and declared them; do other things need initialization?

- Will the continuation condition eventually fail? *Be sure to think about this!*

- Separate the actions to be done once before the repetition (code before the loop) from repetitive actions (in the loop) from actions not repeated, but done after the loop (code after the loop). Missing this distinction is a *common error*!

6.1.1 Sum To n

Let us write a function to sum the numbers from 1 to n:

```
/// Return the sum of the numbers from 1 through n.
static int SumToN(int n)
{
    ...
}
```

For instance SumToN(5) calculates $1 + 2 + 3 + 4 + 5$ and returns 15. We know how to generate a sequence of integers, but this is a place that a programmer gets

---

tripped up by the speed of the human mind. You are likely so quick at this that you just see it all at once, with the answer.

In fact, you and the computer need to do this in steps. To help see, let us take a concrete example like the one above for SumToN(5), and write out a detailed sequence of steps like:

```
3 = 1 + 2
6 = 3 + 3
10 = 6 + 4
15 = 10 + 5
```

You could put this in code directly for a specific sum, but if `n` is general, we need a loop, and hence we must see a *pattern* in code that we can repeat.

In each calculation the second term in the additions is a successive integer, that we can generate. Starting in the second line, the first number in each addition is the sum from the previous line. Of course the next integer and the next partial sum change from step to step, so in order to use the same code over and over we will need changeable variables, with names. We can make the partial sum be `sum` and we can call the next integer `i`. Each addition can be in the form:

```
sum + i
```

We need to remember that result, the new sum. You might first think to introduce such a name:

```
newSum = sum + i;
```

This will work. We can go through the `while` loop rubric:

The variables are `sum`, `newSum` and `i`.

To evaluate

```
newSum = sum + i;
```

the first time in the loop, we need *initial* values for sum and i. Our concrete example leads the way:

```
int sum = 1, i = 2;
```

We need a `while` loop heading with a continuation condition. How long do we want to add the next `i`? That is for all the value up to and including n:

```
while (i <= n) {
```

There is one more important piece - making sure the same code

> newSum = sum + i;

works for the *next* time through the loop. We have dealt before with the idea of the next number in sequence:

```
i = i + 1;
```

What about `sum`? What was the `newSum` on *one* time through the loop becomes the old or just plain `sum` the *next* time through, so we can make an assignment:

```
sum = newSum:
```

All together we calculate the sum with:

```
int sum = 1, i = 2;
while (i <= n) {
    int newSum = sum + i;
    sum = newSum:
    i = i + 1;
}
```

This exactly follows our general rubric, with preparation for the next time through the loop at the end of the loop. We can condense it in this case: Since `newSum` is only used once, we can do away with this extra variable name, and directly change the value of sum:

```
int sum = 1, i = 2;
while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
```

Finally this was supposed to fit in a function. The ultimate purpose was to *return* the sum, which is the final value of the variable `sum`, so the whole function is:

```
/// Return the sum of the numbers from 1 through n.
static int SumToN(int n)      // line 1
{
    int sum = 1, i = 2;       // 2
    while (i <= n) {          // 3
```

6.1. While-Statements   **175**

```
        sum = sum + i;          // 4
        i = i + 1;              // 5
    }
    return sum;                 // 6
}
```

The comment before the function definition does not give a clear idea of the range of possible values for n. How small makes sense for the comment? What actually works in the function? The smallest expression starting with 1 would just be 1: (n is 1). Does that work in the function? You were probably not thinking of that when developing the function! Now look back now at this *edge case*. You can play computer on the code or directly test it. In this case the initialization of sum is 1, and the body of the loop *never* runs (2 <= 1 is false). The function execution jumps right to the return statement, and does return 1, and everything is fine.

Also you should check the program in a more general situation, say with n being 4. You should be able to play computer and generate this table, using the line numbers shown in comments at the end of lines, and following one statement of execution at a time. We only make entries where variables change value.

Line	i	sum	Comment
1			assume 4 is passed for n
2	2	1	
3			2<=4: true, enter loop
4		3	1+2=3
5	3		2+1=3, bottom of loop
3			3<=4: true
4		6	3+3=6
5	4		3+1=4, bottom of loop
3			4<=4: true
4		10	6+4=10
5	5		4+1=5, bottom of loop
3			5<=4: false, skip loop
6			return 10

The return only happens once, so it is not in the loop. You get *a* value for a sum each time through, but not the final one. A common beginner error is to put the return statement inside the loop, like

```
static int SumToN(int n)   // 1    BAD VERSION!!!
{
   int sum = 1, i = 2;     // 2
   while (i <= n) {        // 3
      sum = sum + i;       // 4
      i = i + 1;           // 5
      return sum;          // 6   WRONG!
   }
}
```

Recall that *when a return statement is reached, function execution ends*, *no matter what comes next in the code*. (This is a way to break out of a `while` loop that we will find useful later.) In this case however, it is not what we want at all. The first sum is calculated in line 4, so `sum` becomes 2 + 1, but when you get to line 6, the function terminates and never loops back, returning 3.

Now about large n....

With loops we can make programs run for a long time. The time taken becomes an issue. In this case we go though the loop n-1 times, so the total time is approximately proportional to n. We write that the time is O(n), spoken "oh of n", or "big oh of n" or "order of n".

Computers are pretty fast, so you can try the testing program sum_to_n_test/sum_to_n_test.cs and it will go by so fast, that you will hardly notice. Try these specific numbers in tests: 5, 6, 1000, 10000, 98765. All look OK? Now try 66000. On many systems you will get quite a surprise! This is the first place we have to deal with the limited size of the `int` type. On many systems the limit is a bit over 2 billion. You can check out the size of `int.MaxValue` in csharp. The answer for 66000, and *also* 98765, is bigger than the upper limit. Luckily the obviously wrong negative answer for 66000 pops out at you. Did you guess before you saw the answer for 66000, that there was an issue for 98765? It is a good thing that no safety component in a big bridge was being calculated! It is a big deal that the system fails *silently* in such situations. *Think* how large the data may be that you deal with!

Now look at and run sum_to_n_long/sum_to_n_long.cs. The sum is a `long` integer here. Check out in csharp how big a `long` can be (`long.MaxValue`). This version of the program works for 100000 and for 98765. We can get correct answers for things that will take perceptible time. Try working up to 1 billion (1000000000, nine 0's). It takes a while: O(n) can be slow!

By hand it is a lot slower, unless you totally change the algorithm: There is a classic story about how a calculation like this was done in grade school (n=100) by the famous mathematician Gauss. His teacher was trying to keep him busy. Gauss discovered the general, exact, mathematical formula:

$$1 + 2 + 3 + ... + n = n(n+1)/2.$$

That is the number of terms (n), times the average term (n+1)/2.

Our loop was instructive, but not the fastest approach. The simple exact formula takes about the same time for any n. (That is as long as the result fits in a standard type of computer integer!) This is basically constant time. In discussing how the speed relates to the size of n, we say it is O(1). The point is here that 1 is a constant. The time is of *constant order*.

We can write a ridiculously short function following Gauss's model. Here we introduce the variable average, as in the motivation for Gauss's answer:

```csharp
/// Return the sum of the numbers from 1 through n.
static long SumToN(int n)   //CHANGED:  quick and WRONG
{
   int average = (n+1)/2;      //from Gausse's motivation
   return n*average;
}
```

Run the example program containing it: sum_to_n_long_bad/sum_to_n_long_bad.cs.

Test it with 5, and then try 6. ???

"Ridiculously short" does not imply correct! The problem goes back to the fact that Gauss was in *math class* and you are doing Computer Science. Think of a subtle difference that might come in here: Though (n+1)/2 is fine as math, recall the division operator does not always give correct answers in C#. You get an integer answer from the integer (or long) operands. Of course the exact mathematical final answer is an integer when *adding* integers, but splitting it according to Gauss's motivation can put a mathematical non-integer in the middle.

The C# fix: The final answer is clearly an integer, so if we do the division last, when we know the answer will be an integer (assuming a long integer), things should be better:

```csharp
long sum = n*(n+1)/2;
return sum;
```

Here is a shot at the whole function:

```
/// Return the sum of the numbers from 1 through n.
static long SumToN(int n)   //CHANGED:  quick and still WRONG
{
   long sum = n*(n+1)/2;   // final division will produce an integer
   return sum;
}
```

Run the example program containing it: sum_to_n_long_bad2/sum_to_n_long_bad2.cs.

Test it with 5, and then try 6. Ok so far, but go on to long integer range: try 66000 that messed us up before. ??? You get an answer that is not a multiple of 1000: not what we got before! What other issues do we have between math and C#?

Further analysis: To make sure the function always worked, it made sense to leave the parameter n an int. The function would not work with n as the largest long. The result can still be big enough to only fit in a long, so the return value is a long. All this is reasonable but the C# result is still wrong! Look deeper. While the result of n*(n+1)/2 is *assigned* to a long variable, the *calculation* n*(n+1)/2 is done with ints not mathematical integers. By the same general type rule that led to the (n+1)/2 error earlier, these operations on ints produce an int result, even when wrong.

We need to force the *calculation* to produce a long. In the correct looping version sum was a long, and that forced all the later arithmetic to be with longs. Here are two variations that work:

```
long nLong = n;
return nLong*(nLong+1)/2;
```

or we can avoid a new variable name by *Casting* to long, converting the first (left) operand to long, so all the later left-to-right operations are forced to be long:

```
return (long)n*(n+1)/2;
```

You can try example sum_to_n_long_quick/sum_to_n_long_quick.cs to finally get a result that is dependably fast and correct.

Important lessons from this humble summation problem:

- *Working* and being *efficient* are two different things in general.

- *Math* operations and C# operations are not always the same. Knowing this in theory is not the same as remembering it in practice!

Further special syntax that only makes sense in any kind of loop is discussed in *Break and Continue*, after we introduce the last kind of loop.

6.2 While-Statements with Sequences

6.2.1 One Character Per Line

We will process many sequences or collections. At this point the only collection we have discussed is a string - a sequence of characters that we can index. Consider the following silly function description and heading as a start:

```
// Print the characters of s, one per line.
static void OneCharPerLine(string s)
```

`OneCharPerLine("bug")` would print:

```
b
u
g
```

We are accessing one character at a time. We can do that with the indexing notation. Thinking concretely about the example above, we are looking to print, `s[0], s[1], s[2]`. If we knew we would always have three characters, we could do this with three explicit print statements, but we are looking to write a general definition for an arbitrary length string: This requires a loop. For now our only option is a `while` loop. We can follow our basic *loop planning rubric*, one step at a time: The index is changing in a simple repetitive sequence. We can call the index `i`. Its initial value is clearly 0. That is our initialization. We need a `while` loop continuation condition. For the 3-character string example, the last index above is 2. In general we want *all* the characters. Recall the index of the last character is the length - 1, or with our parameter s, `s.Length - 1` The `while` loop condition needs to allow indices through `s.Length - 1`. We could write a condition with <= or more concisely:

```
while (i < s.Length) {
```

In the body of the loop, the main thing is to print the next character, and the next character is `s[i]`:

```
Console.WriteLine(s[i]);
```

We also need to remember the part to get ready for the next time through the loop. We have dealt with regular sequence of values before. We change `i` with:

```
i = i+1;
```

This change is so common, there is a simpler syntax:

```
i++;
```

This increases the value of the numeric variable i by 1. (The reverse is `i--;`) [1]

So all together:

```
// Print the characters of s, one per line.
static void OneCharPerLine(string s)
{
    int i = 0;
    while (i < s.Length) {
        Console.WriteLine(s[i]);
        i++;
    }
}
```

You can test this with example char_loop1/char_loop1.cs.

This is a very common pattern. We could do anything we want with each individual character, not just print it.

String Backwards Exercise/Example

Here is a variation:

---

[1] To be complete, the statements `c = c + 1;` and `c++;` are not always equivalent.

In `c++` the type of `c` must be numeric, but not necessarily `int`. It could be a *smaller* type, like `char`.

With a `c` of type `char` the `c++` could not be replaced by `c = c + 1`, but you could use `c = (char)(c + 1)`: The `int` literal 1 forces the sum expression to be an `int`, which must be cast back to a `char` to be assigned to `c`. Similarly with the `--` operator.

```
/// Print s in reverse order; no extra newlines
static void PrintReversed(string s)
```

There are a few changes:

- You do not want to go on to the next line, so use `Write`, not `WriteLine`.

- It is still a regular sequence of character indices, but we are working backwards through the string. We have created a decreasing sequence before. Where do you start? Where do you stop? What is the condition? How do you get ready for the next time through the loop? (Remember our newest notation.)

Our code with driver is in reversed_print/reversed_print.cs.

6.2.2 Print Vowels Function

Let us get more complicated. Consider the function described:

```
/// Print the vowels (aeiou) in s, one per line.
static void PrintVowels(string s)
```

For instance PrintVowels("computer") would print:

```
o
u
e
```

We have seen that we can go through the whole string and do the same thing each time through the loop, using `s[i]` in some specific way.

This new description seems problematic. We do *not* appear to want to do the same thing each time: We only want to print *some* of the characters. Again your eyes and mind are so fast, you likely miss what you need to do when you go through `PrintVowels` by hand. Your eyes let you just grab the vowels easily, but think, what is actually happening? You are checking each character to see **if** it is a vowel, and printing it if it is: You are doing the same thing each time - *testing* **if** the character is a vowel. The pseudocode is

```
if (s[i] is a vowel) {
    print s[i]
}
```

We *do* want to do this each time through the loop. We *can* use a `while` statement.

Next task: convert the pseudocode "s[i] is a vowel" to C#.

There are multiple approaches. The one you get by following your nose is just to consider all the cases where it is true:

```
s[i] == 'a'
s[i] == 'e'
s[i] == 'i'
s[i] == 'o'
s[i] == 'u'
```

How do you combine them into a condition? The letter can be a *or* e *or* i *or* o *or* u. We get the code:

```
/// Print the vowels (aeiou) in s, one per line.
static void PrintVowels(string s)
{
    int i = 0;
    while (i < s.Length) {
        if (s[i] == 'a' || s[i] == 'e' || s[i] == 'i' ||
            s[i] == 'o' || s[i] == 'u') {
            Console.WriteLine(s[i]);
        }
        i = i+1;
    }
}
```

That has a long condition! Here is a nice trick to shorten that: We want to check if a character is in a group of letters. We have already seen the string method IndexOf. Recall we can use it to see if a character is in or not in a string. We can use `"aeiou".IndexOf(s[i])`. We do not care *where* `s[i]` comes in the string of vowels. All we care is that `"aeiou".IndexOf(s[i]) >= 0`.

This is still a bit of a mouthful. Often it is just important if a character or string is *contained* in another string, not where it appears, so it is easier to use the string method `Contains`. Though IndexOf takes either a string or a character as parameter, `Contains` only takes a string. There is a nice quick idiom to convert anything to a string: use `""+`. The condition could be `"aeiou".Contains(""+s[i])`. The `""` + `s[i]` adds the string version of `s[i]` to the empty string.

6.2. While-Statements with Sequences **183**

The function is still not as general as it might be: Only lowercase vowels are listed. We could do something with `ToLower`, or just use the condition: `"aeiouAEIOU".Contains(""+s[i])`

This variation is in example vowels2/vowels2.cs.

```
/// Print the vowels (aeiou) in s, one per line.
static void PrintVowels(string s)
{
   int i = 0;
   string vowels = "aeiouAEIOU";
   while (i < s.Length) {
      if (vowels.Contains(""+s[i])) {
         Console.WriteLine(s[i]);
      }
      i++;
   }
}
```

6.2.3 IsDigits Function

Consider a variation, determining if *all* the characters in a string are vowels. We could work on that, but it is not very useful. Instead let us consider if all the characters are digits. This is a true-false question, so the function to determine this would return a Boolean result:

There are several ways to check if a character is a digit. We could use the `Contains` idiom from above, but here is another option: The integer codes for digits are sequential, and since characters are technically a kind of integer, we can compare: The character `s[i]` is a digit if it is in the range from `'0'` to `'9'`, so the condition can be written:

```
'0' <= s[i] && s[i] <= '9'
```

Similarly the condition `s[i]` is not a digit, can be written negating the compound condition as in *Compound Boolean Expressions*:

```
s[i] < '0' || s[i] > '9'
```

If you think of going through by hand and checking, you would check through the characters sequentially and if you find a non-digit, you would want to *remember*

that the string is not only digits.

One way to do this is have a variable holding an answer so far:

```
bool allDigitsSoFar = true;
```

Of course initially, you have not found any non-digits, so it starts off true. As you go through the string, you want to make sure that answer is changed to `false` if a non-digit is encountered:

```
if ('0' > s[i] || s[i] > '9') {
    allDigitsSoFar = false;
}
```

When we get all the way through the string, the answer so far is the final answer to be returned:

```
/// Return true if s contains one or more digits
/// and nothing else. Otherwise return false.
static bool IsDigits(string s)
{
   bool allDigitsSoFar = true;
   int i = 0;
   while (i < s.Length) {
      if (s[i] < '0' || s[i] > '9') {
         allDigitsSoFar = false;
      }
      i++;
   }
   return allDigitsSoFar;
}
```

Remember something to always consider: edge cases. In the description it says it is true for a string of *one or more* digits.

Check examples of length 1 and 0. Length 1 is fine, but it fails for the empty string, since the loop is skipped and the initial answer, `true` is returned.

There are many ways to fix this. We will know right up front that the answer is false if the length is 0, and we could immediately set `allDigitsSoFar` to false. We would need to change the initialization so it checks the length and chooses the right value for `allDigitsSoFar`, true or false. Since we are selecting between two values, an `if` statement should occur to you:

```
bool allDigitsSoFar;
if (s.Length > 0) {
    allDigitsSoFar = true;
}
else {
    allDigitsSoFar = false;
}
```

If we substitute this initialization for `allDigitsSoFar`, the code will satisfy the edge case, and the code will always work. Still, this code can be improved:

Examine the `if` statement more closely:

> if the condition is `true`, `allDigitsSoFar` is `true`;
>
> if the condition is `false`, `allDigitsSoFar` is `false`;

See the symmetry: the value assigned to `allDigitsSoFar` is always the *value of the condition*.

A *much* more concise and still equivalent initialization is just:

```
bool allDigitsSoFar = (s.Length > 0);
```

In more generality this conciseness comes from the fact that it is a *Boolean* value that you are trying to set each time, based on a *Boolean* condition: You do not need to do that with an `if` statement! You just need an assignment statement. If you use an `if` statement in such a situation, you being verbose and marking yourself as a novice!

It could even be slightly more concise: The precedence of assignment is very low, lower than the comparison >, so the parentheses could be omitted. We think the code is easier to read with the parentheses left in, as written above.

The whole function would be:

```
/// Return true if s contains one or more digits
/// and nothing else. Otherwise return false.
static Boolean IsDigits(string s)
{
    Boolean allDigitsSoFar = (s.Length > 0);
    int i = 0;
    while (i < s.Length) {
        if (s[i] < '0' || s[i] > '9') {
```

```
            allDigitsSoFar = false;
        }
        i++;
    }
    return allDigitsSoFar;
}
```

You can try this code in example check_digits1/check_digits1.cs.

Note that we earlier made an improvement by replacing an `if-else` statement generating a Boolean value by a simple Boolean assignment. In the most recent sample code, there is an `if` statement setting a Boolean value:

```
if (s[i] < '0' || s[i] > '9') {
    allDigitsSoFar = false;
}
```

You might be tempted to replace this `if` statement by a simple Boolean assignment:

```
allDigitsSoFar = (s[i] < '0' || s[i] > '9');  // bad!
```

Play computer with this change to see for yourself why it is bad, before looking at our explanation below....

The place where we originally said to use a simple Boolean assignment was replacing an `if-else` statement, that *always* set a Boolean value. In the more recent correct code for digits, we had a simple `if` statement, and were only setting the boolean variable to `false` *some* of the time: when we had *not* found a digit. The bad code sets the variable for *each* character in the string, so it can change an earlier `false` value back to `true` for a later digit. The final value always comes from the the *last* character in the string! We want the function to come up with an answer `false` if *any* character is not a digit, not just the last character. The bad code would give the wrong answer with the string "R2D2". If you do not see that, play computer with this string and the bad code variation that sets `allDigitsSoFar` every time through the loop.

There is a less commonly useful way to make an assignment without `if` work here [2], but a much more important, improved approach follows:

---

[2] The Boolean assignment did not work when `allDigitsSoFar` was already `false`, and the next character was a digit. This could be fixed with a compound Boolean expression in the assignment:
`allDigitsSoFar = allDigitsSoFar && (s[i] < '0' || s[i] > '9');`

The last correct code is still inefficient. If an early character in a long string is not a digit, we already know the final answer, but this code goes through and still checks all the other characters in the string! People checking by hand would stop as soon as they found a non-digit. We can do that in several ways with C#, too. Since this is a function, and we would know the final answer where we find a non-digit, the simplest thing is to use the fact that a return statement *immediately terminates* the function (even if in a loop).

Instead of setting a variable to `false` to *later* be returned, we can return right away, using the loop:

```
while (i < s.Length) {
    if (s[i] < '0' || s[i] > '9') {
        return false;
    }
    i++;
}
```

What if the loop terminates normally (no return from inside)? That means no non-digit was found, so if there are any characters at all, they are all digits. There are *one or more* digits as long as the string length is *positive*. Again we do *not* need an `if-else` statement to check the length and set the Boolean result. Look in the full code for the function:

```
/// Return true if s contains one or more digits
/// and nothing else. Otherwise return false.
static Boolean IsDigits(string s)
{
    int i = 0;
    while (i < s.Length) {
        if (s[i] < '0' || s[i] > '9') {
            return false;
        }
        i++;
    }
    return (s.Length > 0);
}
```

The full code with a `Main` testing program is in example check_digits2/check_digits2.cs.

This way, once `allDigitsSoFar` is false, it stays `false`.

Returning out of a loop is a good pattern to remember when you are searching for something, and you know the final answer for your function as soon as you find it.

6.2.4 Play Computer With a Loop

We have not given you a chance to play computer with a loop. Here is some simple silly code, loop_steps/loop_steps.cs, also using a sequence:

```csharp
using System;

class LoopSteps
{
    static void Main()
    {   // play computer and predict what this loop does
        string s = "abcd";
        int i = 1;
        while (i < 4) {
            Console.Write ("/" + s[i] + s[i - 1]);
            i++;
        }
        Console.WriteLine();
    }
}
```

Play computer, completing the table. You fill in the line numbers, carefully. The sequence is *not* 9, 10, 11, 12, 13!

Line	i	Comment
5	-	Start at beginning of Main
7		set s = "abcd" (does not change)
8	1	initialize i
...		

Duplicate Character Exercise

Create a file `double_char_test.cs`, and write and test a function with the documentation and heading below:

```
/// If two consecutive characters in s are the same, return true.
/// Return false otherwise.  Examples:
/// HasDoubleChar("bigfoot") and HasDoubleChar("aaah!") are true;
/// HasDoubleChar("treated") and HasDoubleChar("haha!") are false.
static bool HasDoubleChar(string s)
```

You may want to play computer on a short example - there is an easy mistake to make.

6.3 Interactive `while` Loops

Next we consider a particular form of `while` loops: Interactive while loops involve input from the user each time through the loop. We consider them now for three reasons:

- Interactive `while` loops have one special 'gotcha' worth illustrating.

- We will illustrate some general techniques for understanding and developing `while` loops.

- As a practical matter, we can greatly improve the utility input functions we have been using, and add some more.

We already have discussed the PromptInt function. The user can choose any int. Sometimes we only want an integer in a certain range. One approach is to not accept a bad value, but make the user repeat trying until explicitly entering a value in the right range. In theory the user could make errors for some time, so a loop makes sense. For instance we might have a slow user, and there could be an exchange like the following when you want a number from 0 to 100. For illustration, user input is shown in boldface:

> Enter a score: (0 through 100) **233**
>
> 233 is out of range!
>
> Enter a score: (0 through 100) **101**
>
> 101 is out of range!
>
> Enter a score: (0 through 100) **-1**
>
> -1 is out of range!
>
> Enter a score: (0 through 100) **100**

and the value 100 would be accepted.

This is a well-defined idea. A function makes sense. Its heading includes a prompt and low and high limits of the allowed range:

```
/// Prompt the user to obtain an int until the response is in the
/// range [lowLim, highLim].  Then return the int in range.
static int PromptIntInRange(string prompt, int lowLim, int highLim)
```

For example to generate sequence above, the call would be:

```
PromptIntInRange("Enter a score: (0 through 100) ", 0, 100)
```

There is an issue with the common term "loop" in programming. In normal English, a loop has no beginning and no end, like a circle. C# loops have a sequence of statements with a definite beginning and end.

Consider the sequence above in pseudocode.

```
Input a number with prompt (233)
Print error message
Input a number with prompt (101)
Print error message
Input a number (-1)
Print error message
Input a number with prompt (100)
Return 100
```

We can break this into a repeating pattern in two ways. The most obvious is the following, with three repetitions of a basic pattern, with the last two line not in the same pattern (so they would go after the loop). :

```
Input a number with prompt (233)
Print error message

Input a number with prompt (101)
Print error message

Input a number (-1)
Print error message

Input a number with prompt (100)
Return 100
```

Another choice, since you can split a loop at any point, would be the following, with the first and last lines not in the pattern that repeats three times in the middle:

```
Input a number with prompt (233)

Print error message
Input a number with prompt (101)

Print error message
Input a number (-1)

Print error message
Input a number with prompt (100)

Return 100
```

When you consider `while` loops, there is a problem with the first version: Before the first pass through the loop and at the end of the block of code in the body of the loop, you must be *able* to run the test in the while heading. We will be testing the latest input from the user.

It is the second version that has us getting new input *before the first* loop *and* at the *end of each* loop!

Now we can think more of the basic process to turn this into a C# solution: What variables do we need? We will call the user's response `number`.

What is the test in the while loop heading? The easiest thing to think of is that we are done when we get something correct. That, however, is a *termination* condition. We need to reverse it to get the *continuation condition*, that the answer is out of range. There are two ways to be out of range:

```
number < lowLim
number > highLim
```

How do we combine them? Either one rules out a correct answer, so `number` is out of range if too high OR too low. Remember the C# symbolism for "or": `||`:

```
while (number < lowLim || number > highLim) {
```

Following the sequence in the concrete example we had above, we can see how to put things together. We need to get input from the user *before* first beginning the

`while` loop, so we immediately have something to test in the `while` heading's condition.

Do not reinvent the wheel! We can use our earlier general `PromptInt` function. It needs a prompt. As a first version, we can use the parameter `prompt`:

```
int number = PromptInt(prompt);
```

That is the initialization step before the loop.

If we get into the body of the loop, it means there is an error, and the concrete example indicates we print a warning message. The concrete example *also* shows another step in the loop, asking the user for input. It is easy to think

> "I already have the code included to read a value from the user, so there is nothing really to do."

WRONG! The initialization code with the input from the user is *before* the loop. C# execution approaches the test in the `while` headings from *two* places at different times: the initialization *and* coming back from the bottom of the loop. To get a *new* value to test, we must *repeat* getting input from the user at the *bottom of the loop body*.

You might decide to be quick and just copy the initialization line into the bottom of the loop (and indent it):

```
int number = PromptInt(prompt);
```

Luckily you will get a compiler error in that situation, avoiding more major troubleshooting: The *complete* copy of the line copies the *declaration* part as well as the assignment part, and the compiler sees the declaration of `number` already there from the scope outside the while block, and complains.

Hence copy the line, *without* the `int` declaration:

```
number = PromptInt(prompt);
```

When the loop condition becomes false, and you get past the loop, you have a correct value in `number`. You have done all the hard work. Do not forget to return it at the end.

```
/// Prompt the user to obtain an int until the response is in the
/// range [lowLim, highLim].  Then return the int in range.
static int PromptIntInRange(string prompt, int lowLim, int highLim)
```

```
{
   int number = UIF.PromptInt(prompt);
   while (number < lowLim || number > highLim) {
      Console.WriteLine("{0} is out of range!", number);
      number = UIF.PromptInt(prompt);
   }
   return number;
}
```

You can try this full example, input_in_range1/input_in_range1.cs. Look at it and then try compiling and running.

Look at the Main code. It is redundant - the limits are written both in the prompt and in the parameters. We can do better. In general we endeavor to supply data only once, and let the program use it in several places if it needs to. Since the limits are given as parameters, anyway, we prefer to have the program elaborate the prompt automatically. If the limits are -10 and 10, automatically add to the prompt something like (-10 through 10).

We could use

```
Console.Write(" ({0} through {1}) ", lowLim, highLim);
```

but we need the code twice, producing the same string each time. If you recall the *string.Format function*, then we can just create the string once, and use it twice. Here is a revised version, in example input_in_range2/input_in_range2.cs, without redundancy in the prompts in `Main`:

```
static void Main() //testing routine
{
   int n = PromptIntInRange("Enter a score: ", 0, 100);
   Console.WriteLine("Your score is {0}.", n);
   Console.WriteLine("Try another test.");
   n = PromptIntInRange("Enter a number: ", -10, 10);
   Console.WriteLine("Your number is {0}.", n);
}

/// Prompt the user to obtain an int until the response is in the
/// range [lowLim, highLim].  Then return the int in range.
/// Use the specified prompt, adding a reminder of the allowed range.
static int PromptIntInRange(string prompt, int lowLim, int highLim)
{
```

```
    string longPrompt = string.Format("{0} ({1} through {2}) ",
                                        prompt, lowLim, highLim);
    int number = UIF.PromptInt(longPrompt);
    while (number < lowLim || number > highLim) {
        Console.WriteLine("{0} is out of range!", number);
        number = UIF.PromptInt(longPrompt);
    }
    return number;
}
```

This time around we did the user input correctly, with the request for new input
repeated at the end of the loop. That repetition *is* easy to forget. Before we see
what happens when you forget, note:

> **Warning:** A `while` loop may be written so the continuation condition is
> *always* true, and the loop *never* stops by itself. This is an *infinite loop*. In
> practice, in many operating environments, particularly where you are getting
> input from the user, you can abort the execution of a program in an infinite
> loop by entering `Ctrl-C`.

In particular you get an infinite loop if you fail to get new input from
the user at the end of the loop.   The condition uses the bad original
choice forever. Here is the loop in the mistaken version, from example in-
put_in_range2_bad/input_in_range2_bad.cs:

```
int number = UIF.PromptInt(longPrompt);
while (number < lowLim || number > highLim) { // infinte loop!
    Console.WriteLine("{0} is out of range!", number);
    // number = UIF.PromptInt(longPrompt); //OMITS repeated prompt!
}
return number;
```

You can run the program. Remember `Ctrl-C` ! There are two tests in `Main`. If
you give a legal answer immediately in the first test, it works fine (never getting
into the loop body). If you give a bad input in the second test, you see that you
can never fix it! Remember `Ctrl-C` !

A more extreme abort is to close the entire console/terminal window running the
program.

6.3.1 Agree Function Exercise

Save example test_agree_stub/test_agree.cs in a project of your own.

Yes-no (true/false) questions are common. How might you write an input utility function `Agree`? You can speed things up by considering only the first letter of responses. Assume that it is important that the user makes a clear response: Then you should consider three categories of answer: ones accepted as true, ones accepted as false, and ambiguous ones. You need to allow for the possibility that the user keeps giving ambiguous answers for some time....

6.3.2 Interactive Sum Exercise

Write a program `sum_all.cs` that prompts the user to enter numbers, one per line, ending with a line containing 0, and keep a running sum of the numbers. Only print out the sum after all the numbers are entered (at least in your *final* version).

6.3.3 Safe Whole Number Input Exercise

Save example test_input_whole_stub/test_input_whole.cs as a project of your own. The code should test a function `PromptWhole`, as described below.

There is an issue with reading in numbers with the PromptInt function. If you make a typo and enter something that cannot be converted from a string to the right kind of number, a naive program will bomb. This is avoidable if you test the string and repeat the input if the string is illegal. Places where more complicated tests for illegality are needed are considered in *Safer PromptInt and PromptDouble Exercise* and *Safest PromptInt Exercise*. For now we just consider reading in whole numbers (integers greater than or equal to 0). Note that such a number is written as just a sequence of digits. Follow the interactive model of PromptIntInRange, looping until the user enters something that is legal: in this case, all digits.

The stub code already includes the earlier function `IsDigits`.

6.4 Short-Circuiting && and ||

Follow along with the following silly, but illustrative csharp sequence:

```
csharp> int x = 5, y = 2, z = 1;
csharp> y/x > z && x != 0;
false
csharp> x = 2; y = 5;
csharp> y/x > z && x != 0;
true
```

The compound condition includes x != 0, so what happens if we change x to 0 and try the condition again. Will you get false?

```
csharp> x = 0;
csharp> y/x > z && x != 0;
System.DivideByZeroException: Division by zero
...
```

No, one of the parts involves dividing by zero, and you see the result. What if we swap the two conditions to get the *logically equivalent*

```
csharp> x != 0 && y/x > z;
false
```

Something is going on here besides pure mathematical logic. Remember the final version in *IsDigits*. We did not need to continue processing when we knew the final answer already. The && and || operators work the same way, evaluating from left to right. If x != 0 is false, then x != 0 && y/x > z starts off being evaluated like false && ??. We do no need the second part evaluated to know the overall result is false, so C# *does not evaluate further*. This behavior has acquired the jargon *short-circuiting*. Many computer languages share this feature.

It also applies to ||. In what situation do you know what the final result is after evaluating the first condition? In this case you know:

```
true || ??
```

evaluates to true. Continuing with the same csharp sequence above (where x is 0, y is 5, and z is 1):

```
csharp> x == 0 || y/x > z;
true
```

The division by 0 in the second condition *never happens*. It is short-circuited.

For completeness, try the other order:

```
csharp> y/x > z || x == 0;
System.DivideByZeroException: Division by zero
...
```

This idea is useful in the `Agree` function, where you want to deal with the first character in the user's answer.

In situations where you want to test conditionThatWillBombWithBadData, you want to avoid causing an Exception. When there is good data, you want the result to actually come from conditionThatWillBombWithBadData. There are two cases, however, depending on what result you want if the data for this condition *is bad*, so you cannot evaluate it:

- If you want the result to be `false` with bad data for the dangerous condition, use

    falseConditionIfDataBad `&&` conditionThatWillBombWithBad-Data

- If you want the result to be `true` with bad data for the dangerous condition, use

    trueConditionIfDataBad `||` conditionThatWillBombWithBad-Data

6.5 While Examples

6.5.1 Bisection Method

For a very different example we look to scientific computing. In math class you likely learned various ways to find roots of functions f(x) exactly. In practice those methods almost never work beyond low order polynomials. Hence the best we can do usually is to approximate solutions numerically. One broadly useful approach

is the *bisection method*. You just need a continuous function (with an unbroken graph), and you need to first find two places, a and b, where f(a) and f(b) have opposite signs. If f is continuous and goes between positive and negative values, then it must cross 0 somewhere in between, and so there must be a real solution. The question is how to get close to a crossing point efficiently.

As an example we show $f(x) = x^2 - 2$, in the range from x = 0 to x = 2. As a first example we choose a simple function where the root can be figured symbolically, in this case the square root of 2. The figure below shows the graph, with extra horizontal and vertical lines that will will be explained.
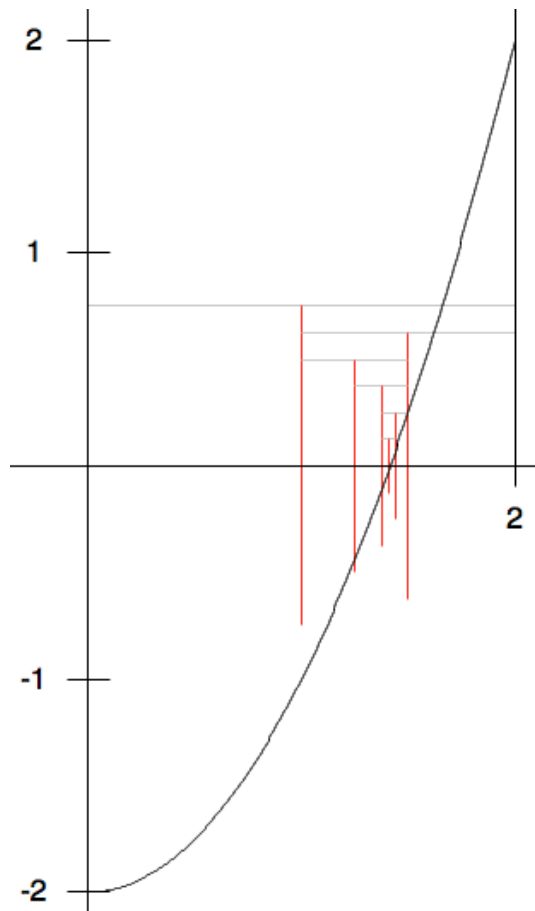
In the figure a= 0, f(0) < 0, b = 2, f(2) > 0.

The basic idea is to bisect the interval between a and b, finding the midpoint, c = (a+b)/2. If f(c) is 0, you are done. Otherwise f(c) has a sign which must be opposite *one* of f(a) and f(b).

In the figure the initial interval has the same x coordinates as for the top gray line. Its midpoint is at 1, the x coordinate of the red vertical segment coming down from the top gray line in the figure. Note f(1) < 0, the opposite sign of f(2), so we next consider the half of the original interval from the midpoint 1 to 2, with the next gray line marking this interval. The function still must cross 0 in the smaller interval because of the opposite signs for f on the endpoints. In the iterative procedure, you continue this process, halving the length of the interval, shifting one endpoint or the other to be the middle of the most recent interval, so for each interval, the signs of f on the two ends are opposite. Repeating this procedure, you can home in on as small an interval around a crossing point (root) as you like. The figure show this process for the first 5 steps, halving the interval length each time. You need to look at the output of the code to follow the results for even smaller intervals.

This approach always works, as long as the signs of f at the initial endpoints are distinct. Our `Bisection` functions check, and if this initial requirement is violated, the function returns the special double code value, `double.NaN`, meaning *not a number*.

There are other approaches to finding roots that may be faster when they work, but many of these methods can also have some chance of completely failing, so root finding algorithms generally have two extra parameters: a maximum number of iterations and a tolerance that indicates how close to a root is close enough.

In the `Bisection` function we use a and b as the endpoints of an interval and c as the midpoint. In each iteration the value of a or b is reset to be the previous midpoint value c. Of course a production version would not print out all the intermediate data, as the interval shrinks, but we do for illustration:

```csharp
public static double Bisection(double a, double b,
                               double tolerance, int iterations) {
   // check the preconditions for the method to work
   // a must be less than b so we can do the interval search
   if (a >= b)
      return double.NaN;
   Console.WriteLine ("a >= b ok");
```

```
    bool ok = false;

    // The function must cross the x-axis between the endpoints,
    //   meaning its sign changes.
    if ((f(a) < 0 && f(b) > 0)) {
       Console.WriteLine ("Test 1 passed");
       ok = true;
    }
    if ((f(a) > 0 && f(b) < 0)) {
       Console.WriteLine ("Test 2 passed");
       ok = true;
    }
    if (!ok)
       return double.NaN;
    int n=0;
    while (n < iterations) {
       double c = (a + b) / 2;
       Console.WriteLine ("a = {0}  b={1}", a, b);
       if (f(c) == 0 || (b - a)/2 < tolerance)
          return c;
       if (Math.Sign(f(c)) == Math.Sign(f(a)))
          a = c;
       else
          b = c;
       n++;
    }
    return double.NaN;
}
```

Since the bisection method always homes in on a real root rapidly, an alternate
version specifically for the bisection method finds the *best* approximation possible
with `double` arithmetic. While you can always halve an interval mathematically,
you eventually run out of distinct `double` values! We can stop when the midpoint
(calculated with limited `double` precision) is *exactly* the same as a or b:

```
/// This bisection method returns the best double approximation
/// to a root of f.  Returns double.NaN if the f(a)*f(b) > 0.
/// Does not require a < b.
public static double Bisection(double a, double b) {
   if (f(a) == 0)
      return a;
   if (f(b) == 0)
```

```
      return b;
   if (Math.Sign(f(b)) == Math.Sign (f(a)))   //or f(a)*f(b)>0
      return double.NaN;
   double c = (a + b) / 2;
   // If no f(c) is exactly 0, iterate until the smallest possible
   // double interval, when there is no distinct double midpoint.
   while (c != a && c != b) {
      Console.WriteLine ("a = {0}  b= {1}, diff = {2}", a, b, b-a);
      if (f(c) == 0)
         return c;
      if (Math.Sign(f(c)) == Math.Sign(f(a)))
         a = c;
      else
         b = c;
      c = (a + b) / 2;
   }
   return c;
}
```

C# remembers `double` values to more decimal places than it will actually display, so the second illustration also shows the difference between `a` and `b`, indicating the double values are still not really equal even after their displays match.

You can try this full example, bisection_method1/bisection_method1.cs. Note the special function checking for `double.NaN` in `Main`, because `double.NaN` is not equal to itself!

The current versions have a major limitation: They just work with the one canned version of the function `f` in the class. You need to edit the source code to use the same process with a different function! There are several ways around this using more advanced C# features. After the section *Interfaces*, a more flexible version should make sense, bisection_method/bisection_method.cs, explored further in *Bisection With Function Interface Exercise*. The more advanced version illustrates with the function in the initial version and several others, all using the same bisection function.

6.5.2 Savings Exercise

The idea here is to see how many years it will take a bank account to grow to at least a given value, assuming a fixed annual interest. Write a program

`savings.cs`. Prompts the user for three numbers: an initial balance, the annual percentage for interest as a decimal. like .04 for 4%, and the final balance desired. Print the initial balance, and the balance each year until the desired amount is reached. Round displayed amounts to two decimal places, as usual.

The math: The amount next year is the amount now times (1 + interest fraction), so if I have $500 now and the interest rate is .04, I have $500*(1.04) = $520 after one year, and after two years I have, $520*(1.04) = $540.80. If I enter into the program a $500 starting balance, .04 interest rate and a target of $550, the program prints:

```
500.00
520.00
540.80
563.42
```

6.5.3 Strange Sequence Exercise

Save the example program strange_seq_stub/strange_seq.cs in a project of your own.

There are three functions to complete. Do one at a time and test.

`Jump`: First complete the definitions of function `Jump`. For any integer n, `Jump(n)` is `n/2` if n is even, and `3*n+1` if n is odd. In the `Jump` function definition use an `if-else` statement. Hint [3]

`PrintStrangeSequence`: You can start with one number, say n = 3, and *keep* applying the `Jump` function to the *last* number given, and see how the numbers jump around!

```
Jump(3)  = 3*3+1 = 10;  Jump(10) = 10/2 = 5;
Jump(5)  = 3*5+1 = 16;  Jump(16) = 16/2 = 8;
Jump(8)  = 8/2   =  4;  Jump(4)  =  4/2 = 2;
Jump(2)  = 2/2   =  1
```

This process of repeatedly applying the same function to the most recent result is called function *iteration*. In this case you see that iterating the `Jump` function, starting from n=3, eventually reaches the value 1.

---

[3] If you divide an even number by 2, what is the remainder? Use this idea in your `if` condition.

It is an *open research question* whether iterating the Jump function from an integer
n will eventually reach 1, for *every* starting integer n greater than 1. Researchers
have only found examples of n where it is true. Still, no general argument has
been made to apply to the *infinite* number of possible starting integers.

In the PrintStrangeSequence you iterate the `Jump` function starting from param-
eter value n, as long as the current number is not 1. If you start with 1, stop
immediately.

`CountStrangeSequence`: Iterate the `Jump` function as in
`PrintStrangeSequence`. Instead of printing each number in the se-
quence, just count them, and return the count.

6.5.4 Roundoff Exercise II

Write a program to complete and test the function with this heading and docu-
mentation:

```
/// Return the largest possible number y, so in C#: x+y = x
/// If x is Infinity return Infinity.
/// If x is -Infinity, return double.MaxValue.
/// Assume x is not NaN (which is equal to nothing).
static double Epsilon(double x)
```

Hint: The non-exceptional case can have some similarity to the bisection in the
best root approximation example: start with two endpoints, a and b, where x+a
= x and x+b > x, and reduce the interval size by half....

6.6 More String Methods

Before we do more elaborate things with strings, some more string methods will
be helpful. Be sure you are familiar with the earlier discussion of strings in *Basic
String Operations*.

Play with the new string methods in csharp!

This variation of `IndexOf` has a second parameter:

int IndexOf(string target, int start) Returns the index of the beginning of the first occurrence of the string `target` in **this** string object, starting at index `start` or after. Returns -1 if `target` is not found. Example:

```
csharp> string state = "Mississippi";
csharp> print("01234567890\n"+state) // to see indices
01234567890
Mississippi
csharp> state.IndexOf("is", 0); // same as state.IndexOf("is");
1
csharp> state.IndexOf("is", 2);
4
csharp> state.IndexOf("is", 5);
-1
csharp> state.IndexOf("i", 5);
7
```

string Trim() Returns a string formed from **this** string object, but with leading and trailing whitespace removed. Example:

```
csharp> string s = "\n  123    ";
csharp> "#" + s + "#";
#
   123    #
csharp> "#" + s.Trim() + "#";
#123#
```

string Replace(string target, string replacement) Returns a string formed from **this** string by replacing all occurrences of the substring `target` by `replacement`. Example:

```
csharp> string s = "This is it!";
csharp> s.Replace(" ", "/");
"This/is/it!"
csharp> s.Replace("is", "at");
"That at it!"
csharp> "oooooh".Replace("oo", "ah");
"ahahoh"
```

bool StartsWith(string prefix) Returns `true` if **this** string object starts with string `prefix`, and `false` otherwise. Example:

6.6. More String Methods   **205**

```
csharp> "-123".StartsWith("-");
true
csharp> "downstairs".StartsWith("down");
true
csharp> "1 - 2 - 3".StartsWith("-");
false
```

bool EndsWith(string suffix) Returns `true` if **this** string object ends with string `suffix`, and `false` otherwise. Example:

```
csharp> "-123".EndsWith("-");
false
csharp> "downstairs".EndsWith("airs");
true
csharp> "downstairs".EndsWith("air");
false
```

6.6.1 Count Repetitions in a String Exercise

Write a program `test_count_rep.cs`, with a Main testing method, that tests a function with the following heading:

```
// Return the number of separate repetitions of target in s.
static int CountRep(string s, string target)
```

For example here is what `CountRep( "Mississippi", target)` would return with various values for `target`:

> `"i"`: 4
> `"is"`: 2
> `"sss"`: 0

Assume each repetition is completely separate, so `CountRep("Wheee!", "ee")` returns 1. The last two e's do not count, since the middle e is already used in the match of the first two e's.

6.6.2 Safer PromptInt and PromptDouble Exercise

Save the example safe_number_input_stub/safe_number_input.cs in a project of your own.

The idea is to write safe versions of the utility functions PromptInt and Prompt-Double (which can then be used in further places like PromptIntInRange).

Be sure you are familiar with *Safe Whole Number Input Exercise*, and the development of its `InputWhole` function.

A legal whole number string consists entirely of digits. We have already written example `IsDigits` to identify a string for a whole number.

The improvements to PromptInt and PromptDouble are very similar and straightforward *if* you have developed the two main Boolean support functions, `IsIntString` and `IsDecimalString` respectively.

A complicating issue with integer and decimal strings is that they may include parts other than digits. An integer may start with a minus sign. A decimal number can also contain a decimal point in an appropriate place. The suggestion is to confirm that these other characters appear in legal places, remove them, and see that what is left is digits. The recently introduced string methods should help....

Using the ideas above, develop the functions in order and test after each one: write `IsIntString`, revise `PromptInt`, write `IsDecimalString`, and revise `PromptDouble`.

Be sure to test carefully. Not only confirm that all appropriate strings return `true`: Also be sure to test that you return `false"` for *all* sorts of bad strings.

There is still one issue with `IsIntString` not considered yet: see the next exercise for the final improvement.

Hopefully you learned something from writing the earlier PromptWhole. Probably it is not worth keeping in our utility library any longer, since we have the more general and safe PromptInt, and we can restrict to many ranges with PromptIntIn-Range.

We will arrange for these functions to be a library class later. For now just develop and test them in this one class.

6.6.3 Safest PromptInt Exercise

With the suggestions so far the in the previous exercise, `IsIntString` will catch a strange stray character, and be sure that the string for an *integer* is en-

tered, but an `int` is not an arbitrary integer: it has limited range, between `int.MinValue` = -2147483648 and `int.MaxValue` = 2147483647.

Revise the `IsIntString` function of the previous exercise so that it checks that the result is in range, too, allowing the PromptInt function to be totally reliable.

There is a problem: your current version of IsIntString is likely to accept a string like `"9876543210"`, and you cannot convert it to an `int` to do the comparison to see that it is in fact too large for an `int`! Catch 22?

There is an alternate approach involving comparing strings, not numbers.

There is a string instance method:

```
public int CompareTo(string t)
```

It does roughly lexicographical string comparisons, so

```
s.CompareTo(t) <= 0
```

is true when s "comes before" t or is equal to t. This works with alphabetizing letter strings: "at" comes before "ate" which comes before "attention" which comes before "eat". It also works with digit strings *of the same length* to give the same relationship as the corresponding numbers:

```
"123456890".CompareTo("2147483647") <= 0
```

is true, and

```
"9876543210".CompareTo("2147483647") <= 0
```

is false.

This idea can be leveraged into a completely reliable version of IsIntString. (With this approach you could also create an IsLongString very similarly, but we skip it since it teaches you nothing new.)

The idea of a `CompareTo` method is much more general and is used much later in *Rationals Revisited*.

6.7 User Input: UI

With the exercises in the last section, we have all we need for a much improved User Input library class.   This will be the class `UI`. We have the original `PromptLine`, improved `PromptInt`, `PromptDouble`, `PromptIntInRange`, `PromptDoubleInRange`, `Agree`, and assorted supporting functions. The only changes to individual methods were to make sure that the static methods are public.

Henceforth we will be using `UI` in place of `UIF`. In fact all the places `UIF` was used before could be replaced by `UI`: It includes all the functionality of `UIF`.

You can look at the code all together in ui/ui.cs.

There are even fancier ways of arranging for legal numeric input. We have only been reading whole lines, but it is possible to read individual characters with `Console.ReadKey`, without a newline being entered yet. A much more extensive advanced subject is the special regular expression language for describing arbitrary patterns in strings using the `Regex` class. Though we will not discuss the details, another slick replacement for UI using these features is in the example class uifnt/UIFNT.cs.

6.8 Greatest Common Divisor

6.8.1 Euclid's Algorithm

The greatest common divisor of two non-zero integers is a great example to illustrate the power of loops. Everyone learns about the *concept* of a greatest common divisor when faced with a fraction that is not in *reduced* form.

Consider the fraction $\frac{2}{4}$, which is the same as $\frac{1}{2}$. The fraction $\frac{2}{4}$ can be reduced, because the numerator and denominator both have greatest common factor of 2. That is, $\frac{2}{4} = \frac{1\cdot2}{2\cdot2}$. So the factor of 2 can be canceled from both the numerator and the denominator.

Euclid (the mathematician from classic times and author of *Elements*) is credited with having come up with a clever algorithm for how to compute the greatest common divisor efficiently. It is written as follows, where $a \bmod b$ means `a %`

`b` in C#.

$$gcd(a, b) = gcd(b, a \bmod b)$$
$$gcd(a, 0) = a$$

It is common in mathematics to list functions as one or more *cases*. The way you read this is as follows:

- In general, the greatest common divisor of `a` and `b` is the same as computing the greatest common divisor of `b` and the remainder of `a` divided by `b`.

- In the case where `b` is zero, the result is `a`. This makes sense because `a` divides itself and 0.

To gain some appreciation of how the definition *always* allows you to compute the greatest common divisor, it is worthwhile to try it out for a couple of numbers where you *know* the greatest common divisor. For example, we already know that the greatest common divisor of 10 and 15 is 5. Let's use Euclid's method to verify this:

- $gcd(10, 15) = gcd(15, 10 \bmod 15) = gcd(15, 10)$

- $gcd(15, 10) = gcd(10, 15 \bmod 10) = gcd(10, 5)$

- $gcd(10, 5) = gcd(5, 10 \bmod 5) = gcd(5, 0)$

- $gcd(5, 0) = 5$

Notice that in the example above, the first number (10) was smaller than the second (15), and the first transformation just swapped the numbers, so the larger number was first. Thereafter the first number is always larger.

6.8.2 GCD "Brute Force" Method

Now that we've gotten the preliminaries out of the way and have a basic mathematical explanation for how to calculate the greatest common divisor, we'll take a look at how to translate this into code using the machinery of while loops that you've recently learned.

The way GCD is formulated above is, indeed, the most clever way to calculate the greatest common divisor. Yet the way we learn about the greatest common divisor in elementary school (at least at first) is to learn how to factor the numbers a and b, often in a brute force way. So for example, when calculating the greatest

common divisor of 10 and 15, we can immediately see it, because we know that both of these numbers are divisible by 5 (e.g. 5 * 2 = 10 and 5 * 3 = 15). So the greatest common divisor is 5.

But if we had something more tricky to do like 810 and 729, we might have to think a bit more.

Before we learn to find the factors of numbers, we will often just "try" numbers until we get the greatest common divisor. This sort of trial process can take place in a loop, where we start at 1 and end at min(a, b). Why the minimum? We know that none of the values after the minimum can divide both a and b (in integer division), because no larger number can divide a smaller positive number. The smaller number would be the (non-zero) remainder.

Now take a look at a basic version of GCD:

```
/// Return the greatest common divisor of positive numbers.
public static int GreatestCommonDivisor (int a, int b)
{
    int n = Math.Min (a, b);
    int gcd = 1, i = 1;

    while (i <= n) {
        if (a % i == 0 && b % i == 0) {
            gcd = i;
        }
        i++;
    }
    return gcd;
}
```

This code works as follows:

- We begin by finding `Math.Min(a, b)`. This is how to compute the minimum of any two values in C#. Technically, we don't need to use the minimum of a and b, but there is no point in doing any more work than necessary.

- We'll use the variable `i` as the loop index, starting at 1.

- The variable `gcd` will hold the largest currently known common divisor. We start with 1, which divides any integer, and we will look for a higher value that also divides a and b.

- The line `while (i <= n)` is used to indicate that we are iterating the values of `i` until the minimum of `a` and `b` (computed earlier) is reached.

- The line `if (a % i == 0 && b % i == 0)` is used to check whether we have found a new value that replaces our previous *candidate* for the GCD. A value can only be a candidate for the GCD if it divides a and b without a remainder. The modulus operator `%` is our way of determining whether there is a remainder from the division operation `a / i` or `b / i`.

- The line `i++` is our way of going to the next value of `i` to be tested as the new GCD.

- When this loop terminates, the greatest common divisor has been found.

So this gives you a relatively straightforward way of calculating the greatest common divisor. While simple, it is not necessarily the most efficient way of determining the GCD. If you think about what is going on, this loop could run a significant number of times. For example, if you were calculating the GCD two very large numbers, say, one billion (1,000,000,000) and two billion (2,000,000,000) it is painfully evident that you would consider a large number of values (a billion, in fact) before obtaining the candidate GCD, which we know is 1,000,000,000.

Brute-Force GCD Exercise

The code above goes though all integers 2 through `min(a, b)`. That is not generally necessary when the GCD is greater than 1, even with a brute-force mindset. Write a `g_c_d_basic_faster.cs` to do this with a slightly different `GreatestCommonDivisor` function. [4]

6.8.3 GCD Subtraction Method

The subtraction method (also attributable to Euclid) to compute the Greatest Common Divisor works as follows:

---

[4] The original brute-force gcd approach always goes through all the integers between 1 and `min(a, b)`. There is a way to stop the first time the real gcd is reached. How can you arrange that?

- Based on the *mathematical* definition in the previous section, the greatest common divisor algorithm saves a step when we already have a and b in the *right order*.

- The *right order* means that $a > b$. As we noted earlier, the cleverness of the *mathematical* definition is that a and b are swapped as the first step to ensure that $a > b$, after which we can repeatedly divide to get the GCD.

- Division, of course, is a form of repetitive subtraction, so the way to divide by b is to repeatedly subtract it (from a) until a is no longer greater than b.

- The subtraction method basically makes no attempt to put a and b in the right order. Instead, we just write similar loops to allow for the possibility of either order.

- A simple check must be performed to ensure that the approach of repeated subtraction actually resulted in the GCD. This will happen if a and b bump into one another, thereby meaning that we have computed the GCD.

```
1   public static int GreatestCommonDivisor (int a, int b)
2   {
3      int c;
4      while (a != b) {
5         while (a > b) {
6            c = a - b;
7            a = c;
8         }
9         while (b > a) {
10           c = b - a;
11           b = c;
12        }
13     }
14     return a;
15  }
```

A look at the source code more or less follows the above explanation.

Let's start by looking at the inner loop at line 5, while (a > b). In this loop, we are repeatedly subtracting b from a, which we know we can do, because a started out as being larger than b. At the end of loop a is reduced to either

1. b, in which case b exactly divided the earlier a, and b is the GCD, or

2. a number less than b, namely $a \bmod b$ (or in C# terms a % b), and the

---

6.8. Greatest Common Divisor 213

process continues....

The loop on line 9 is similar to the loop in line 5. For the same reasons as we already explained, b ends up equal to a, which is the GCD, or b ends up as $b \bmod a$.

As discussed above, if a and b end up as the same number, that is the GCD. On the other hand, the first GCD algorithm example showed how remainders may need to be to be calculated over and over. The outer loop in this version keeps this up until a and b are reduced to equal values. At this point the inner loops would make no further changes, and the common value is the GCD.

As an exercise to the reader, you may want to consider adding some `Console.WriteLine()` statements to print the values of a and b within each loop, and after both loops have executed. It will allow you to see in visual terms how this method does its work.

6.8.4 GCD Remainder Loop

There are several ways to code the shorter Euclidean algorithm at the beginning of this GCD section. It is a repetitive pattern, and a loop can be used. There are two parameters, a and b, to the gcd, and they can be successively changed, suggesting a loop. What is the continuation condition? You stop when b is 0, so you continue while `b != 0`. The parameters a and b need to be replaced by b and `a % b`. One extra variable needs to be introduced to make this double change work. The simplest is to introduce a variable r for the remainder. Check and see for yourself that you need an extra variable like r:

```csharp
/// Return the greatest comon divisor of nonnegative numbers,
/// not both 0.
public static int GreatestCommonDivisor (int a, int b)
{
   while (b != 0) {
      int r = a % b;
      a = b;
      b = r;
   }
   return a;
}
```

More verbose demonstration code, that prints the progress each time through loop is in g_c_d_remainder_loop/g_c_d_remainder_loop.cs.

6.8.5 Preview: Recursive GCD

The first statement of Euclid's algorithm said (in C#) when

```
gcd(a, b) = gcd(b, a % b)
```

It is saying the result of the function with one set of parameters is equal to calling the function with another set of parameters. If we put this into a C# function definition, it would mean the instructions for the function say to *call itself*. This is a broadly useful technique called *recursion*, where a function calls *itself* inside its definition. We don't expect you to master this technique immediately but do feel that it is important you at least *hear* about it and see its tremendous power:

```csharp
/// Return the greatest comon divisor of nonnegative numbers,
/// not both 0.
public static int GreatestCommonDivisor (int a, int b)
{
    if (b == 0) {                                  // base
        return a;                                  //   case
    } else {
        return GreatestCommonDivisor (b, a % b); // recursion
    }
}
```

- Recalling our earlier definition, the case $gcd(a, 0) = a$ is handled directly by lines commented as "base case".

- And the case $gcd(a, b) = gcd(b, a \bmod b)$ is handled by line with comment "recursion", with the function calling itself.

In g_c_d_euclid_recursive/g_c_d_euclid_recursive.cs is a wordier demonstration version that prints to the screen the progress at each recursive call.

The recursive version of the gcd function *refers to itself* by *calling* itself. Though this seems circular, you can see from the examples that it works very well. The important point is that the calls to the same function are not completely the same: *Successive* calls have *smaller* second numbers, and the second number eventually

reaches 0, and in that case there is a direct final answer. Hence the function is not really circular.

This recursive version is a much more direct translation of the original mathematical algorithm than the looping version!

The general idea of recursion is for a function to call itself with *simpler* parameters, until a simple enough place is reached, where the answer can be directly calculated.

6.9 Do-While Loops

Suppose you want the user to enter three integers for sides of a right triangle. If they do not make a right triangle, say so and make the user try again.

One way to look at the while statement rubric is:

```
set data for condition
while (condition) {
    accomplish something
    set data for condition
}
```

As we have pointed out before this involves setting data in two places. With the triangle problem, three pieces for data need to be entered, and the condition to test is fairly simple. (In any case the condition could be calculated in a function.)

A `do-while` loop will help here. It tests the condition at the end of the loop, so there is no need to gather data before the loop:

```csharp
int a, b, c;
do {
    Console.WriteLine("Think of integer sides for a right triangle.");
    a = UI.PromptInt("Enter integer leg: ");
    b = UI.PromptInt("Enter another integer leg: ");
    c = UI.PromptInt("Enter integer hypotenuse: ");
    if (a*a + b*b != c*c) {
        Console.WriteLine("Not a right triangle: Try again!");
    }
} while (a*a + b*b != c*c);
```

The general form of a `do-while` statement is

```
do {
    statement(s)
} while ( continuationCondition );
```

Here the block of statement(s) is *always* executed at least once, but it continues to be executed in a loop only so long as the condition tested after the loop body is true.

Note: A `do-while` loop is the *one* place where you *do* want a semicolon right after a condition, unlike the places mentioned in *Dangerous Semicolon*. At least if you omit it here you are likely to get a compiler error rather than a difficult logical bug.

A `do-while` loop, like the example above, can accomplish exactly the same thing as the `while` loop rubric at the beginning of this section. It has the general form:

```
do {
    set data for condition
    if (condition) {
        accomplish something
    }
} while (condition);
```

It only sets the data to be tested *once*. (The trade-off is that the condition is tested *twice*.)

6.9.1 Loan Table Exercise

Loans are common with a specified interest rate and with a fixed periodic payment. Interest is charged at a fixed rate on the amount left in the loan after the last periodic payment (or start of the loan for the first payment).

For example, if an initial $100 loan is made with 10% interest per pay period, and a regular $20 payment each pay period: At the time of the first payment interest of $100*.10 = $10 is accrued, so the total owed is $110. Right after the payment of $20, $110 - $20 = $90 remains. That $90 gains interest of $90*.10 = $9 up to the next payment, when $90 + $9 = $99 is owed. After the regular payment of

---

$20, $99 - $20 = $79 is left, and so on. When a payment of at most $20 brings the amount owed to 0, the loan is done.

We can make a table showing

- Payment number (starting from 1)

- The principal amount after the previous payment (or the beginning of the loan for the first payment)

- The interest on that principal up until the next periodic payment

- The payment made as a result.

Continuing the example above, the whole table would look like:

```
Number Principal   Interest    Payment
    1     100.00      10.00      20.00
    2      90.00       9.00      20.00
    3      79.00       7.90      20.00
    4      66.90       6.69      20.00
    5      53.59       5.36      20.00
    6      38.95       3.90      20.00
    7      22.85       2.29      20.00
    8       5.14       0.51       5.65
```

In the final line, the principal plus interest equal the payment, finishing off the loan.

Similarly, with a $1000.00 starting loan, 5% interest per pay period, and $196 payments due, we would get

```
Number Principal   Interest    Payment
    1    1000.00      50.00     196.00
    2     854.00      42.70     196.00
    3     700.70      35.04     196.00
    4     539.74      26.99     196.00
    5     370.73      18.54     196.00
    6     193.27       9.66     196.00
    7       6.93       0.35       7.28
```

If a $46 payment were specified, the principal would not decrease from the initial amount, and the loan would never be paid off.

There are a couple of wrinkles here: `double` values do not hold decimal val-

ues exactly. The `decimal` type does hold decimal numbers exactly (and in an enormous range, see *Numeric Types and Limits*) and hence are beter for monetary calculations. Decimal literals end with m, like `34.56m` for *exactly* 34.56.

Though decimals are exact, money only has two decimal places. We make the assumption that interest will always be calculated as current principal*rate, rounded to two decimal places: `Math.Round(principal*rate, 2)`.

Write `loan_calc.cs`, completing `LoanTable` and write a `Main` testing program:

```
/// Print a loan table, showing payment number, principal at the
/// beginning of the payment period, interest over the period, and
/// payment at the end of the period.
/// The principal is the initial amount of the loan.
/// The rate is fraction representing the rate of interest per PAYMENT.
/// The periodic regular payment is also specified.
/// If the payment is insufficient, merely print "payment too low".
public static void LoanTable(decimal principal, decimal rate,
                             decimal payment)
```

Note that the `rate`, too, is a `decimal`, even though it does not represent money. That is important, because arithmetic with a `decimal` and a `double` is forbidden: A `double` would have to be explicitly cast to a `decimal`. Insisting on `decimal` parameter simplifies the function code.

This exercise is much more sophisticated than the *Savings Exercise*, so it is placed in this section, much later in the chapter. Use what ever form of loop makes the most sense to you.

6.10 Number Guessing Game Lab

Objectives:

- Work with functions
- Work with interactive while loops
- Use decisions
- Introduce random values

This lab is inspired by a famous children's game known as the number-guessing game. We suppose two people are playing.

The rules are:

- Person A chooses a positive integer less than N and keeps it in his or her head.

- Person B makes repeated guesses to determine the number. Person A must indicate whether the guess is higher or lower.

- Person A must tell the truth.

So as an example:

- George and Andy play the game.

- George chooses a positive number less than 100 (29) and puts it in his head.

- Andy guesses 50. George says "Lower". Andy now knows that $1 \leq number < 50$.

- Andy guesses 25. George says "Higher". Andy now knows that $26 \leq number < 50$.

- Andy guesses 30. George says "Lower". Andy now knows that the $26 \leq number < 30$.

- Andy starts thinking that he is close to knowing the correct answer. He decides to guess 29. Andy guesses the correct number. So George says, "Good job! You win."

We are going to elaborate this game in small steps. You might save the intermediate versions under new names.

The computer code for the game is going to be acting like Player A.

6.10.1 Part 1: No Hints; Fixed Secret Number

You will want to use the UI class, so either copy ui.cs into your project, or (for Xamarin Studio) create a new project in a solution in which you already have added the ui library project, and add the ui project as a *reference* for the lab project. Make sure your program has `namespace IntroCS;` to match the UI class.

You are going to play a game, and later may repeat it, so put the code for playing the number game in a function called `Game`:

```
static void Game()
```

For now your write a `Main` function to just call `Game()`.

In `Game`:

1. For the simplest versions, which help testing, have the program assign a specific secret number (like 29), and call it `secret`. Admittedly, this is not much fun for the player the second time!

2. Prompt the player for a guess. Use `UI.PromptInt`. Every time the player guesses wrong, print "Wrong!". A later version will give clues. Keep prompting for another number until the player guesses correctly. (Since you, the programmer, knows the secret number, this need not go on forever.)

3. When the player guesses the right number, print "Correct! You win!"

Sample play could look like:

> Guess the number: **55**
> Wrong!
> Guess the number: **12**
> Wrong!
> Guess the number: **29**
> Good job! You win!

You could also make the game stop immediately, (since you know the secret number):

> Guess the number: **29**
> Good job! You win!

6.10.2 Part 2: Add Hints

In `Game`: Instead of just printing "Wrong!" when the player is incorrect, print "Lower!" or "Higher!" as appropriate. For example:

> Guess the number: **55**

Lower!

Guess the number: **12**

Higher!

Guess the number: **25**

Higher!

Guess the number: **29**

Good job! You win!

6.10.3 Part 3: Add a Random Secret Number

In `Game`, make the following alterations and additions:

1. For now set an `int` variable `big` to 100. We will make sure the secret number is less than `big`.

2. Have the Game function print "In this game you guess a positive number less than 100." For future use it is best if you have the printing statement reference the variable `big`, rather than the literal `100`.

1. Thus far the secret number was fixed in the program. Now we are going to let it vary, by having the game generate a *random* number. For your convenience, we are going to give you the C# code to compute the random number. Assuming we want a secret number so $1 \leq secret < big$, we can use the code:

```
Random r = new Random();
int secret = r.Next(1, big);
```

In case you are wondering, we are creating a *new object* of the *class* `Random` which serves as the random number generator. We'll cover this in more detail when we get to the *Classes and Object-Oriented Programming* chapter. Here is some illustration using a `Random` object in csharp. Your answers will not be the same!

```
csharp> Random r = new Random();
csharp> r.Next(1, 100);
55
csharp> r.Next(1, 100);
31
csharp> r.Next(1, 100);
```

```
79
csharp> r.Next(2, 5);
2
csharp> r.Next(2, 5);
4
csharp> r.Next(2, 5);
3
csharp> r.Next(2, 5);
3
```

In general the minimum possible value of the number returned by `r.Next` is the first parameter, and the value returned is always *less* than the second parameter, *never equal*.

You can see that `r.Next()` is smart enough to give what appears to be a randomly chosen number every time.

Example (where `secret` ended up as 68):

> Guess a number less than 100!
> Guess the number: **60**
> Higher!
> Guess the number: **72**
> Lower!
> Guess the number: **66**
> Higher!
> Guess the number: **68**
> Good job! You win!

For debugging purposes, you might want to have `secret` be printed out right away. (Eliminate that part when everything works!)

6.10.4 Part 4: Let the Player Set the Range of Values

In `Game`: Instead of declaring `big` and automatically initializing it to 100, make `big` be a parameter, so the heading looks like:

```
static void Game(int big)
```

In `Main`:

1. Prompt the player for the limit on the secret number. An exchange might look like:

   Enter a secret number bound: **10**

2. Pass the value given by the player to the `Game` function (so it will be `big` inside `Game`).

Hence the program might start with:

Enter a secret number bound: **10**
In this game you guess a number less than 10!
Guess the number: **5**
Higher!
Guess the number: **7**
Lower!
Guess the number: **6**
Good job! You win!

6.10.5 Part 5: Count the Guesses

In `Game`: When the player finally wins, print the number of guesses the player made. For example, for the game sequence shown above, the last line would become:

Good job! You win on guess 3!

You need to keep a count, adding 1 with each guess.

6.10.6 Possible Extra Credit Improvements or Variations

Should you finish everything early, try the following:

1. **(20% extra credit)** In `Main`:

   Use an outer `while` loop to allow the game to be played repeatedly. Change the prompt for the bound in `Main` to:

   Enter a secret number bound (or 0 to quit):

   Continue to play games until the player enters 0 for the bound.

2. **(40% extra credit)** In `Main` prompt users to see if they want to guess numbers or reverse roles and choose the secret number. In the first case, just call the existing Game function. In the second case you need a new function, where the user is the one who knows the secret number and the computer guesses numbers until the answer is obtained. Write and use a new function

```
static void GameReversed(int big)
```

Pass it the parameter `big`, still set in `Main`. The new `GameReversed` will tell the user to put a number in his/her head, and press return to continue. (You can throw away the string entered - this is just to cause a pause.) Then the computer guesses. For simplicity let the human enter "L" for lower, "H" for higher, and "E" for equal (when the computer wins). As you saw in the initial example with George and Andy, each hint reduces the range of the possible secret numbers. Have the computer guess a *random* number in the *exact* range that remains possible.

To do this you must note the asymmetry of the parameters for the method `Next`: suppose `n = r.Next(low, higher)`, then

$$low \leq number < higher$$

The first parameter *may* be returned, but second parameter is *never* returned.

You will need two variables `low` and `higher` that keep bracketing the allowed range. The simplest thing is to set them so they will be the parameters for the following call to `Next`.

That would mean initially `low` is 1 and `higher` is equal to `big`. With each hint you adjust one or the other of `low` and `higher` so they get closer together. The game ends after the human enters "E".

Have the computer complain that the human is cheating (and stop the game) if the computer guesses the only possible value, and the human does *not* respond with "E".

6.11 Chapter Review Questions

1. While loops are a very important part of your programming tools. Put in your own words: when should you think to use a while loop?

2. Loops are also among the hardest things for many students – with lots of things to think about. There is a sequence of general process questions that you can ask yourself to help you organize your work. What are they in your words? Do you know them well, or have them written down in a place you can easily jump to?

3. Compare do-while and while loops: How do you think about which one to use?

4. In an interactive while loop you need to continuously get data from the user. Where do you generally put the code to get more data?

5. In general, what causes an infinite `while` loop?

6. What is wrong with this statement: When the condition in a `while` loop heading becomes false, the loop statement immediately terminates.

7. A `while` loop will terminate when the program evaluates the condition in its heading and the value becomes false. What is the important difference in this statement from the previous incorrect statement?

8. When does a program evaluate the condition in a `while` loop heading? (There are two situations.)

9. A `while` loop is generally terminated when the program evaluates the condition in its heading and it becomes false. How else can a program exit from a `while` loop?

10. We generally construct a loop so its body is a compound statement, composed of a sequence of statements inside. If this body is a sequence of simple statements, does it make sense for one of them to be a return statement?

11. When inside a loop, a return statement should generally only appear as a *sub-statement* of what kind of statement?

12. Which of these conditions is safer in general, with *arbitrary* `string s` and `int i`?

```
s[i] != '#' && i >= 0 && i < s.Length

i >= 0 && i < s.Length && s[i] != '#'
```

13. What is printed?

```
//          012345678901234567890
string s = "Is coding cool?  Yes!"
Console.WriteLine(s.Trim());
string t = s.Substring(9, 8);
Console.WriteLine(t.Replace(" ", "/"));
Console.WriteLine(t.Trim().Replace(" ", "/"));
Console.WriteLine(s.StartsWith("is"));
Console.WriteLine(s.ToLower().StartsWith("is"));
int i = s.IndexOf("co"), j = s.IndexOf("co", i+1),
    k = s.IndexOf("co", j+1);
Console.WriteLine(i + " " + j + " " + k);
```

FOREACH LOOPS

7.1 `foreach` Syntax

These sections on `foreach` loops and the later *For Loops* introduce new looping statements. Neither is absolutely necessary: You could do all the same things with `while` loops, but there are many situations where `foreach` loops and `for` loops are more convenient and easier to read.

A `foreach` statement only works with an object that holds a sequence or collection. We will see many more kinds of sequences later. For now we can illustrate with a string, which is a sequence of characters.

We have already processed strings a character at a time, with `while` loops. We took advantage of the fact that strings could be indexed. Our `while` loops directly controlled the sequence of indices. Then we could look up the character at each index of a given string `s`:

```
int i = 0;
while (i < s.Length) {
   use value of s[i]...
   i++;
}
```

Examples have been in *While-Statements with Sequences*, like

```
      // Print the characters of s, one per line.
      static void OneCharPerLine(string s)
      {
```

```
        int i = 0;
        while (i < s.Length) {
            Console.WriteLine(s[i]);
            i++;
        }
    }
```

In this example we really only care about the characters, not the indices. Managing the indices is just a way to get at the underlying sequence of characters.

A conceptually simpler view is just:

```
for each character in s
    use the value of the character
```

To *use* "the character" in C#, we must be able to refer to it. We might name the current character `ch`. The following is a variant of `OneCharPerLine` with a `foreach` loop:

```
static void OneCharPerLine(string s)
{
    foreach (char ch in s) {
        Console.WriteLine(ch);
    }
}
```

That is all you need! The `foreach` heading feeds us one character from `s` each time through, using the name `ch` to refer to it. Of course any new variable name must be declared in C#, so `ch` is preceded in the heading by its type, `char`. Then we can use `ch` inside the body of the loop. Advancing to the next element in the sequence is automatic in the next time through the loop. No `i++` to remember; no possibility of an infinite loop!

The general syntax of a `foreach` loop is

> foreach (**type itemName** in *sequence*) {
>
>   statement(s) using **itemName**
>
> }

Here is a version of IsDigits:

```
static Boolean IsDigits(string s)
{
```

```
    foreach (char ch in s) {
        if (ch < '0' || ch > '9') {
            return false;
        }
    }
    return (s.Length > 0);
}
```

See the advantages of `foreach` in these examples:

- They are more concise than the indexing versions.

- They keep the emphasis on the characters, not the secondary indices.

- The `foreach` heading emphasizes that a particular sequence is being processed.

> **Warning:** *If* you have explicit need to refer to the indices of the items in the sequence, then a `foreach` statement does not work.

Of course you can refer to the indices of items in sequence with a flexible `while` loop, or see *For Loops*, coming soon....

7.2 `foreach` Examples

In *IsDigits* we use the underlying int Unicode value of the characters in comparisons. When printing, you cannot see this code directly, since the `char` type prints as *characters*! To see the underlying code value for a character, ch, it can be cast to an int: `(int)ch`

We can easily write a loop to print the unicode value of each character in a string, s. We do not need indices here, so a `foreach` loop is appropriate:

```
foreach (char ch in s) {
    Console.WriteLine("Unicode for {0} is {1}.", ch, (int)ch);
}
```

Try this in csharp.

We will have many more examples after we introduce more kinds of sequences.

7.3 Chapter Review Questions

1. Describe in general when a `foreach` loop is going to be easier to use than a `while` loop.

2. Even if you want to process every element of a sequence, what would keep you from using a `foreach` loop?

FOR LOOPS

8.1 For-Statement Syntax

We now introduce the last kind of loop syntax: `for` loops.

A `for` loop is an example of *syntactic sugar*: syntax that can simplify things for the programmer, but can be immediately translated into an equivalent syntax by the compiler. For example:

```
for (i = 2; i <= n; i++) {
    sum = sum + i;
}
```

is exactly equivalent to this code similar to part of *SumToN*:

```
i= 2;
while (i <= n) {
    sum = sum + i;
    i++;
}
```

More generally:

```
for ( initialization ; condition ; update ) {
        statement(s)
    }
```

translates to

initialization ;
while (*condition*) {
    statement(s)
    update ;
}

In the example above, *initialization* is i=2, *condition* is i <= n, and *update* is i++.

Why bother with this rearrangement? It is a matter of taste, but the heading:

```
for (i = 2; i <= n; i++) {
```

puts all the information about the variable controlling the loop into one place at the top, which may help quickly visualize the overall sequence in the loop. If you use this format, and get used to the three parts you are less likely to forget the i++ than when it comes tacked on to the end of a while loop body, after all the specific things you were trying to accomplish.

Although the for loop syntax is very general, a strongly recommended convention is to only use a for statement when all the control of variables determining loop repetition are in the heading.

For example if a for loop uses i in the heading, i can have a value assigned or reassigned in the heading, but should *not* have its value modified anywhere inside the loop body. If you want more complicated behavior, use a while loop.

A for loop can also include variable declaration in the initialization, as in:

```
for (int i = 2; i <= n; i++) {
    sum = sum + i;
}
```

This is close, but not quite equivalent to:

```
int i = 2;
while (i <= n) {
    sum = sum + i;
    i++;
}
```

Variables declared in a for loop heading are local to the for loop heading and body. The variable i declared before the while statement above is still defined

after the `while` loop.

The two semicolons are always needed in the `for` heading, but any of the parts they normally separate may be omitted. If the condition part is omitted, the condition is interpreted as always true, leading to an infinite loop, that can only terminate due to a `return` or *break statement* in the body.

Note the different parts of the heading used at different times (consistent with the positions in the corresponding while loop):

- When starting the whole statement, the initialization is done, and then the test.

- After finishing the body and returning to the heading, the update operations are done, followed by the test.

Other variations

As in a regular local variable declaration, there may be several variables of the same type initialized at the beginning of a `for` loop heading, separated by commas. Also, at the end of the `for` loop heading, the update portion may include more than one expression, separated by commas. For example:

```
for (int i = 0, j = 10; i < j; i = i+2, j++) {
    Console.WriteLine("{0} {1}", i, j);
}
```

Guess what this does, and then paste it into csharp to check.

The comma separated lists in a `for` statement heading are mentioned here for completeness. Later we will find a situation where this is actually useful.

8.1.1 Break and Continue

This section concerns special *break* and *continue* statements that can *only* occur inside a loop (any kind: `while`, `for` or `foreach`). The syntax is convenient in various circumstances, but not necessary. You are free to use it, but for this course it is an *optional extra*:

You can already stop a loop in the middle with an `if` statement that leads to a choice with a `return` statement. Of course that forces you to completely leave the current function. If you only want to break out of the *innermost current loop*, but *not* out of the whole function, use a break statement:

```
      break;
```

in place of return. Execution continues after the end of the whole innermost currently running loop statement. The `break` and `continue` statements only make practical sense inside of an `if` statement that is inside the loop.

Examples, assuming `target` already has a string value and `a` is an array of strings:

```csharp
bool found = false;
for (int i = 0; i < a.Length; i++) {
   if (a[i] == target) {
      found = true;
      break;
   }
}
if (found) {
   Console.WriteLine("Target found at index " + i);
} else {
   Console.WriteLine("Target not found");
}
```

When an element is reached that matches `target`, execution goes on *past the loop* with `if (found)` ....

An alternate implementation with a compound condition in the heading and no `break` is:

```csharp
bool found = false;
for (int i = 0; i < a.Length && !found; i++) {
   if (a[i] == target) {
      found = true;
   }
}
if (found) {
   Console.WriteLine("Target found at index " + i);
} else {
   Console.WriteLine("Target not found");
}
```

With a `foreach` loop, which has no explicit continuation condition, the `break` would be more clearly useful. Here is a variant if you do not care about the specific location of the target:

```
bool found = false;
foreach (string s in a) {
   if (s == target) {
      found = true;
      break;
   }
}
if (found) {
   Console.WriteLine("Target found");
} else {
   Console.WriteLine("Target not found");
}
```

Using break statements is a matter of taste. There is some advantage in reading and following a loop that has only one exit criteria, which is easily visible in the heading. On the other hand, in many situations, using a break statement makes the code much less verbose, and hence easier to follow. If you *are* reading through the loop, it may be clearer to have an immediate action where it is certain that the loop should terminate.

All the modifiers about *innermost* loop are important in a situation like the following:

```
for (....) {
   for (....) {
      ...
      if (...) {
         ...
        break;
      }
      ...
   }
}
```

The break statement is in the inner loop. If it is reached, the inner loop ends, but the inner loop is just a single statement inside the outer loop, and the outer loop continues. If the outer loop continuation condition remains true, the inner loop will be executed again, and the break may or may not be reached that time, so the inner loop may or may not terminate normally....

For completeness we mention the much less used continue statement:

```
    continue;
```

It does not break out of the whole loop: It just skips the rest of the *body* of the innermost current loop, *this time* through the loop. In the simplest situations a `continue` statement just avoids an extra `else` clause. It can considerable shorten code if the test is inside of complicated, deeply nested `if` statements.

8.2 Examples With `for` Statements

Thus far all of our `for` loops have used a sequence of successive integers. Suppose you want to print the first n multiples of k, like the first 5 multiples of 3: 3, 6, 9, 12, 15. This could be handled by generating a sequence i = 1 through n, and multiply each i by k:

```
for (int i = 1; i <= n; i++) {
    Console.WriteLine(i*k);
}
```

Another approach is to note that the numbers you want to print advance in a regular fashion, too, but with an increment 3 in the example above, or k in general:

```
for (int i = k; i <= n*k; i = i+k) {
    Console.WriteLine(i);
}
```

The

```
i = i + k;
```

is a common pattern, less common than incrementing by one, but still very common. C# and many other languages allow a shorter version:

```
i += k;
```

This means to increment the variable i by k.

> **Warning:** Be careful: the += must be in that order, with no space between. Unfortunately the reverse order:
>
> ```
> i =+ k;
> ```
>
> is also legal, and just assigns the value of k to i.

Most C# binary operations have a similar variation. For instance if *op* is +, −, *, / or %,

> **variable** *op*= *expression*

means the same as

> **variable** = **variable** op *expression*

For example

```
x *= 5;
```

is the same as

```
x = x * 5;
```

8.2.1 Tables

Reports commonly include tables, often with successive lines generated by a consistent formula. As a simple first table, we can show the square, cube, and square root of numbers 1 through 10. The Math class has a function Sqrt, so we take the square root with Math.Sqrt function. The pattern is consistent, so we can loop easily:

```
for ( int n = 1; n <= 10; n++) {
    Console.WriteLine("{0} {1} {2} {3}", n, n*n, n*n*n, Math.Sqrt(n));
}
```

The numbers will be there, but the output is not pretty:

```
1 1 1 1
2 4 8 1.4142135623731
3 9 27 1.73205080756888
4 16 64 2
```

```
5 25 125 2.23606797749979
6 36 216 2.44948974278318
7 49 343 2.64575131106459
8 64 512 2.82842712474619
9 81 729 3
10 100 1000 3.16227766016838
```

First we might not need all those digits in the square root approximations. We can replace {3} by {3:F4} to just show 4 decimal places.

We can adjust the spacing to make nice columns by using a further formatting option. The longest entries are all in the last row, where they take up, 2, 3, 4, and 6 columns (for 3.1623). Change the format string:

```
for ( int n = 1; n <= 10; n++) {
    Console.WriteLine("{0,2} {1,3} {2,4} {3,6:F4}",
                          n, n*n, n*n*n, Math.Sqrt(n));
}
```

and we generate the neater output:

```
 1   1    1 1.0000
 2   4    8 1.4142
 3   9   27 1.7321
 4  16   64 2.0000
 5  25  125 2.2361
 6  36  216 2.4495
 7  49  343 2.6458
 8  64  512 2.8284
 9  81  729 3.0000
10 100 1000 3.1623
```

We are using two new formatting forms:

> {index, fieldWidth} and
> {index, fieldWidth:F#}

where index, fieldWidth, and # are replaces by specific literal integers. The new part with the comma (not colon) and fieldWidth, sets the *minimum* number of columns used for the substituted string, padding with blanks as needed.

> **Warning:** There is a *special* language for the characters between the braces in a format string. The rules are different than in regular C# code, where comma and colon are symbols, and the parser allows *optional whitespace* around them. This is *not* the case inside the braces of a format string: There cannot be a space after the colon or before the comma. Some blanks are legal; some blanks lead to exceptions being thrown, and other positions for blanks just silently give the wrong format.
>
> The safest approach for a programmer is just to have *no* blanks between the braces in a format string.

If the string to be inserted is wider than the fieldWidth, then the *whole* string is inserted, *ignoring* the fieldWidth. Example:

```
string s = "stuff";
Console.WriteLine("123456789");
Console.WriteLine("{0,9}\n{0,7}\n{0,5}\n{0,3}", s);
```

generates:

```
123456789
    stuff
  stuff
stuff
stuff
```

filling 9, 7, and then 5 columns, by padding with 4, 2, and 0 blanks. *The last line sticks out past the proposed 3-column fieldWidth.*

One more thing to add to our power table is a heading. We might want:

```
n   square    cube    root
```

To make the data line up with the heading titles, we can expand the columns, with code in example power_table/power_table.cs:

```
        Console.WriteLine("{0,2}{1,7}{2,5}{3,7}",
                        "n", "square", "cube", "root");
        for ( int n = 1; n <= 10; n++) {
            Console.WriteLine("{0,2}{1,7}{2,5}{3,7:F4}",
                            n, n*n, n*n*n, Math.Sqrt(n));
        }
```

8.2. Examples With `for` Statements   **241**

generating:

```
 n   square      cube      root
 1        1         1   1.0000
 2        4         8   1.4142
 3        9        27   1.7321
 4       16        64   2.0000
 5       25       125   2.2361
 6       36       216   2.4495
 7       49       343   2.6458
 8       64       512   2.8284
 9       81       729   3.0000
10      100      1000   3.1623
```

Note how we make sure the columns are consistent in the heading and further rows: We used a format string for the headings with the same field widths as in the body of the table. A separate variation: We also reduced the length of the format string by putting all the substitution expressions in braces right beside each other, and generate the space between columns with a larger field width.

Left Justification: Though our examples have always right justified in a field (padding on the left), for completeness we note this alternative: A minus sign in front of the fieldWidth places the result left justified (padded on the right). For example:

```
string s = "stuff";
Console.WriteLine("1234567890");
Console.WriteLine("{0,-9}|\n{0,-7}|\n{0,-5}|\n{0,-3}|", s);
```

prints:

```
1234567890
stuff    |
stuff  |
stuff|
stuff|
```

where the 'l' appears after any blank padding in each line.

8.2.2 ASCII Codes

Here is a reverse lookup from the *Numeric Code of String Characters*: Find the characters for a list of numeric codes. Just as we can cast a `char` to an `int`, we can cast an `int` 0-127 to a `char`.

The Unicode used by C# is an extension of the ASCII codes corresponding to the characters on a US keyboard. The codes were originally used to drive printers, and the first 32 codes are non-printable instructions to the printer. Characters 32 - 126 yield the 95 characters on a standard US keyboard.

A loop to print each code followed by a space and the corresponding printable character would be:

```
for (int i = 32; i < 127; i++) {
    Console.WriteLine("{0,3} {1}", i, (char)i);
}
```

To make all the character line up we added a field width 3 for the code column.

If you run this in csharp, the first line printed does not appear to have a character: That is the blank character. All the other characters are visible.

Let us make a more concise table, putting 8 entries per line. We can print successive parts using `Write` instead of `WriteLine`, but we still need to advance to the next line after every 8th entry, for codes 39, 47, 55, ... Since they are 8 apart, their remainder when divided by 8 is always the same:

$$7 = 39 \% 8 = 47 \% 8 = 55 \% 8 = ...$$

We can add a newline after each of these is printed. This requires a test:

```
for (int i = 32; i < 127; i++) {
    Console.Write("{0,3} {1}  ", i, (char)i);
    if (i % 8 == 7) {
        Console.WriteLine();
    }
}
```

Recall that `Console.WriteLine()` with no parameters *only* advances to the next line.

Paste that whole code at once into csharp to see the result.

The next csharp> prompt appears right after `126 ~`. There is no eighth entry on the last line, and hence no advance to the next line. A program printing this table should include an extra `Console.WriteLine()` after the loop.

8.2.3 Modular Multiplication Table

We have introduced the remainder operator `%` and mentioned that the corresponding mathematical term is "mod". We can extend that to the idea of modular arithmetic systems. For example, if we only look at remainders mod 7, we can just consider numbers 0, 1, 2, 3, 4, 5, and 6. We can do multiplication and addition and take remainders mod 7 to get answers in the same range. For example 3 * 5 mod 7 is `(3 * 5) % 7` in C#, which is 1. As we look more at this system, we will observe and explain more properties.

The next example is to make a table of multiplication, mod 7, and later generalize.

Tables generally have row and column labels. We can aim for something like:

```
* | 0 1 2 3 4 5 6
-----------------
0 | 0 0 0 0 0 0 0
1 | 0 1 2 3 4 5 6
2 | 0 2 4 6 1 3 5
3 | 0 3 6 2 5 1 4
4 | 0 4 1 5 2 6 3
5 | 0 5 3 1 6 4 2
6 | 0 6 5 4 3 2 1
```

The border labels make the table much more readable, but let us start simpler, with just the modular multiplications:

```
0 0 0 0 0 0 0
0 1 2 3 4 5 6
0 2 4 6 1 3 5
0 3 6 2 5 1 4
0 4 1 5 2 6 3
0 5 3 1 6 4 2
0 6 5 4 3 2 1
```

This is more complicated in some respects than our previous table, so start slow, with some pseudocode. We need a row for each number 0-6, and so a `for` loop

suggests itself:

```
for (int r = 0; r < 7; r++) {
    print row
}
```

Each individual row also involves a repeated pattern: calculate for the next number. We can name the second number c for column. The next revision replaces "print row" by a loop: a *nested* loop, inside the loop for separate rows:

```
for (int r = 0; r < 7; r++) {
    for (int c = 0; c < 7; c++) {
        print modular multiple on same line
    }
}
```

and the modular multiplication is just regular multiplication followed by taking the remainder mod 7, so you might come up with the C# code:

```
for (int r = 0; r < 7; r++) {
    for (int c = 0; c < 7; c++) {
        int modProd = (r*c) % 7;
        Console.Write(modProd + " ");
    }
}
```

You can test this in csharp, and see it is not quite right! Chopped-off output starts:

```
0 0 0 0 0 0 0 0 1 2 3 4 5 6 0 2 4 6 1 3 5 0 3 6 2 5 1 4 0...
```

Though we want each entry in a row on the same line, we need to go down to the next line at the end of each line! Where do we put in the newline in the code? A line is *all* the modular products by r, *followed* by *one* newline. Each modular product for a row is printed in the inner `for` loop. We want to advance *after* that, so the newline must be inserted *outside the inner loop*. On the other hand we do want it done for *each* row, so it must be *inside the outer loop*:

```
1  for (int r = 0; r < 7; r++) {
2      for (int c = 0; c < 7; c++) {
3          int modProd = (r*c) % 7;
4          Console.Write(modProd + " ");
5      }
```

8.2. Examples With `for` Statements   **245**

```
6    Console.WriteLine();
7  }
```

You can copy and test that code in csharp, and it works!

It is important to be able to play computer on nested loops and follow execution, statement by statement. Look more closely at the code above, noting the added line numbers. The basic pattern *is* sequential: *Complete* one statement before going on to the next. *Inside* the execution of a looping statement, there are extra rules, for testing and looping through the whole body. Within a loop body, each *complete* statement is executed sequentially.

Most new students can get successfully to line 4:

line	r	c	modProd	comment
1	0	-	-	initialize outer loop
2	0	0	-	initialize inner loop
3	0	0	0	
4	0	0	0	Write 0

After reaching the bottom of the loop, where do you go? You finish the *innermost* enclosing statement. You are in the inner loop, so the next line is the *inner* loop heading where you increment c and continue with the loop since $1 < 7$. This inner loop continues until you reach the bottom of the inner loop, line 4, with $c = 6$, and return to the heading, line 2, and the test fails, finishing the inner row loop:

line	r	c	modProd	comment
1	0	-	-	initialize outer loop
2	0	0	-	$0 < 7$, enter loop body
3	0	0	0	(0*0)%7
4	0	0	0	Write 0
2	0	1	-	c=0+1=1, $1 < 7$: true
3	0	1	0	(0*1)%7
4	0	1	0	Write 0
2	0	2	-	c=1+1=2, $2 < 7$: true
...				... through $c = 6$
4	0	6	0	Write 0
2	0	7	-	c=+1=7, $7 < 7$: false

At this point the inner loop statement, lines 2-4, has completed, and you continue. You go on to the next statement in the same sequential chuck as the inner loop

statement in lines 2-4: That sequential chunk is the the outer loop body, lines 2-6. The next statement is line 6, advancing printing to the next line. That is the last statement of the outer loop, so you return to the heading of the outer loop and modify its loop variable r. The two lines just described are:

line	r	c	modProd	comment
6	0	-	-	print a newline
1	1	-	-	r=s0+1=1, 1 < 7 enter outer loop

Then you go all the way through the inner loop again, for all columns, with c going from 0 through 6, and exit at c=7, finish the body of the outer loop by advancing to a new print line, and return to the outer loop heading, setting r = 2..., until all rows are completed.

The common error here is to forget what loop is the innermost one that you are working on, and exit that loop before is is totally finished: It finishes when the test of the condition controlling the loop becomes false.

Look back one more time and make sure the code for this *simpler* table makes sense before we continue to the one with labels....

The fancier table has a couple of extra rows at the top. These two rows are unlike the remaining rows in the body of the table, so they need special code.

If we go back to our pseudocode we could add to it:

```
print heading row
print dash-row
for (int r = 0; r < 7; r++) {
    print body row
}
```

First analyze the heading row: Some parts are repetitive and some are not: Print `" * | "` once, and then there is a repetitive pattern printing 0 - 6, which we can do with a simpler loop than in the table body:

```
Console.Write("* | ");
for ( int i = 0; i < 7; i++) {
    Console.Write(i + " ");
}
Console.WriteLine();
```

The dashed line can be generated using `StringOfReps` from *Lab: Loops*. How many dashes? A digit and a space for each of seven columns and for a row header,

---

so we need (7+1)*(1+1) characters, plus one for the 'I': 1 + (7+1)*(1+1). Thinking ahead, we will leave that expression unsimplified.

We have done most of the work for the rows of the body of the table in the simpler version. We just have a bit of printing for the initial row label. Where does the code go? It is repeated for each row, so it is inside the outer loop, but it is just printed once per row, so it comes *before* the inner column loop. The row label is r. The whole code is in example mod7_table/mod7_table.cs and below:

```
//heading
Console.Write("* | ");
for ( int i = 0; i < 7; i++) {   //column headings
   Console.Write(i + " ");
}
Console.WriteLine();

Console.WriteLine(StringOfReps("-", 1 + (7+1)*(1+1)));

for (int r = 0; r < 7; r++) {      // table body
   Console.Write(r + " | ");       // row heading
   for (int c = 0; c < 7; c++) {   // data columns
      int modProd = (r*c) % 7;
      Console.Write(modProd + " ");
   }
   Console.WriteLine();
}
```

Besides the 0 row and 0 column in the mod 7 table, note that in each line the products are a permutation of all the numbers 1-6. That means it is possible to define the *inverse* of the multiplication operation, and mod 7 arithmetic actually forms a mathematical *field*. Modular arithmetic (with much larger moduli!) is extremely important in *public key cryptography*, which protects all your online financial transactions.... Knowing a lot more math is useful! (But it is not required for this course.)

The inverse operation to multiplication for prime moduli is easy to work out by brute force, going through the row of products. A much more efficient method is needed for cryptography: That method involves an elaboration of *Greatest Common Divisor*.

Finally, let us generalize this table to mod n. With n up to about 25, it is reasonable to print. Most of the changes are just replacing 7 by n. There is a further

complication with column width, since the numbers can be more than one digit long. We can do formatting with a field width. Unfortunately in C# the field width must be a *literal* integer embedded in the format string, but our number of digits in n is *variable*.

Here is a good trick: Construct the format string inside the program. To get the format for a number and an extra space mod 7, we want format string "{0,1} ", but for mod 11, we want "{0,2} ". This 1 or 2 is the number of characters in n as a string, given by

```
numberWidth = ("" + n).Length;
```

We can create the format string with a string concatenation expression:

```
string colFormat = "{0," + numberWidth + "}";
```

or use *another* format string and substitution. This is an excuse to illustrate including explicit braces (for the main format string). Recall the explicit braces are doubled. Check out this version:

```
string colFormat = string.Format("{{0,{0}}} ", numberWidth);
```

which we use in the code for the whole function, below, and in example program mod_mult_table/mod_mult_table.cs.

```
    /// Print a table for modular multiplication mod n.
    static void MultTable(int n)
    {
        int numberWidth = ("" + n).Length;
        string colFormat = string.Format("{{0,{0}}} ", numberWidth);
        string rowHeaderFormat = colFormat + "| ";
        Console.Write(rowHeaderFormat,"*"); // start main heading
        for ( int i = 0; i < n; i++) {
            Console.Write(colFormat, i);
        }
        Console.WriteLine();

        Console.WriteLine(StringOfReps("-",(numberWidth+1)*(n+1) + 1));

        for (int r = 0; r < n; r++) { //rows of table body
            Console.Write(rowHeaderFormat, r);
            for (int c = 0; c < n; c++) {
                Console.Write(colFormat, (r*c) % n);
```

```
        }
        Console.WriteLine();
    }
}
```

8.2.4 Reversed String Returned

In *String Backwards Exercise/Example* we discuss iterating through a string's indices and characters to print the string reversed. That might be useful, but it logically the joining of two separate ideas: reversing a string and printing it. We already know how to print a string as a step. Now consider the first part as its own function:

```
    /// Return s in reverse order.
    /// If s is "drab", return "bard".
    static string Reverse (string s)
```

To go along with this chapter, we will use a `for` loop heading rather a `while` loop as in reversed_print/reversed_print.cs:

```
for (int i = s.Length - 1; i >= 0; i--) {
```

A more significant difference is that in the previous example we immediately printed, individually, each letter that we wanted. Now we need to create a single string, with all the characters, before returning the result.

Let us think of the example in the documentation: If we start with `s` as `"drab"`, and we go through the letters one at a time in reverse order, b a r d, we build up successively:

```
b
ba
bar
bard
```

We need a loop with variables and operations. The sequence of reversed letters, `s[i]`, are the last character on the end of each line above.

At least lines after the first are constructed from previous parts, so, for instance, `"bar"` comes from combining the initial part `"ba"` with the latest character `'r'` (`s[i]`). We need a name for the initial part. I used the name `rev`. Combining

with a string is done with the + operator. Then when `rev` is `"ba"` and `s[i]` is `'r'`, the combination, using the variable names, is

```
rev + s[i]
```

We want this in our loop, so we must be able to use that expression *each* time through the loop, so `rev` changes each time through the loop. In the next iteration `rev` is the *result* of the previous expression. The assignment statement to give us the next version of `rev` can just be:

```
rev = rev + s[i];
```

That gives us the general rule. Pay attention now to the beginning and end: The end is simple: The last value for `rev` is the complete reversed string, so that is what we return.

How do we initialize `rev`? You could imagine `rev` starting as `"b"`, but the the first character that we add is `'a'`, and we would not be going through all the characters in our loop. It is better to go all the way back to the beginning: If we use the general form with the first letter in the reversed sequence,

```
rev = rev + s[i];
```

then the result of the initial `rev + 'b'` should just be `"b"`. So what would `rev` be?

Remember the empty string: initialize `rev` to be `""`.

The result is:

```csharp
/// Return s in reverse order.
/// If s is "drab", return "bard".
static string Reverse (string s)
{
    string rev = "";
    for (int i = s.Length - 1; i >= 0; i--) {
        rev += s[i];
    }
    return rev;
}
```

We used our new operator += to be more concise.

This function and a `Main` used to demonstrate it are in re-versed_string/reversed_string.cs.

8.2.5 Exercises

Head or Tails Exercise

Write a program `heads_tails.cs`. It should include a function `Flip()`, that will just randomly print `Heads` or `Tails` *once*. Accomplish this by choosing 0 or 1 arbitrarily with a random number generator. More details follow.

Use a `Random` object, as in *Number Guessing Game Lab*, *except* this time it is important *not* to make the `Random` object be a local variable inside the `Flip` function: A new `Random` object in likely initialized using the current time. The `Flip` function has no interaction with the user, so it can be repeated very quickly, and new `Random` objects may not register a new value through several reruns of `Flip`. This would give the same answer, and be completely contrary to the idea of random results!

Hence it is generally a good idea to only create a single `Random` object that stays in scope for the whole program. One way to do that is to make it *static*. Place the declaration

```
static Random r = new Random();
```

inside your class but outside of any function, positioned like the static constants discussed in *Static Variables*.

Then you can use `r` in any function in your class. For `int` variables `low` and `higher`, with `low < higher`:

```
int n = r.Next(low, higher);
```

returns a (pseudo) random `int`, satisfying `low <= n < higher`. If you select `low` and `higher` as 0 and 2, so there are only two possible values for n, then you can choose to print `Heads` or `Tails` with an `if-else` statement based on the result.

> **Warning:** We have discovered some problems with the `Next()` implemen-
> tation when running on Mono that sometimes results in random values not
> being generated. This is likely a bug that will be fixed. If you experience any
> problems with `Next()`, the following is for you!

An alternative to generating random 0 and 1 values for heads and tails is to gen-
erate random double-precision values. Using the same variable, `r`, you can call
`r.NextDouble()` to get a random value between 0 and 1. You can consider
any generated value $n < 0.5$ to be heads; $n >= 0.5$ represents tails:

```csharp
double n = r.NextDouble();
if (n < 0.5) {
   // heads
} else {
   // tails
}
```

In your `Main` method have a `for` loop calling `Flip()` 10 times to test it, so you
generate a random sequence of 10 heads and/or tails. With these 10 rapid calls, it
is important that a new Random object is only created once. The suggested static
variable declaration ensures that.

Group Flips Exercise

Write a program `format_flips.cs`. It should include the function `Flip()`
and the `static Random` declaration from the last exercise. Also include another
function:

```csharp
/// Print out the results from the total number of random flips of a coin.
/// Group them groupSize per line, each followed by a space.
/// The last line may contain fewer than groupSize flips
/// if total is not a multiple of groupSize.  The last line
/// should be followed by exactly one newline in all cases.
/// For example, GroupFlips(10, 4) *could* produce:
///    Heads Heads Tails Heads
///    Heads Tails Heads Tails
///    Tails Tails
static void GroupFlips(int total, int groupSize)
```

8.2. Examples With `for` Statements

Complete this function definition and test with a variety of calls to `GroupFlips` in `Main`. The output from the previous exercise would be produced by the call:

```
GroupFlips(10, 1);
```

Reverse String `foreach` Exercise

We already have discussed *Reversed String Returned*. It used a `for` loop to go through the characters in reverse order. Write a version with the only loop heading:

```
foreach(char ch in s) {
```

and no reference to indices in s.

Only Letters Exercise

Write a program that defines and tests a function with description and heading:

```
/// Return s with all non-letters removed.
/// For example OnlyLetters("Hello, World!") returns "HelloWorld".
static string OnlyLetters(string s)
```

Assume the English alphabet.

Palindrome Exercise

Write a program `palindrome.cs` that defines and tests a function with description and heading:

```
/// Return true when s is a palindrome.
/// For example IsPalindrome("A Toyota!") returns true.
static bool IsPalindrome(string s)
```

A palindrome is a string that contains the same sequence of letters, ignoring capitalization, forward and backward. Non-letters are ignored. Examples are "Madam, I'm Adam." and "Able was I 'ere I saw Elba."

`IsPalindrome` can be written very concisely by copying and using functions from previous exercises.

Nested Play Computer Exercise

Predict what these code fragments print. Then check yourself in csharp:

```csharp
for (int i = 3; i > 0; i--) {
    for (int j = i; j < 4; j++) {
        Console.Write(j);
    }
    Console.WriteLine();
}

string s = "abcdef";
for (int i = 1; i < s.Length; i += 2) {
    for (int k = 0; k < i; k++) {
        Console.Write(s[i]);
    }
}
```

Power Table Exercise

1. Write a program `power_table.cs` that completes and tests the function with this heading. Be sure your program tests with several values for each parameter:

```csharp
/// Print a table of powers of positive integers.
/// Assume 1 <= nMax <= 12, 1 <= powerMax <= 7.
/// Example: output of PowerTable(3, 4)
///       n^1       n^2       n^3       n^4
///         1         1         1         1
///         2         4         8        16
///         3         9        27        81
///
public static void PowerTable(int nMax, int powerMax)
```

Make sure the table always ends up with right-justified columns.

2. Make the table have columns all the same width, but make the width be as small as possible for the parameters provided, leaving a minimal one space (but not less!) between columns somewhere in the table. Consider heading widths, too.

8.3 Lab: Loops

Goals for this lab:

- Practice with loops. You are encouraged to use a `for` loop where appropriate.

- Use nested loops where appropriate.

Copy example loop_lab_stub/loop_lab.cs to a new project of yours, and fill in function bodies for each part below:

1. Complete

```
/// Print n copies of s, end to end.
/// For example PrintReps("Ok", 9) prints: OkOkOkOkOkOkOkOkOk
static void PrintReps(string s, int n)
```

Hint: How would you do something like the example `PrintReps("Ok", 9)` or with a higher count by hand? Probably count under your breath as you write:

```
 1 2 3 4 5 6 7 8 9
OkOkOkOkOkOkOkOkOk
```

This is a counting loop.

2. Complete

```
/// Return a string containing n copies of s, end to end.
/// For example StringOfReps("Ok", 9) returns: "OkOkOkOkOkOkOkOk
static string StringOfReps(string s, int n)
```

Note the distinction from the previous part: Here the function prints nothing. Its work is *returned* as a single string. You have to build up the final string.

3. Complete `Factorial`, in a format much like SumToN in example sum_to_n_test/sum_to_n_test.cs:

```
/// Return n! (n factorial: 1*2*3 *...* n if n >=1;
/// 0! is 1.).  For example Factorial(4) returns 1*2*3*4 = 24.
static int Factorial(int n)
```

It is useful to think of the sequence of steps to calculate a concrete example of a factorial, say 6!:

```
Start with 1
2 *1 = 2
3*2 = 6
4 * 6 = 24
5*24 = 120
6*120 = 720
```

ALSO find the largest value of `n` for which the function works. (You might want to add a bit of code further testing Factorial, to make this easier.) Caution: although a negative result from the product of two positive numbers is clearly wrong, only half of the allowed values are negative, so the first wrong answer could equally well be positive.

4. Modify the function to return a `long`. Then what is the largest value of `n` for which the function works?

   Remember the values from this part and the previous part to tell the TA's checking out your work.

5. Complete the method

```
// Print a filled rectange, where the filling is
// the specified number of columns and rows of the character inChar,
// surrounded by a border made of the character edgeChar.
// For example printRectangle(3, 2, 'i', 'e') prints
//     eeeee
//     eiiie
//     eiiie
//     eeeee
static void PrintRectangle(int columns, int rows,
                           char inChar, char edgeChar)
```

Here are further examples:

```
PrintRectangle(5, 1, ' ', 'B');
PrintRectangle(0, 2, '-', '+');
```

would print

```
BBBBBBB
B     B
```

```
BBBBBBB
++
++
++
++
```

Suggestion: You are always encouraged to build up to a complicated so-
lution incrementally. You might start by just creating the inner rectangle,
without the border.

6. **40% Extra Credit** Complete the method below.

```
/// Print the borders of the cells of a table.
/// The borders divide the table into rows and columns.
/// The blank space within a cell is width characters wide
/// and continues down for height lines.
/// The horizontal borders are dashes '-' and the vertical borders
/// are vertical bars, '|', except that all intersections are '+'.
/// For example PrintTableBorders(3, 2, 4, 1) prints
///     +----+----+----+
///     |    |    |    |
///     +----+----+----+
///     |    |    |    |
///     +----+----+----+
static void PrintTableBorders(int columns, int rows,
                             int width, int height)
```

Here is further example:

```
PrintTableBorders(2, 1, 6, 3);
```

would print (with actual vertical bars)

```
+------+------+
|      |      |
|      |      |
|      |      |
+------+------+
```

You can do this with lots of nested loops, or much more simply you can use
`StringOfReps`, possibly six times in several assignment statements, and
print a single string. Think of larger and larger building blocks.

The source of this book is plain text where some of the tables are laid out in

a format similar to the output of this function. The Emacs editor has a mode that maintains a fancier related setup on the screen, on the fly, as content is added inside the cells!

8.4 Chapter Review Questions

1. When might you prefer a `for` loop in place of a `while` loop? What do you gain?

2. When might you prefer a `while` loop or a `foreach` instead of a `for` loop?

3. When you have nested `for` loops, and you reach the bottom of the *body* of the *inner* loop, where does execution go next?

4. May you legally omit the initialization part of a `for` loop?

5. What happens when you omit the condition in a `for` loop?

6. In the heading of a `for` loop, how do you initialize or update several variables?

7. Rewrite

```
num /= 2;
```

equivalently without the operand `/=`.

8. Rewrite

```
bigName = bigName - 10;
```

with a statement that only includes `bigName` once.

9. Distinguish the effects of these two statements:

```
x-=2;
```

```
x=-2;
```

10. What is printed?

```
    Console.WriteLine("12345678");
    for( int p = 1; p < 6; p++) {
        string formatStr = "{0:F" + p + "}";
        Console.WriteLine(formatStr, 1.2345678);
    }
```

11. What is printed? (Just ",4" has been inserted.)

```
    Console.WriteLine("12345678");
    for( int p = 1; p < 6; p++) {
        string formatStr = "{0,4:F" + p + "}";
        Console.WriteLine(formatStr, 1.2345678);
    }
```

12. What is printed?

```
    Console.WriteLine("123456");
    for( int w = 6; w >= -6; w -= 4) {
        string formatStr = "{0," + w + "}|";
        Console.WriteLine(formatStr, "here");
    }
```

FILES, PATHS, AND DIRECTORIES

9.1 Files As Streams

Thus far you have been able to save programs, but anything produced during the execution of a program has been lost when the program ends. Data has not *persisted* past the end of execution. Just as programs live on in files, you can generate and read data files in C# that persist after your program has finished running.

As far as C# is concerned, a file is just a string (often very large!) stored on your file system, that you can read or write, gradually, line by line, or all together.

C# has the abstraction of a *stream*, as a sequence of characters to be processed sequentially. A stream can either be written sequentially or read sequentially. You have already read and written streams of characters to the Console. Most of the syntax that we use for files will be very similar, using methods `ReadLine`, `WriteLine`, and `Write` in the same way you used them for the `Console`.

Files can be handled very differently by different operating systems, but C# abstracts away the differences and provides stream interfaces between a C# program and files.

9.2 Writing Files

Try the following:

1. In Xamarin Studio build, *not* run, the project first_file. Build is the first selection in the local popup menu for first_file in the Solution pad. Recall to get the local popup menu

    - go to the Solution pad

    - right click on the project (Mac control-click)

2. Next open an operating system directory window for the project. With Xamarin Studio open, a quick way to do that is to go to the same popup window, and this time select "Open Containing Folder".

3. Besides the project files from the Solutions pad, in the directory window you should also see a folder `bin`. Change to that folder and then to its sub-folder `Debug`. This is where the build step put its result `first_file.exe` and debug information `first_file.exe.mdb`. You should see no other file. Leave this window open.

4. Now, back in Xamarin Studio, run the project. Depending on your operating system, you may or may not see a Console Window pop up. If you do see one, you should not see any evidence of program results. If you got a window, close it.

5. Look at the directory window again. You should see a file `sample.txt`. This is a file created by the program you ran.

Here is the program:

```csharp
using System;
using System.IO;

namespace IntroCS
{
   class FirstFile  // basics of file writing
   {
      public static void Main()
      {
         StreamWriter writer = new StreamWriter("sample.txt");
         writer.WriteLine("This program is writing");
         writer.WriteLine("our first file.");
         writer.Close();
      }
   }
}
```

Look at the code. Note the extra namespace being used at the top. You will always need to be using `System.IO` when working with files. Here is a slightly different use of a dot, `.`, to indicate a subsidiary namespace.

The first line of `Main` creates a `StreamWriter` object assigned to the variable `writer`. A `StreamWriter` links C# to your computer's file system for writing, not reading. Files are objects, like a Random, and use the `new` syntax to create a new one. The parameter in the constructor gives the name of the file to connect to the program, `sample.txt`.

> **Warning:** If the file already existed, the old contents are *destroyed* silently by creating a `StreamWriter`.

If you do not use any operating system directory separators in the name (`'\'` or `'/'`, depending on your operating system), then the file will lie in the *current directory*, discussed more shortly. The Xamarin Studio default is for this current directory to be this Debug directory. This will be inconvenient in many circumstances, and later in the chapter we will see how to minimize the issue.

The second and third lines of `Main` write the specified strings to lines in the file. Note that the `StreamWriter` object `writer`, not `Console`, comes before the dot and `WriteLine`. This is yet another variation on the use of a dot, `.`: between an object and a function tied to this object. In this situation the function tied to an object is more specifically called a *method*, in object-oriented terminology. All the uses of a dot (except for a numerical literal value) share a common idea, indicating a named part or attribute of a larger thing.

The last line of `Main` is important for cleaning up. Until this line, this C# program controls the file, and nothing may be actually written to the operating system file yet: Since initiating a file operation is thousands of times slower than memory operations, C# *buffers* data, saving small amounts and writing a larger chunk all at once.

> **Warning:** The call to the `Close` method is essential for C# to make sure everything is really written, and to relinquish control of the file for use by other programs.

It is a common bug to write a program where you have the code to add all the data you want to a file, but the program does not end up creating a file. Usually this means you forgot to close the file!

As discussed above, Xamarin Studio places `sample.txt` in the `Debug` sub-subfolder, a hard-to-guess place in the file system, that is *not* shown in the Solution pad, so do not look for it there! As you should have checked above, you *can* see it in an operating system file window. Do drill down to the Debug folder if you have not already; open the `sample.txt` file with your favorite text processor. It should contain just what was written!

If you were to run the program from the command line instead of from Xamarin Studio, the file would appear in the current directory.

Just as you can use a *String Format Operation* with functions `Write` and `WriteLine` of the `Console` class, you can also use a format string with the corresponding methods of a `StreamWriter`, and embed fields by using braces in the format string.

9.3 Reading Files

In Xamarin Studio, go to project print_first_file. Right click on the project in the Solution pad, and select Open Containing Folder. Drill down two folders through `bin` to `Debug`. You should find a copy of the `sample.txt` that we stored there. You can open it and look at it if you like.

Run the example program print_first_file/print_first_file.cs, shown below:

```csharp
using System;
using System.IO;

namespace IntroCS
{
   class PrintFirstFile  // basics of reading file lines
   {
      public static void Main()
      {
         StreamReader reader = new StreamReader("sample.txt");
         string line = reader.ReadLine();  // first line
         Console.WriteLine(line);
         line = reader.ReadLine();         // second line
         Console.WriteLine(line);
         reader.Close();
      }
```

```
        }
}
```

Now you have read a file and used it in a program.

In the first line of `Main` the operating system file (`sample.txt`) is associated again with a C# variable name (`reader`), this time for reading as a `StreamReader` object. A `StreamReader` can only open an existing file, so `sample.txt` must already exist.

Again we have parallel names to those used with `Console`, but in this case the `ReadLine` method returns the next line from the file. Here the string from the file line is assigned to the variable `line`. Each call the ReadLine reads the next line of the file.

Using the `Close` method is generally optional with files being read. There is nothing to lose if a program ends without closing a file that was being read. [1]

9.3.1 Reading to End of Stream

In `first_file.cs`, we explicitly coded reading two lines. You will often want to process each line in a file, without knowing the total number of lines while programming. This means that files provide us with our second kind of a sequence: this time the sequence of lines in the file! To process all of them will require a loop and a new test to make sure that you have not yet come to the end of the file's stream: You can use the `EndOfStream` property. It has the wrong sense (true at the end of the file), so we negate it, testing for `!reader.EndOfStream` in the example program `print_file_lines.cs`. This little program reads and prints the contents of a file specified by the user, one line at a time:

```csharp
using System;
using System.IO;

namespace IntroCS
{
    class PrintFileLines  // demo of using EndOfStream test
    {
        public static void Main()
```

_____

[1] If, for some reason, you want to reread this same file while the same program is running, you need to close it and reopen it.

```
    {
        string userFileName = UI.PromptLine("Enter name of file to print: ")
        var reader = new StreamReader(userFileName);
        while (!reader.EndOfStream) {
            string line = reader.ReadLine();
            Console.WriteLine(line);
        }
        reader.Close();
    }
  }
}
```

var For conciseness (and variety) we declared `reader` using the more compact syntax with `var`:

```
    var reader = new StreamReader(userFileName);
```

You can use `var` in place of a declared type to shorten your code with a couple of restrictions:

- Use an initializer, from which the type of the variable can be inferred.

- Declare a local variable inside a method body or in a loop heading.

- Declare only a single variable in the statement.

We could have used this syntax long ago, but as the type names become longer, it is more useful!

Things to note about reading from files:

- Reading from a file returns the part read, of course. Never forget the *side effect*: The location in the file advances past the part just read. The next read does *not* return the *same* thing as last time. It returns the *next* part of the file.

- Our `while` test conditions so far have been in a sense "backward look-ing": We have tested a variable that has *already been set*. The test with `EndOfStream` is *forward looking*: looking at what has not been processed yet. Other than making sure the file is opened, there is no variable that needs to be set before a `while` loop testing for `EndOfStream`.

- If you use ReadLine at the end of the file, the special value `null` (no object) is returned. *This* is not an error, but if you try to apply any string methods

to the `null` value returned, *then* you get an error!

Though `print_file_lines.cs` was a nice simple illustration of a loop read-
ing lines, it was very verbose considering the final effect of the program, just to
print the whole file. You can read the entire remaining contents of a file as a single
(multiline) string, using the `StreamReader` method `ReadToEnd`. In place of
the reading and printing loop we could have just had:

```
string wholeFile = reader.ReadToEnd();
Console.Write(wholeFile);
```

`ReadToEnd` does not strip off a newline, like `ReadLine` does, so we do not
want to add an extra newline when writing. We use the `Write` method instead of
`WriteLine`.

9.3.2 Example: Sum Numbers in File

We have summed the numbers from 1 to `n`. In that case we generated the next
number i automatically using i++. We could also read numbers from a file con-
taining one number per line (plus possible white space):

```
static int CalcSum(string filename)
{
   int sum = 0;
   var reader = new StreamReader(filename);
   while (!reader.EndOfStream) {
      string sVal = reader.ReadLine().Trim();
      sum += int.Parse(sVal);
   }
   reader.Close();
   return sum;
}
```

Below and in sum_file/sum_file.cs is a more elaborate, complete example, that
also exits gracefully if you give a bad file name. If you give a good file name, it
skips lines that contain only whitespace.

```
using System;
using System.IO;

namespace IntroCS
```

```
{
   class SumFile  // sum a file integers, one per line
   {
      static void Main()
      {
         string filename = UI.PromptLine(
                           "Enter the name of a file of integers: ");
         if (File.Exists(filename)) {
            Console.WriteLine("The sum is {0}", CalcSum(filename));
         }
         else {
            Console.WriteLine("Bad file name {0}", filename);
         }
      }

      /// Read the named file and
      /// return the sum of an int
      /// from each line that is not just white space.
      static int CalcSum(string filename)
      {
         int sum = 0;
         var reader = new StreamReader(filename);
         while (!reader.EndOfStream) {
            string sVal = reader.ReadLine().Trim();
            if (sVal.Length > 0) {
               sum += int.Parse(sVal);
            }
         }
         reader.Close();
         return sum;
      }
   }
}
```

A useful function used in `Main` for avoiding filename typo errors is `File.Exists` in the `System.IO` namespace

```
bool File.Exists(string filenamePath)
```

It is true if the named files exists in the operating system's file structure.

You should see the file sum_file/numbers.txt in the Xamarin Studio project. It is

in the right form for the program. If you run the program and enter the response:

```
numbers.txt
```

you should be told that the file does not exist. Recall that the executable created by Xamarin Studio is two directories down through `bin` to `Debug`. This is the default *current directory* when Xamarin Studio runs the program. You can refer to a file that is not in the current directory. A full description is in the next section, but briefly, what we need now: The symbol for the parent directory is `..` The hierarchy of folders and files are separated by \ in Windows and / on a Mac, so you can test the program successfully if you use the file name: `..\..\numbers.txt` in Windows and `../../numbers.txt` on a Mac. On a Mac, running the program looks like:

```
Enter the name of a file of integers: ../../numbers.txt
The sum is 16
```

In *FIO Helper Class* we will discuss a more flexible way of finding files to open, that works well in Xamarin Studio and many other situations.

Safe Sum File Exercise

1. Copy `sum_file.cs` to a file `safe_sum_file.cs` in a new project of yours. Modify the program: Write a new function with the heading below. Use it in `Main`, in place of the `if` statement that checks (only once) for a legal file:

```csharp
// Prompt the user to enter a file name to open for reading.
// Repeat until the name of an existing file is given.
// Open and return the file.
public static StreamReader PromptFile(string prompt)
```

2. A user who forgot the file name woud be stuck! Elaborate the function and program, so that an empty line entered means "give up", and `null` (no object) should be returned. The main program needs to test for this and quit gracefully in that case.

9.3.3 Example Copy to Upper Case

Here is a simple fragment from example file copy_upper/copy_upper.cs. It copies
a file line by line to a new file in upper case:

```csharp
var reader = new StreamReader("text.txt");
var writer = new StreamWriter("upper_text.txt");
while (!reader.EndOfStream) {
    string line = reader.ReadLine();
    writer.WriteLine(line.ToUpper());
}
reader.Close();
writer.Close();
```

You may test this in the Xamarin Studio example project copy_upper:

1. Expand the copy_upper project in the Solution pad. The project includes
   the input file. You may not see it at first. You need to expand the folder for
   `bin` and then `Debug`. You see `text.txt`.

2. To see what else is in the file system directory, you can select the Debug
   folder and right click and select Open Containing Folder. You should see
   `text.txt` but not `upper_text.txt`. Leave that operating system file
   folder open.

3. Go back to Xamarin Studio and run the project. Now look at the operating
   system `Debug` folder again. You should see `upper_text.txt`. You
   can open it and see that it holds an upper case version of the contents of
   `text.txt`.

This is another case where the `ReadToEnd` function could have eliminated the
loop. [2]

```csharp
string contents = reader.ReadToEnd();
writer.Write(contents.ToUpper());
```

---

[2] Besides the speed and efficiency of this second approach, there is also a technical improvement:
There may or may not be a newline at the end of the very last line of the file. The `ReadLine` method
works either way, but does not let you know the difference. In the line-by-line version, there is always
a newline after the final line written with `WriteLine`. The `ReadToEnd` version will have newlines
exactly matching the input.

9.4 Path Strings

When a program is running, there is alway a *current working directory*. When you run a project through Xamarin Studio, by default the current directory is the directory two levels below the project directory.

Files in the current working directory can to referred to by their simple names, e.g., *sample.txt*.

Referring to files not in the current directory is more complicated. You should be aware from using the Windows Explorer or the Finder that files and directories are located in a hierarchy of directories in the file system. On a Mac, the file system is unified in one hierarchy. On Windows, each drive has its own hierarchy.

Files are generally referred to by a chain of directories before the final name of the file desired. A *path string* is used to represent such a sequence of names. Elements of the directory chain are separated by operating system specific punctuation: In Windows the separator is backslash, \, and on a Mac it is (forward) slash, /. For example on a Mac the path

```
/Users/anh
```

starts with a /, meaning the *root* or top directory in the hierarchy, and Users is a subdirectory, and anh is a subdirectory of Users (in this case the home directory for the user with login anh). It is similar with Windows, except there may be a drive in the beginning, and the separator is a \, so

```
C:\Windows\System32
```

is on C: drive; Windows is a subdirectory of the root directory \, and System32 is a subdirectory of Windows. Each drive in Windows has a separate file hierarchy underneath it.

Paths starting from the root of a file system, with \ or / are called *absolute paths*. Since there is always a current directory, it makes sense to allow a path to be *relative* to the current directory. In that case do *not* start with the slash that would indicate the root directory. For example, if the current directory is your home directory, you likely have a subdirectory `Downloads`, and the `Downloads` directory might contain `examples.zip`. From the home directory, this file could be referred to as `Downloads\examples.zip` or `Downloads/examples.zip` on a Mac.

Relative to a Xamarin Studio project directory, the current directory for execution of the program is `bin\Debug` or `bin/Debug` on a Mac.

Referring to files in the current directory just by their plain file name is actually an example of using relative paths.

With relative paths, you sometimes want to move up the directory hierarchy: `..` (two periods) refers to the directory one level up the chain.

Next imagine reversing the relative path from a Xamarin Studio project directory to the current directory for execution: If the current directory is the execution directory, then `..` refers to directory `bin`, and then `..\..` or `../..` refers to the project directory. Further, if the project directory contains the file `numbers.txt`, then it could be referred to relative to the execution directory as `..\..\numbers.txt` or `../../numbers.txt`.

Occasionally you need to refer explicitly to the current directory: It is referred to as "." (a single period).

9.4.1 Paths in C#

The differing versions of paths for Windows and a Mac are a pain to deal with. Luckily C# abstracts away the differences. It has a `Path` class in the `System.IO` namespace that provides many handy functions for dealing with paths in an operating system independent way:

For one thing, C# knows the path separator character for your operating system, `Path.DirectorySeparatorChar`.

More useful is the function `Path.Combine`, which takes any number of string parameters for sequential parts of a path, and creates a single string appropriate for the current operating system. For example, `Path.Combine("bin", "Debug")` will return `"bin\Debug"` or `"bin/debug"` as appropriate. `Path.Combine("..", "..", "numbers.txt")` will return a string with characters `..\..\numbers.txt` or `../../numbers.txt`.

Even if you know you are going to be on Windows, file paths are a problem because `\` is the string escape character. To enter the Windows path above explicitly you would need to have `"..\\..\\numbers.txt"`, or the raw string prefix, `@` can come to the rescue: `@"..\..\numbers.txt"`.

You can look at the `Path` class in the MSDN documentation for many other operations with path strings.

Path strings are used by the *Directory Class* and by the *File Class*.

9.5 Directory Class

The `Directory` class is in the `System.IO` namespace. Directories in the file system are referenced by *Path Strings*. You can look at the MSDN documentation for a wide variety of functions in the `Directory` class including ones to list all the files in a directory or to check if a path string represents an actual directory.

9.6 File Class

We will generally access operating system files using the `Stream` abstraction, discussed earlier in this chapter. There is also a `File` class in the `System.IO` namespace, with a number of specialized and convenience functions. We already used the `File.Exists` function.

You can look at the MSDN documentation for other uses of the `File` class.

9.7 Command Line Execution

C# shields you from the differences between operating systems with its `File`, `Path`, and `Directory` classes.

If you leave Xamarin Studio and go to the command line as described in *Command Line Introduction*, then you are exposed to the differences between the operating systems. *Look over that section.*

Thus far we have let Xamarin Studio hide what actually is happening when you execute a program. The natural environment for the text-based programs that we are writing is the command line. We need to get outside of Xamarin Studio.

1. To show off the transition, first build or run the `addition1` example project from inside Xamarin Studio.

2. Open a terminal on a Mac or a Mono Command Prompt console in Windows.

3. Following the *Command Line Introduction*, change the current directory to the addition1 example project directory.

4. Then change to the subdirectory `bin` and then the subdirectory `Debug`.

5. Enter the command to list the directory (`dir` in Windows; `ls` on a Mac).

6. You should see `addition1.exe`. This is the compiled program created by Xamarin Studio. Enter the command

```
mono addition1.exe
```

This should run your program. Note that when you complete it, the window does not disappear! You keep that history. Keep this terminal/console window open until the end of the chapter.

7. Windows only: On Windows, Xamarin Studio creates a regular Windows executable file. For consistency you can use the command above, but you no longer need Mono. You can just enter the command `addition1.exe` or the shorter `addition1`.

9.7.1 MCS: Compiling

Continue with the same terminal/console window. Let us now consider creating an executable program for `addition1.cs`, directly, without using Xamarin Studio:

1. Enter the command `cd ..` and then *repeat*: `cd ..`, to go up to the project directory.

2. Print a listing of the directory. You should see `addition1.cs` but not `addition1.exe`.

3. Try the command

```
mono addition1.exe
```

You should get an error message, because `addition1.exe` is not in the current directory.

4. Enter the command

```
mcs addition1.cs
```

This is the Mono system compiler, building from the source code.

5. Print a listing of the directory. You should see now `addition1.exe`, created by the compiler.

6. Try the command again:

```
mono addition1.exe
```

Now try a program that had multiple files. The project version addition3 uses the library class UIF. Continue with the same terminal/console window:

1. Enter the commands:

```
cd ../addition3
mcs addition3.cs
```

You should get an error about missing the UIF class. The mcs program does not know about the information Xamarin Studio keeps in its references.

2. Extend the command:

```
mcs addition3.cs ../ui/uif.cs
```

That should work, now referring to both needed files.

3. Enter the command

```
mono addition3.exe
```

4. Now let us try a project where we read a file. Enter commands

```
cd ../sum_file
mcs sum_file.cs
mono sum_file.exe
```

We ran this program earlier through Xamarin Studio. Recall that that entering the file name `numbers.txt` failed, and to refer to the right place for the `numbers.txt` file, we needed to use `..\..\numbers.txt` or `../../numbers.txt`. This time *just enter* `numbers.txt`. The program should work, giving the answer 16.

By default mcs and mono read from and write to the current directory of the terminal/console. In the situation above, `sum_file.cs` and `numbers.txt` were in the project directory, which is the current directory. Then sum_file.exe was written to and run from the same directory.

This is unlike the Xamarin Studio default, where the current directory for execution is not the project directory.

Under the hood, Xamarin Studio uses mcs also, with a bunch of further options in the parameters, changing the execution directory and also arranging for better debugging information when you get a runtime error.

Xamarin Studio keeps track of all of the parts of your projects, and recompiles only as needed. There are also command-line tools that manage multi-file projects neatly, remembering the parts, and compiling only as necessary. One example is NAnt, which comes with Mono.

9.8 FIO Helper Class

We have already discussed and used the `UI` class to aid keyboard input. Now we are going to develop an `FIO` class for our libraries. The `FIO` class aids file input and output with Xamarin Studio, and illustrates a number of more generally useful ideas.

You saw in the last section how we might refer to `numbers.txt` in different ways depending on the execution environment. Our situation is based on the particular choices made by the creators of Xamarin Studio. More generally, there are many times when a program may need a file that may be stored in one of several directories.

Our `FIO` class will address this issue, and we will set up the parameters to work specifically with both Xamarin Studio and command line development.

We use one idea that is discussed more in the next chapter: We need a sequence of directory strings to look through. At this point we have only discussed sequences of individual characters. The variable `paths` contains a sequence of directory paths to check. In our case we make the sequence contain `"."`, the current directory, `".."`, the parent directory, and `Path.Combine("..", "..")`, the parent's parent. We make `paths` a static variable, so it is visible in all the functions in the class.

Then the sequence `paths` can be used in the `foreach` loop:

```
/// Find a directory containing the filename
/// and return the full file path, if it exists.
/// Otherwise return null.
public static string GetPath(string filename)
{
   foreach (string dir in paths) {
      string filePath = Path.Combine(dir, filename);
      if (File.Exists(filePath))
         return filePath;
   }
   return null;
}
```

For each directory path in `paths`, we create a `filePath` as if the file were in that directory. We return the first path that actually exists. We allow for the file to not be in any of the directories in `paths`. If we do not find it, we return `null` (no object).

For convenience, we have an elaboration, using `GetPath`, that directly opens the file to read:

```
/// Find a directory containing filename;
/// return a new StreamReader to the file
/// or null if the file does not exist.
public static StreamReader OpenReader(string filename)
{
   string filePath = GetPath(filename);
   if (filePath == null)
      return null;
   else
      return new StreamReader( filePath);
}
```

We have a variation on `GetPath` that just return the path to the directory containing the file. Here is the heading:

```
/// Return a directory conaining the filename, if it exists.
/// Otherwise return null.
public static string GetLocation(string filename)
```

This is useful in case you want to later write into the same directory that you read

from. You can get a location from GetLocation and then write to the same directory, creating a StreamWriter. Use the convenience function:

```
        /// Join the location directory and filename;
        /// open and return a StreamWriter to the file.
        public static StreamWriter OpenWriter(string location, string filename)
        {
            string filePath = Path.Combine(location, filename);
            return new StreamWriter(filePath);
        }
```

The entire FIO class is in fio/fio.cs

We illustrate the use of FIO functions in example file fio_usage/fio_usage.cs:

```
using System;
using System.IO;

namespace IntroCS
{
   class FIOTest
   {
      public static void Main(string[] args)
      {
         string sample = "sample.txt";
         string output = "output.txt";
         Console.WriteLine("Directory of {0}: {1}",
                           sample, FIO.GetLocation(sample));
         Console.WriteLine("Path to  {0}: {1}",
                           sample, FIO.GetPath(sample));
         StreamReader reader1 = FIO.OpenReader(sample);
         if (reader1 != null) {
            Console.Write(reader1.ReadToEnd());
            Console.WriteLine("First reader test passed.");
            reader1.Close();
         }

         StreamReader reader2 = FIO.OpenReader(FIO.GetLocation(sample),
                                               sample);
         if (reader2 != null) {
            Console.WriteLine("Second reader test passed.");
            reader2.Close();
         }
```

```
        StreamWriter writer1 = FIO.OpenWriter(FIO.GetLocation(sample),
                                          output);
        writer1.WriteLine("File in the same directory as {0}.", sample);
        writer1.Close();
        Console.WriteLine("Writer test passed; file written at \n {0}",
                        FIO.GetPath(output));
    }
  }
}
```

IF you look at the fio_usage project in our examples solution, you see that
`sample.txt` is a file in the project folder. The program ends up writing to a
new file in the same (project) directory. Remember that even though the new file
`output.txt` appears in the project directory, it does not appear in the Solution
pad unless you add it to the project. You can see it in the file system, and open it
if you like.

If you want to open a terminal/console and go to the project directory, you can
compile and run this program, and it will still work, even though the current di-
rectory has changed.

You are encouraged to make a library project fio in *your* work solution, copying
the fio.cs file. (Follow instruction like for ui in *Library Projects in Xamarin Studio
(Optional)*.) You can test your new library by also copying the fio_test project to
your solution. If you do this now and stick to one work solution, then you will be
ready for several later uses of `FIO`.

9.8.1 File Line Removal Exercise

Complete the function described below, and make a Main program and sample
file to test it:

```
/// Take all lines from reader that do not start with startToRemove
/// and copy them to writer.
static void FileLineRemoval(StreamReader reader, StreamWriter writer
                         char startToRemove)
```

For example, in Unix/Mac scripts lines starting with '#' are comment lines.
Making `startToRemove` be '#' would write only non-comment lines to the
writer.

9.9 Chapter Review Questions

1. After writing everything you want to a file through a `StreamWriter`, what do you still need to remember to do?

2. If you want to create a file path in an operating system independent way for file f.txt in directory d2, which is a subdirectory of d1, which is a subdirectory of the current directory, how would you do it?

3. Windows uses \ as a path separator. If you want to write a literal directly for a Windows path, what issue is there in C#?

4. In a file path, how do you refer to the parent directory of the current directory, without using the actual name of the parent directory?

5. If you are reading from a `StreamReader inFile`, what is logically wrong with the following:

```
if (inFile.ReadLine().Contains("!")) {
    Console.WriteLine(inFile.ReadLine() + "\n contains the symbol !"
}
```

ARRAYS

10.1 One Dimensional Arrays

10.1.1 Basic Syntax

A string is an immutable sequence of characters. Arrays provide more general sequences, with the same indexing notation, but with free choice of the type of the items in the sequence, and the ability to change the elements in the sequence.

For example, if we want the type for an array with `int` elements, it is `int[]`. In general for any element type, the type for an array of the element type is

> **type**`[]`

so

```
int[] a;
```

declares `a` to refer to an array containing `int` elements. You do *not* know how many elements will be allowed in this array from this declaration. We must give further information to create the corresponding array object. A new object can be created using the `new` syntax. An array must get a definite length, which can be a literal integer or any integer expression. For example

```
int[] a;
a = new int[4];
```

or combined with the declaration,

```
int[] a = new int[4];
```

creates an array that holds 4 integers. The elements of the array must get initial values. Numerical arrays get initialized to all 0's with this syntax.

For a variety of reasons, including bookkeeping by the compiler, the actual data for an array is *not* stored directly in the memory location allocated by the declaration. The array could have any number of items, and hence the memory requirements are not known at compile time. Like all other object (as opposed to primitive) types, what is actually stored at the memory location declared for `a` is a *reference* to the actual place where the data for the array is stored. In actual compiler implementation this reference is an address in memory. In diagrams we will illustrate object references with an arrow *pointing* to the actual location for the object's data. For example after `a` is initialized:

The small box beside `a` is meant to indicate the memory space allocated when `a` is declared. As you can see that space does not actually contain the array, but only a *reference* to the array, pointing to the actual sequence of data for the array. To make it easy to refer to the elements in the diagram, we also label the indices associated with each element, though they are not actual a part of what is stored in memory.

The general syntax to create a new array is

new **type** [*length*]

After the type, there are square brackets enclosing an expression for the length of the array - this length is unchangeable after creation.

The elements inside an array can to referenced with the same index notation used earlier for strings.

```
a[2]
```

refers to the element at index 2 (third element because of 0 based indexing).

Unlike with strings, this element can not only be read, but also be assigned to:

```
a[0] = 7;
a[1] = 5;
a[2] = 9;
a[3] = 6;
```

These four assignment statements would replace the original 0 values for each element in the array.

This is a verbose way to specify all array values. An array with the same final data could be created with the single declaration:

```
int[] b = {7, 5, 9, 6};
```


The list in braces ONLY is allowed as an initialization of a variable in a *declaration*, not in a later assignment statement. Technically it is an initializer, not an array literal.

Individual array elements can *both* be used in expressions, and be assigned to. Continuing with the earlier example code:

```
a[2] = 4*a[1] - a[3];
```

a[2] now equals 4*5 - 6 = 14.

Arrays, like strings, have a Length property:

```
Console.WriteLine(b.Length); // prints 4
```

Just as we saw that using a variable for an index was useful with strings, in practice array elements are almost always referred to with an index variable. A very common pattern is to deal with each element in sequence, and the syntax is the same as for a string. Print all elements of array b:

```
for (int i= 0; i < b.Length, i++) {
    Console.WriteLine(b[i]);
}
```

You could also use while syntax. The foreach syntax would be:

```
foreach( int x in b) {
    Console.WriteLine(x);
}
```

The `int` type for `x` matches the element type of the array `b`.

The shorter `foreach` syntax is not as general as the `for` syntax. For example, to print only the first *3* elements of b:

```
for(int i= 0; i < 3; i++) {
    Console.WriteLine(b[i]);
}
```

but the `foreach` syntax would not work, since it must process *all* elements.

Also use the `for` syntax to assign new values to the array elements, rather than just use the values in expressions:

```
for(int i= 0; i < b.Length; i++) {
    b[i] = 5*i;
}
```

Now the array `b` of our earlier examples (of length 4) would contain 0, 5, 10, and 15.

> **Warning:** There is no analog of *changing* the value of `b[i]` with a `foreach` loop. To change values in an array, we must assign to each location in the array by *index*. A `foreach` loop only provides the *value* of each sequence element for us to read.

We have had the array indices so far be given by a single symbol, which is the most common case in practice, but in fact what appears inside the square braces can be any `int` *expression*. Like parentheses, square brackets *delimit* the inside expression, which gets evaluated first, before the array value is looked up. Consider this csharp sequence:

```
csharp> int[] a = {5, 9, 15, -4};
csharp> int i = 2;
csharp> a[i];
15
```

This should be clear. Now think first, what should `a[i+1]` be?

...

```
csharp> a[i+1];
-4
```

In steps: `a[i+1]` is `a[2+1]` is `a[3]` is -4. Be careful, `a[i+1]` is *NOT* `a[i] + 1` (which would be 16).

The code above to print each element of an array performs a unified and possibly useful operation, so it would make sense to encapsulate it into a function. A function can take any type as a parameter, so an array type is perfectly reasonable! Above we printed each element of an array of integers. This time lets choose strings, so the formal parameter is an array of strings: `string[]`.

```
1      /// Print the strings in data, one per line.
2      public static void PrintStrings(string[] data)
3      {
4         foreach( string s in data) {
5            Console.WriteLine(s);
6         }
7      }
```

With this definition, the code fragment

```
string[] hamlet = {"To be", "or not", "to be!"};
PrintStrings(hamlet);
```

would print:

```
To be
or not
to be!
```

Here we are just reading the data from the array parameter. We will see that there are more wrinkles to array parameters in *References and Aliases*.

An array type can also be returned like any other type. Examine the function definition:

```
/// Return an array with string data obtained from the user.
/// The length of the array and the number of entries to
///   prompt the user for is n.
public static string[] InputNStrings(int n)
{
```

```
        string[] lines = new string[n];
        Console.WriteLine ("Enter {0} string(s).", n);
        for (int i = 0; i < n; i++) {
            lines[i] = UI.PromptLine("next string: ");
        }
        return lines;
    }
```

This code follows a standard pattern for functions returning an array:

- In order to return an array, we must *first create* a new array with the `new` syntax. We must set the proper length (n here).

- And we are not done with one line of creation: Since the array has multiple parts, we need a loop to assign all the values. We have a simple `for` loop to assign to each element in turn.

- Finally we must return the array that we created!

Follow Array Loop Exercise/Example

1. What is printed by this program? Play computer first to figure out.

```
1   using System;
2
3   class ArrayLoop1
4   {   //Play computer on this code and then test
5       public static void Main()
6       {
7           int[] a = {1, 2, 3}, b = {7, 2, 3, 5},
8                 c = {7, 0, 3, 2, 5};
9           Console.WriteLine(foo(a, b, 2));
10          Console.WriteLine(foo(c, b, 4));
11      }
12
13      static int foo(int[] x, int[] y, int n)
14      {
15          int k = 0;
16          for (int i = 0; i < n; i++)
17              if (x[i] == y[i]) {
18                  k++;
19              }
```

```
20          return k;
21       }
22    }
```

Then you can run example array_loop1/array_loop1.cs to check the results and see our table from playing computer included in the project, array_loop1/play_computer1.txt.

2. What is printed by this program? Play computer first to figure out. Be careful to keep the data current!

```
1   using System;
2
3   class ArrayLoop2
4   {  //Play computer on this code first and then test
5       public static void Main()
6       {
7           int[] a =  {5, 7, 6, 9, 8};
8           for (int i = 0; i < 3; i++) {
9               a[i+2] = a[i];
10          }
11          foreach (int x in a) {
12              Console.Write(x);
13          }
14          Console.WriteLine();
15      }
16  }
```

Then you can run example array_loop2/array_loop2.cs to check the results and see our table from playing computer included in the project, array_loop2/play_computer2.txt.

3. What is printed by this program? Play computer first to figure out.

```
1   using System;
2
3   class ArrayLoop3
4   {  //Play computer on this code first and then test
5       public static void Main()
6       {
7           string[] strArray = {"abcdefgh", "wxyz"};
8           for (int i = 2; i < 4; i++) {
9               foreach(string s in strArray) {
```

```
10                Console.Write(s.Substring(s.Length/i) + "/");
11              }
12            Console.WriteLine();
13          }
14        }
15    }
```

Then you can run example array_loop3/array_loop3.cs to check the results and see our table from playing computer included in the project, array_loop3/play_computer3.txt.

Sign Array Exercise/Example

Complete the code for this function:

```
/// Return an array contqining the sign (1, -1 or 0)
/// of each element of x.
/// For example if x contains elements 2, -5, 0, 7,
/// then return a new array containing 1, -1, 0, 1.
static int[] Signs1(int[] x)
```

and place it in a program with a main function that demonstrates it.

You can compare your solution with ours in sign_array1/sign_array1.cs.

10.1.2 Parameters to Main

The `Main` function may take an array of strings as parameter, as in example print_param/print_param.cs:

```
/// Demonstrate the use of command line parameters.
static void Main(string[] args)
{
    Console.WriteLine("There are {0} command line parameters.", args.Length
    foreach(string s in args) {
        Console.WriteLine(s);
    }

}
```

By convention, the formal parameter for `Main` is called `args`, short for arguments.

Compile and run the program from the command line. Run it again with some things at the end of the line like:

```
mono print_param.exe hi there 123
```

This should print for you:

```
There are 3 command line parameters.
hi
there
123
```

See what quoted strings do. Use command line parameters (with the quotes) `"hi there"` `123`. This should print for you:

```
There are 2 command line parameters.
hi there
123
```

You can simulate command line parameters inside Xamarin Studio:

1. Open the local popup menu for the project you are using.

2. Select Run With >

3. In the submenu select Custom Parameters.

4. That brings up a dialog where you can enter the desired command line parameters.

5. Optionally you can remember this setup by clicking on box in front of "Save this configuration as a custom execution mode". If you check it, you get a place to enter a Custom Mode Name.

6. End up clicking the Execute button.

7. If you set a Custom Mode, later when you get to the submenu after "Run With >", you will see your custom mode name to select!

Try it!

An alternative when you want to use command line parameters repeatedly is

---

1. Use Xamarin Studio for editing and compiling. To compile but not run, the command is **Build** rather than **Run**.

2. Meanwhile keep a console/terminal window open in the `Debug` directory, and enter execution commands there, including the command line parameters actually on the command line! Even if you have a long set of parameters, you can easily run your program multiple times with the same parameters by just pressing the up arrow key in the terminal when you see the next command line prompt, taking you back to the previous command (or keep going back several commands). This is also convenient if you want to slightly edit the parameters: you can edit a line that you redisplay from your command history.

3. Do not close this window until you are done with your session of executing from the command line. By using the same terminal window, you also save the history of all your runs. You can scroll the window up to see past executions.

If one run leads you to go back and fix a bug, go back to step one to build the program again, and continue executing in the same terminal window.

Modified Parameter Print Exercise

Modify a copy of print_param/print_param.cs to contain the earlier example function *PrintStrings*, and call it.

Command Line Adder Exercise

Write a program `adder.cs` that calculates and prints the sum of command line parameters, so if you make the command line parameters in Xamarin Studio be

```
2 5 22
```

then the program prints 29.

Do try running from the command line: If you compiled with Xamarin Studio, that means going down to the bin/Debug directory. Recall Xamarin Studio for Windows produces a Windows executable, not a Mono file, so you can run

```
adder 2 5 22
```

but on a Mac you need to run with mono:

```
mono adder.exe 2 5 22
```

10.1.3 String Method Split

A string method producing an array:

string[] Split(char *separator*) Returns an array of substrings from
this string. They are the pieces left after chopping out the separator charac-
ter from the string. A piece may be the empty string. Example:

```
csharp> var fruitString = "apple pear banana";
csharp> string[] fruit = fruitString.Split(' ');
csharp> fruit;
{ "apple", "pear", "banana" }
csharp> fruit[1];
"pear"
csharp> var s = "  extra   spaces ";
csharp> s.Split(' ');
{ "", "", "extra", "", "", "spaces", "" }
```

Note: The response with the list in braces is a purely *csharp* convention for dis-
playing sequences for the user. There is no corresponding string displayed by C#
Write commands. Also see that the string is split at *each* separator, even if
that produces empty strings.

Split is useful for parsing a line with several parts. You might get a group of
integers on a line of text, for instance from:

```
string input = UI.PromptLine(
    "Please enter some integers, separated by single spaces: ");
```

To extract the numbers, you want to the separate the entries in the string with
Split, *and* you probably want further processing: If you want them as integers,
not strings, you must convert each one separately.

It is useful to put this idea in a function. See the type returned. It is an array
int[] for the int results:

```csharp
/// Return ints taken from space separated integers in a string.
public static int[] IntsFromString1(string input)
{
    string[] integers = input.Split(' ');
    int[] data = new int[integers.Length];
    for (int i=0; i < data.Length; i++)
        data[i] = int.Parse(integers[i]);
    return data;
}
```

In a call to `IntsFromString1("2 5 22")`, `integers` would be an array containing strings `"2"`, `"5"`, and `"22"`. We need the conversions to `int` to go in a new array that we call `data`. We must set its length, which will clearly be the same as for `integers`, `integers.Length`. To assign elements into `data` we need a loop providing indices, like the `for` loop provided. Then for each index, we parse a string in `integers` into an `int`, and place the `int` in the corresponding location in `data`. We need to return `data` at the end to make it accessible to the caller.

Again we use the basic pattern for returning an array.

Dealing with arrays is hard for many students for several reasons:

- You have new array declaration and creation syntax.

- Array are compound objects, so there is a lot to think about.

- Loops are hard for many people, and you almost always deal with loops.

- You usually must deal with index variables, and there are many patterns.

The last point is significant, so it is important to note the special pattern in the example above:

Note: The use of the same index variable for more than one array is a standard way to have *related* entries in *corresponding* positions in the arrays.

We will introduce a refinement of this function in the *IntsFromString Exercise*. It will rely on a more complicated index-handling pattern.

NewUpper Exercise

Complete the definition for

```
/// Return an array that is the same as data
/// except all strings are in upper case.
public static string[] NewUpper(string[] data)
```

and write a Main driver to demonstrate it. Use the example function *PrintStrings* in your demonstration.

10.1.4 References and Aliases

Object variables, like arrays, are references, and this has important implications for assignment.

With a primitive type like an int, an assignment copies the data:

$$\text{int b = 7;} \quad \text{b} \boxed{7}$$

$$\text{int d = b;} \quad \text{d} \boxed{7}$$

In the diagram, the contents of the memory box labeled b is copied to the memory box labeled d. The value of d starts off equal to the value of b, but can later be changed independently.

Contrast an assignment with arrays. The value that is copied is the *reference*, not the array data itself, so both end up pointing at the *same* actual array:
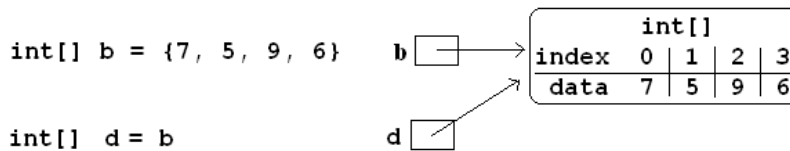
Hereafter, array assignments like:

```
b[2] = -10;
d[1] = 55;
```

would both change the *same* array. Now `b` and `d` are essentially names for the same thing (the actual array). The technical term matches English: The names are *aliases*.

This may seem like a pretty silly discussion. Why bother to give two different names to the same object? Isn't one enough? In fact it is very important in function/method calls. An array reference can be passed as an actual value, and it is the array *reference* that is copied to the formal parameter, so the formal parameter name is an **alias** for the actual parameter name.

Note: If an array passed as a parameter to a method has elements changed in the method, then the change affects the actual parameter array. The change *remains* in the actual parameter array *after* the method has terminated.

For example, consider the following function:

```csharp
// Modify a by multiplying all elements by multiplier.
static void Scale(int[] a, int multiplier)
{
   for (int i = 0; i < a.Length; i++) {
      a[i] *= multiplier;  // or:  a[i] = a[i] * multiplier
   }
}
```

The fragment:

```csharp
int[] nums = {2, 4, 1};
Scale(nums, 5);
```

would *change* nums, so it ends up containing elements 10, 20, and 5.

AllToUpper Exercise

Complete the function with this heading:

```csharp
/// Modifiy the array data so
///    all strings are in upper case.
public static void AllToUpper(string[] data)
```

Write a `Main` method to demonstrate it. Use the example function *PrintStrings* to show off your result.

Sign Array II Exercise/Example

Create a variation on *Sign Array Exercise/Example* with a function with heading

```
/// Mutate array x, replacing each element by
/// its sign (1, -1 or 0)
/// For example if array a contains elements  2, -5, 0, 7,
/// then after the call Signs2(a), a contains 1, -1, 0, 1.
static void Signs2(int[] x)
```

and a main function to demonstrate it.

You can compare your solution with ours in sign_array2/sign_array2.cs.

10.1.5 Anonymous Array Initialization

Sometimes you only want to use an array with specific values as a parameter to a function. You could write something like

```
int[] temp = {3, 1, 7};
SomeFunc(temp);
```

but if `temp` is never going to be referenced again, you can do this without using a name:

```
SomeFunc(new int[] {3, 1, 7});
```

Like with the use of `var`, the compiler can infer the type of the array, and the last example could be shortened to

```
SomeFunc(new[] {3, 1, 7});
```

It is essential to include the `new int[]` or `new[]` *in addition to* the `{3, 1, 7}`.

Such an approach could also be used if you want to return a fixed length array, where you have values for each parts, as in:

```
int minVal = ...
int maxVal = ...
// ...
return new[] {minVal, maxVal};
```

Testing NewUpper Exercise/Example

Elaborate *NewUpper Exercise* so your `Main` method calls `NewUpper` with an anonymous array as part of the demonstration.

You can see our code for all the string array exercises in example project string_array/string_array.cs, and with the `Main` demonstration method in string_array/string_array_demo.cs.

10.1.6 Default Initializations

Did you notice that when the first example array of integers was created, it was filled with zeros? It is a safety feature of C# that the internal fields of objects always get a specific value, not random data. Here are the defaults:

Table 10.1: Default Values

Type	Value
primitive numeric types	0
bool	false
all object types	null

> **Warning:** An array with elements of object type, like `string[]`, without a specific initializer, gets initialized to all `null` values. The creation is totally legal, but if you try to use the created value, like
>
> ```
> string[] words = new string[10];
> Console.WriteLine(words[0].Length);  // run time error here
> ```
>
> The error is because `null` is not an object - it does not have a `Length` property. If, for example, you want an array of empty strings you would need to initialize it with a loop:
>
> ```
> string[] words = new string[10];
> for (int i = 0; i < words.Length; i++) {
>    words[i] = "";
> }
> ```

10.1.7 Array Examples and Exercises

We have been using array index variables all though this chapter. We have been getting you started in situations where they all just advanced continually in a `for` loop heading. The fanciest situations have been where the same index is used to reference more than one array in parallel.

Now that you have some experience, this section will include a variety of exercises where array index variables need to be manipulated in fancier ways. Consider this heading:

```
/// Return a new array with all the 0's that are in
/// data removed.  If data contains 0, 3, 0, 0, 5, 9
/// then an array containing 3, 5, 9 is returned.
public static int[] NoZeros(int[] data)
```

We have a starting array `data` and we need to create an ending array, but the corresponding nonzero data is *not* at corresponding index values in `data`!

Since we are returning a new array, we need to create it, and for that we need a length. How would you do that by hand? Go through the original array, look at individual elements, and count the nonzero ones. We can do a counting loop Say we put our count into the variable `countNonZero`. Then create a new `int` array, say `notzero`, with the proper length.

The next part is new. Clearly we need to get non-zero values from the original array `data` and put them in the other array, `notzero`. As we said, the array indices are not in sync. That means we are going to need to deal with their indices separately: The index in `data` is not going to relate directly to the index in `notzero`.

We could just have a separate index variable for each array. Think about `data`: We do want to go through it sequentially, and we are only *reading* the sequential values, so we can actually use a `foreach` loop and not keep track of that index directly at all!

On the other hand we need to assign values *into* `notzero`, and hence we will need to refer to an index variable for `notzero`, say i.

However, we cannot just assign the index values in a `for` loop heading as we have been before! We have to be more careful and think when and how does i change?

This might be a good place to do this by hand, for instance with the sample data in

the function documentation. Keep track of what `i` should be as you iterate through the elements of `data`, one step at a time: How do you change `i` and when? You are *encouraged* to stop and actually do this manually, on paper, and think before going on....

You should see that:

- We start by being ready to fill the place at index 0 in `notzero`.

- We only copy a non-zero element of `data`, so we need an `if` statement in the body again.

- Each such non-zero number is placed after the last number we copied into `notzero`.

- This means that each time we copy an element to `notzero` we advance `i`!

If you get those ideas together, hopefully you can write the needed code. Our version is:

```csharp
        /// Return a new array with all the 0's that are in
        /// data removed.  If data contains 0, 3, 0, 0, 5, 9
        /// then an array containing 3, 5, 9 is returned.
        public static int[] NoZeros(int[] data)
        {
            int countNonZero = 0;
            foreach (int n in data) {  //find new length
                if (n != 0) {
                    countNonZero++;
                }
            }
            int[] notzero = new int[countNonZero];
            int i = 0;  // index where to put the next value
            foreach(int n in data) {
                if (n != 0) { // copy non-zero elements
                    notzero[i] = n;
                    i++;
                }
            }
            return notzero;
        }
```

Adding a `Main` demostration method, you get our full example remove_zeros/remove_zeros.cs.

Initialization Exercise

1. In the `NoZeros` function above, what are the values in the array `notzero` just after line 12 is executed?

2. In the *NewUpper Exercise* or our version of `NewUpper` in string_array/string_array_demo.cs consider the execution of the `NewUpper` function immediately after you first create the string array that you are going to later return. Right then, what are the element values in that array?

ExtractItems Exercise

A string intended to indicate a sequence of items could be like in the discussion above of *IntsFromString1*. As illustrated there, individual items are separated out neatly with `Split`. If you want to act on a user-generated string, it is probably better to allow more leeway: Commas are often used to separate items or comma with blank, or several blanks.

In this exercise write a version that will accept all those variations and return an array of non-empty strings, without the commas or blanks. Complete this function:

```
/// Return an array of non-empty strings that are separated
/// in the original string by any combination of commas and blanks.
/// Example:  ExtractItems(" extra  spaces,plus, more, ") returns an
/// array containing {"extra", "spaces", "plus", "more"}
public static string[] ExtractItems(string s)
```

Hints: It is possible to deal with more than one separator character, but the simplest thing likely is to use string method `Replace` and just replace all the commas by spaces. If you then `Split` on each space you get all the non-empty strings that you want *and* maybe a number of empty strings. You need to create a final array with just the nonempty strings from the split. When you create the array to be returned, you need know its size. Then populate it with just the nonempty string pieces. Handling the indices for the new array also adds complication.

IntsFromString Exercise

Write a function `IntsFromString` with a corresponding signature and intent like *IntsFromString1*, but make it more robust by allowing all the separator combinations of `ExtractItems` from the last exercise, so `IntsFromString("2, 33 4,55 6 77 ")` returns an array containing `int` values 2, 33, 4, 55, 6, 77. (Don't reinvent the wheel: call `ExtractItems`.) Also write a `Main` function so you can demonstrate the use of `IntsFromString`.

Trim All Exercise

Write a program `trimmer.cs` that includes and tests a function with heading:

```
// Trim all elements of a and replace them in the array.
//  Example: If a contains {" is  ", " it", "trimmed?   "}
//  then after the function call the array contains
//  {"is", "it", "trimmed?"}.
static void TrimAll(string[] a)
```

Count Duplicates Exercise

Write a program `count_dups.cs` that includes and tests a function with heading:

```
// Return the number of duplicate pairs in an array a.
// Example: for elements 2, 5, 1, 5, 2, 5
// the return value would be 4 (one pair of 2's three pairs of 5's.
public static int dups(int[] a)
```

Mirror Array Exercise

Write a program `make_mirror.cs` that includes and tests a function with heading:

```
// Create a new array with the elements of a in the opposite order.
// {"aA", "bB", "cC"} produces a new array {"cC", "bB", "aA"}
public static string[] Mirror(string[] a)
```

Reverse Array Exercise

Write a program `reverse_array.cs` that includes and tests a function with heading:

```
// Reverse the order of array elements
// If array a first contains "aA", "bB", "cC",
// than it ends up containing "cC", "bB", "aA".
public static void Reverse(string[] a)
```

Do this *without* creating a second array. (There is a trick here.)

Histogram Exercise

Write a program `make_histogram.cs` that includes and tests a function with heading:

```
// Return a histogram array counting repetitions of values
// start through end in array a.  The count for value start+i
// is at index i of the returned array, starting at i == 0.
// For example:
// Histogram(new int[]{2, 0, 3, 5, 3, 5}, 2, 5) counts how
// many times the numbers 2 through 5, inclusive, occur in
// the original array, and returns a new array containing
// {1, 2, 0, 2}, that is, 1 2, 2 3's, 0 4's, and 2 5's. The
// count of 2's appears as the first (0th) element of the
// returned array, the count of 3's as the second, etc.
// Similarly, Histogram(new int[]{2, 0, 3, 5, 3, 5}, -1, 1)
// returns the new array {0, 1, 0},
// that is, 0 -1's, 1 0, and 0 1's.
public static int[] Histogram(int[] a, int start, int end)
```

This problem clearly requires you to loop through all the elements of array `a`. You should *not* need any further nested loop.

Histogram Interval Exercise

This is a slight elaboration of the previous problem, where you count entries in intervals, not just of width 1.

Write a program `make_histogram2.cs` that includes and tests a function with heading:

```
// Return a histogram array counting repetitions of values
// in array a in the n half-open intervals [start, start + width),
// [start+width, start+2*width), ... [
// [start + (n-1)*width, start + n*width) .  The counts for
// each of the n intervals, in order, goes in the returned array
// of length n.  For example
// Histogram(new[]{89, 69, 100, 83, 99, 81}, 60, 10, 5)
// would return an array containing counts 1, 0, 3, 1, 1,
// for 1 in sixties, 0 in seventies, 3 in eighties, 1 in nineties,
// and 1 in range 100 through 109.
public static int[] HistogramIntervals(int[] a, int start,
                                       int width, int n)
```

The previous exercise version `Histogram(a, start, end)` would return the same result as `HistogramIntervals(a, start, 1, end-start+1)`.

Again, the only loop needed should be to process each element of `a`.

Power Table Exercise 2

Write a program `power_table2.cs` ` producing a table much like *Power Table Exercise*, with right-justified columns, but this time make each separate column have the minimum width necessary - so there is a single space (and no less) in front of some entry in *each* column, except the first. Be careful: take the heading widths into account; the parameter limits are important, too; test them:

```
/// Print a table of powers of positive integers.
/// Assume 1 <= nMax <= 14, 1 <= powerMax <= 10
/// Example: output of PowerTable(4, 5)
/// n^1 n^2 n^3 n^4  n^5
///   1   1   1   1    1
///   2   4   8  16   32
///   3   9  27  81  243
///   4  16  64 256 1024
public static void PowerTable(int nMax, int powerMax)
```

10.2 Musical Scales and Arrays

Music in the western classical tradition uses a twelve-tone *chromatic* scale. Any of the tones in this scale can be the basis of a major scale. Most musicians (especially pianists) learn the C-major scale in the early days of study, owing to the ability to play this scale entirely with the ivory (white) keys.

The following declaration shows how to initialize an array consisting of the twelve tones of the chromatic scale, starting from the C note.

```
static string[] tones = { "C", "C#", "D", "D#", "E", "E",
                          "F#", "G", "G#", "A", "A#", "B"
                        };
```

Even if you're not a musician, learning the basic principles is fairly straightforward.

The well-known C-major scale, which is often sung as:

```
Do Re Mi Fa So La Ti Do
```

has the following progression:

```
C  D  E  F  G  A  B  C
```

This progression is known as the *diatonic major* scale. If you look at the `tones` array, you can actually figure out the intervals associated with this array:

```
C + 2 = D
D + 2 = E
E + 1 = F
F + 2 = G
G + 2 = A
A + 2 = B
B + 1 = C
```

So given any starting note, the major scale can be *generated* from the intervals (represented as an array).

So, for example, if you want the F-major scale, you can get it by starting at F and applying the steps of 2, 2, 1, 2, 2, 2, 1:

```
F + 2 = G
G + 2 = A
A + 1 = B' (flat) a.k.a. A#)
B'+ 2 = C
C + 2 = D
D + 2 = E
E + 1 = F
```

So this is the F-major scale:

```
F G A B' C D E F
```

We begin by creating a helper function, `FindTone()`, which does a linear search
to find the key of the scale we want to compute. The aim is to make it easy for the
user to just specify the key of interest. Then we can use this position to compute
the scale given the major (or minor, covered shortly) interval array.

```
1         static int FindTone(string key) {
2             for (int i=0; i < tones.GetLength(0); i++) {
3                 if (key == tones[i])
4                     return i;
5             }
6             return -1;
7         }
```

To see what this function does, pick your favorite key (C and G are very common
for beginners).

- `FindTone("C")` gives 0, the first position in the `tones` array.

- `FindTone("G")` gives 8.

For example, C is the first note in the array of tones, so `FindTone("C")` would
give us 0. `FindTone("F")` would give us 6.

So let's take a look at `ComputeScale()` which does the work of computing
a scale, given a key and an array of steps. The scale array is allocated by the
`Main()` method, primarily to allow the same array to be used repeatedly for
calculating other scales.

```
1         static void ComputeScale(string key, int[] steps, int[] scale) {
2             int tonePosition = 0;
3             int startTone;
```

```
 4
 5            startTone = FindTone(key);
 6            if (startTone < 0)
 7                return;
 8            if (steps.GetLength(0)+1 != scale.GetLength(0))
 9                return;
10            tonePosition = startTone;
11            for (int i=0; i < steps.GetLength(0); i++) {
12                scale[i] = tonePosition % tones.GetLength(0);
13                tonePosition += steps[i];
14            }
15        }
```

1. The first thing to note is the *setup* of this code. We're going to keep the startTone (obtained by calling FindTone()) and tonePosition, which is the note we are presently visiting in the tones array.

2. Remember that every scale (e.g. C, D, F#, etc.) can always be obtained by looking at tones and using the appropriate intervals (the steps parameter) to compute the next note, given a current note.

3. We do some simple checks in line 6 (to ensure that a valid key was specified by the caller) and in line 8 to ensure that the number of steps + 1 is the length of the scale–and the length of the scale is 8. (We technically don't have to limit the scale to 8, because scales can keep going until you run out of playable notes on the instrument.)

4. We'll now start at the initial position (where we found the base note of the key) and enter a for loop to compute all of the notes in the scale. This loop iterates over the entries in the steps array to decide what the next note is.

5. The next note in the scale, scale[i] is computed by taking tonePosition % tones.GetLength(0). We need to do this, because in most scales, you will eventually end up "falling off the end" of the tones array, which mens that you need to continue computing notes from the *beginning* of the array. You can inspect this for yourself by picking a scale (say, B) that is starting at the end of the tones array. This means you will need to go to the beginning of the array to get C# (which is 2 tones away from B).

6. The next note is found by adding steps[i] to tonePosition.

The following function writes the scale out (rather naively) by just printing the

---

notes from our existing `tones` array.

```
static void WriteScale(int[] scale) {
    foreach (int i in scale) {
        Console.Write ("{0} ", tones[i]);
    }
    Console.WriteLine ();
}
```

We say that the output is *naive* because any musician will tell you that a scale should be printed in a normalized way. For example, the F-major scale (shown above in our earlier explanation) is never written with A# as one of its notes. It is written as B-flat. It's easy to manage the various cases by consulting the circle of fifths, which gives us guidance on the number of flats/sharps each scale has.

Lastly, we put this all together.

```
public static void Main (string[] args)
{
    int[] scale = new int[8];
    int[] major = { 2, 2, 1, 2, 2, 2, 1 };
    int[] minor = { 2, 1, 2, 2, 1, 2, 2 };

    string name = args[0];  // need command line tone name
    Console.WriteLine("{0} major scale", name);
    ComputeScale(name, major, scale);
    WriteScale(scale);
    Console.WriteLine("{0} minor scale", name);
    ComputeScale(name, minor, scale);
    WriteScale(scale);
}
```

This `Main()` method shows how to set up the steps for both major and minor scales. We've already explained how to express the steps of a major scale. The minor scale basically drops the 3rd and 7th by a semitone (a single step), which gives us a different pattern.

You can run this program to see the major and minor scales.

10.3 Linear Searching

In this section, we'll take a look at how to search for a value in an array. Although a fairly straightforward topic, it is one that comes up repeatedly in programming.

10.3.1 Linear Search

By far, one of the most common searches you will see in typical programs. It also happens to be one of the more *misused* searches, which is another reason we want you to know about it.

Here is the code from example searching/searching.cs to perform a linear search for an integer in an array:

```
1  /// Return the index of the first position in data
2  /// where item appears, or -1 if item does not appear.
3  public static int IntArrayLinearSearch(int[] data, int item)
4  {
5      int N=data.Length;
6      for (int i=0; i < N; i++) {
7          if (data[i] == item) {
8              return i;
9          }
10     }
11     return -1;
12 }
```

Here's what it does:

- In lines 5-6 we set up a loop to go from 0 to N-1. We often use N to indicate the size of the array (and it's much easier to type than `data.Length`.

- In line 7, we see whether we found a match for the item we are searching. If we find the match, we immediately leave the loop by returning the position where it was found.

- It is worth noting here that the array, `data`, may or my not be in sorted order. So our search reports the first location where we found the value. It is entirely possible that the more than one position in the array contains the matching value. If you wanted to find the next one, you could modify the `IntArrayLinearSearch()` method to have a third parameter, `start`,

that allows us to continue searching from where we left off. It might look something like the following:

```
/// Return the first index >= start in data where
/// item appears, or -1 if item does not appear there.
public static int IntArrayLinearSearch(int[] data, int item,
                                       int start)
{
   int N=data.Length;
   if (start < 0) {
      return -1;
   }
   for (int i=start; i < N; i++) {
      if (data[i] == item) {
         return i;
      }
   }
   return -1;
}
```

The following code in `Main` of searching/searching_demo.cs demonstrates how to use the linear search:

```
string input = UI.PromptLine(
   "Please enter integers, separated by spaces and/or comma: ");
int[] data = ExtractFromString.IntsFromString(input);
for (int i=0; i < data.Length; i++) {
   Console.WriteLine("data[{0}]={1}", i, data[i]);
}
string prompt =
   "Please enter a number to find (blank line to end): ";
input = UI.PromptLine(prompt);
while (input.Length > 0) {
   int searchItem = int.Parse(input);
   int searchPos = UI.PromptIntInRange(
                      "At what position should the search start? ",
                      0, data.Length);
   int foundPos =
      Searching.IntArrayLinearSearch(data,searchItem, searchPos);
   if (foundPos < 0) {
      Console.WriteLine("Item {0} not found", searchItem);
   }
```

```
20        else {
21            Console.WriteLine("Item {0} found at position {1}",
22                              searchItem, foundPos);
23        }
24        input = UI.PromptLine(prompt);
25    }
```

In this example, we ask the user to enter all the data for the array on one line. To convert the string to an int array we use the result of the *IntsFromString Exercise* that we put in searching/extract_from_string.cs.

To allow easy termination of the testing loop, we do not use `PromptInt` for `searchItem`, because any `int` could be the search target. By using `PromptLine`, we can allow an empty string as the response, and we test for that to terminate the loop.

The rest is mostly self-explanatory.

10.4 Sorting Algorithms

Sorting algorithms represent foundational knowledge that every computer scientist and IT professional should at least know at a basic level. And it turns out to be a great way of learning about why arrays are important.

In this section, we're going to take a look at a number of well-known sorting algorithms with the hope of sensitizing you to the notion of *performance*–a topic that is covered in greater detail in courses such as algorithms and data structures.

This is not intended to be a comprehensive reference at all. The idea is to learn how these classic algorithms are coded in the teaching language for this course, C#, and to understand the essentials of analyzing their performance, both theoretically and experimentally. For a full theoretical treatment, we recommend the outstanding textbook by Niklaus Wirth *[WirthADP]*, who invented the Pascal language. (We have also adapted some examples from Thomas W. Christopher's *[TCSortingJava]* animated sorting algorithms page.

The sorts and supporting functions are all in sorting/sorting.cs, but we start one bit at a time:

10.4.1 Exchanging Array Elements

We'll begin by introducing you to a simple method, whose only purpose in life is to swap two data values at positions m and n in a given integer array:

```
1      /// Exchange the elements of data at indices m and n.
2      public static void Exchange(int[] data, int m, int n)
3      {
4          int temporary = data[m];
5          data [m] = data[n];
6          data [n] = temporary;
7      }
```

For example if we have an array nums, shown with indices:

```
 nums: -1   8 11 22   9 -5   2
index:  0   1  2  3   4  5   6
```

Then after Exchange(nums, 2, 5) the array would look like

```
 nums: -1   8 -5 22   9 11   2
index:  0   1  2  3   4  5   6
```

In general, swapping two values in an array is no different than swapping any two integers. Suppose we have the following integers a and b:

```
int a, b;
int t;

a = 25;
b = 35;
t = a;
a = b;
b = t;
```

After this code does its job, the value of a would be 35 and the value of b would be 25.

So in the Exchange() function above, if we have two different array elements at positions m and n, we are basically getting each value at these positions, e.g. data[m] and data[n] and treating them as if they were a and b in the above code.

You might find it helpful at this time to verify that the above code does what we're saying it does, and a good way is to type it directly into the C# interpreter (csharp) so you can see it for yourself.

The `Exchange()` function is vital to all of the sorting algorithms in the following way. It is used whenever two items are found to be out of order. When this occurs, they will be *swapped*. This doesn't mean that the item comes to its final resting place in the array. It just means that for the moment, the items have been reordered so we'll get closer to having a sorted array.

Let's now take a look at the various sorting algorithms.

10.4.2 Bubble Sort

The Bubble Sort algorithm works by repeatedly scanning through the array exchanging adjacent elements that are out of order. Watching this work with a strategically-placed `Console.WriteLine()` in the outer loop, you will see that the sorted array grows right to left. Each sweep picks up the largest remaining element and moves to the right as far as it can go. It is therefore not necessary to scan through the entire array each sweep, but only to the beginning of the sorted portion.

We define the number of *inversions* as the number of element pairs that are out of order. They needn't be adjacent. If `data[7] > data[16]`, that's an inversion. Every time an inversion is required, we also say that there is corresponding data *movement*. If you look at the `Exchange()` code, you'll observe that a swap requires three movements to take place, which happens very quickly on most processors but still amounts to a significant cost.

There can be at most $N \cdot \frac{N-1}{2}$ inversions in the array of length N. The maximum number of inversions occurs when the array is sorted in reverse order and has no equal elements.

Bubble Sort exchanges only adjacent elements. Each such exchange removes precisely one inversion; therefore, Bubble Sort requires $O(N^2)$ exchanges.

```
1    public static void IntArrayBubbleSort (int[] data)
2    {
3        int N = data.Length;
4        for (int j=N-1; j>0; j--) {
5            for (int i=0; i<j; i++) {
```

```
6              if (data[i] > data[i + 1]) {
7                  Exchange (data, i, i + 1);
8              }
9          }
10      }
11    }
```

10.4.3 Selection Sort

The Selection Sort algorithm works to minimize the amount of data movement, hence the number of Exchange() calls.

```
1        /// Among the indices >= start for data,
2        /// return the index of the minimal element.
3        public static int IntArrayMin (int[] data, int start)
4        {
5            int N = data.Length, minPos = start;
6            for (int pos=start+1; pos < N; pos++)
7                if (data[pos] < data[minPos]) {
8                    minPos = pos;
9                }
10           return minPos;
11       }
12
13       public static void IntArraySelectionSort (int[] data)
14       {
15           int N = data.Length;
16           for (int i=0; i < N-1; i++) {
17               int k = IntArrayMin(data, i);
18               if (i != k) {
19                   Exchange(data, i, k);
20               }
21           }
22       }
```

It's a remarkably simple algorithm to explain. As shown in the code, the actual sorting is done by a function, IntArraySelectionSort(), which takes an array of data as its only parameter, like Bubble sort. The way Selection Sort works is as follows:

1. An outer loop visits each item in the array to find out whether it is the

minimum of all the elements after it. If it is not the minimum, it is going to be swapped with whatever item in the rest of the array is the minimum.

2. We use a helper function, `IntArrayMin()` to find the position of the minimum value in the rest of the array. This function has a parameter, `start` to indicate where we wish to begin the search. So as you can see from the loop in `IntArraySelectionSort()`, when we start with position `i`, and we compare to the later elements from position `i + 1` to the end of the array, updating the position of the smallest element so far.

An illustration to accompany the discussion:

```
 data: 12   8 -5 22   9   2
index:  0   1   2   3   4   5
        i       k
```

The first time through the loop, `i` is 0, and `k` gets the value 2, since data[2] is -5, the smallest element. Those two positions get swapped.

```
 data: -5   8 12 22   9   2
index:  0   1   2   3   4   5
```

The next time through the loop i is 1 and k becomes 5:

```
 data: -5   8 12 22   9   2
index:  0   1   2   3   4   5
            i               k
```

After the swap:

```
 data: -5   2 12 22   9   8
index:  0   1   2   3   4   5
```

and so on. Here is the data after each of the last three swaps:

```
-5   2   8 22   9 12
```

```
-5   2   8   9 22 12
```

```
-5   2   8   9 12 22
```

Consider the first call to `IntArrayMin`.

```
 data: 12   8 -5 22   9   2
index:  0   1   2   3   4   5
```

Initially `minPos` is 0. Here are the changes for each value of `pos`:

```
pos=1:   8 = data[1] < data[0] = 12, so minPos becomes 1
pos=2:  -5 = data[2] < data[1] = 8, so minPos becomes 2
pos=3: 22 = data[3] is not < data[2] = -5, so minPos still 2
pos=4:   9 = data[4] is not < data[2] = -5, so minPos still 2
pos=5:   2 = data[5] is not < data[2] = -5, so minPos still 2
```

and 2 gets returned, and we ge the swap

```
 data: -5   8 12 22   9   2
index:  0   1   2   3   4   5
```

The next call to `IntArrayMin` has `start` as 1, so `minPos` is initially 1, and we compare to the elements of data at index 2, 3, 4, and 5....

We won't do the full algorithmic analysis here. Selection Sort is interesting because it does most of its work through *comparisons*, with the same number of them no matter how the data are ordered, exactly $N \cdot \frac{N-1}{2}$, which is $O(N^2)$ The number of *exchanges* is O(N). The comparisons are a non-trivial cost, however, and do show in our own performance experiments with randomly-generated data.

Shuffle Exercise

Complete the `Shuffle` function and add a `Main` method to test it:

```
/// Shuffle the elements of an array into random positions,
/// changing the array.  An array containing
/// 2, 5, 7, 7, 7, 9 *might* end up in the order
/// 7, 7, 2, 9, 7, 5.
static void Shuffle(int[] a)
```

Use a Random and do something close to a reverse of selection sort, using `Exchange` with a random position.

10.4.4 Insertion Sort

In the Insertion Sort algorithm, we build up a longer and longer sorted list from the bottom of the array. We repeatedly insert the next element into the sorted part of the array by sliding it down (using our familiar `Exchange()` method) to its proper position.

```
1        public static void IntArrayInsertionSort (int[] data)
2        {
3           int N = data.Length;
4           for (int j=1; j<N; j++) {
5              for (int i=j; i>0 && data[i] < data[i-1]; i--) {
6                 Exchange (data, i, i - 1);
7              }
8           }
9        }
```

Consider the earlier example array as we illustrate some of the steps. I use the symbol '-' for an element that we know to be in the sorted list at the beginning of the array, and '@' over the next one we are trying to insert *at* the right position. We start with a one-element sorted list and try to position the second element:

```
          -   @
 data: 12   8  -5  22   9   2
index:  0   1   2   3   4   5
```

After each outer loop in sequence we end up with:

```
          -   -   @
 data:  8  12  -5  22   9   2
index:  0   1   2   3   4   5


          -   -   -   @
 data: -5   8  12  22   9   2
index:  0   1   2   3   4   5


          -   -   -   -   @
 data: -5   8  12  22   9   2
index:  0   1   2   3   4   5


          -   -   -   -   -   @
 data: -5   8   9  12  22   2
```

```
index:   0   1   2   3   4   5


         -   -   -   -   -   -
 data:  -5   2   8   9  12  22
index:   0   1   2   3   4   5
```

Let us illustrate several times just through the inner loop, the first time, when the 8 is moved into position from index j = 1, so i starts at 1. We show the letter i over the data at index i, and show the comparison test to be done with a > with a '?' over it.

```
           ? i
 data: 12 > 8  -5   22    9    2    1>0 and 12>8: true, so swap, loop
index:  0    1    2    3    4    5


         i
 data:  8   12  -5   22    9    2    0>0 false, skip comparison of data, end loop
index:  0    1    2    3    4    5
```

Let us also illustrate at a later time through the inner loop, when the 9 is moved into position from index j = 4, so i starts at 4.

```
                       ? i
 data: -5    8   12   22 > 9    2   4>0 and 22>9: true, so swap, loop
index:  0    1    2    3    4    5


                    ? i
 data: -5    8   12 > 9   22    2   3>0 and 12>9: true, so swap, loop
index:  0    1    2    3    4    5


                 ? i
 data: -5    8 > 9   12   22    2   2>0 and 8>9: false, end inner loop
index:  0    1    2    3    4    5
```

The 9 started at index j = 4, and now the list is sorted up through index 4.

This will require as many exchanges as Bubble Sort, since only one inversion is removed per exchange. So Insertion Sort also requires $O(N^2)$ exchanges. On average Insertion Sort requires only half as many comparisons as Bubble Sort, since the average distance an element must move for random input is one-half the length of the sorted portion.

10.4.5 Shell Sort

Shell Sort is basically a trick to make Insertion Sort run faster. If you take a quick glance at the code and look beyond the presence of two additional *outer loops*, you'll notice that the code looks very similar.

Since Insertion Sort removes one inversion per exchange, it cannot run faster than the number of inversions in the data, which in worst case is $O(N^2)$. Of course, it can't run faster than N, either, because it must look at each element, whether or not the element is out of position. We can't do any thing about the lower bound O(N), but we can do something about the number of steps to remove inversions.

The trick in Shell Sort is to start off swapping elements that are further apart. While this may remove only one inversion sometimes, often many more inversions are removed with intervening elements. Shell Sort considers the subsequences of elements spaced k elements apart. There are k such sequences starting at positions 0 through k-1 in the array. In these sorts, elements k positions apart are exchanged, removing between 1 and 2(k-1)+1 inversions.

Swapping elements far apart is not sufficient, generally, so a Shell Sort will do several passes with decreasing values of k, ending with k=1. The following examples experiment with different series of values of k.

In this first example, we sort all subsequences of elements 8 apart, then 4, 2, and 1. Please note that these intervals are to show how the method works–not how the method works *best*.

```
1    /// Shell sort of data using specified swapping intervals.
2    public static void IntArrayShellSort (int[] data, int[] intervals)
3    {
4        int N = data.Length;
5        // The intervals for the shell sort must be sorted, ascending
6        for (int k=intervals.Length-1; k>=0; k--) {
7            int interval = intervals[k];
8            for (int m=0; m<interval; m++) {
9                for (int j=m+interval; j<N; j+=interval) {
10                    for (int i=j; i>=interval && data[i]<data[i-interval];
11                        i-=interval) {
12                        Exchange (data, i, i - interval);
13                    }
14                }
15            }
```

```
16              }
17          }
```

```
1          public static void IntArrayShellSortNaive (int[] data)
2          {
3              int[] intervals = { 1, 2, 4, 8 };
4              IntArrayShellSort (data, intervals);
5          }
```

In general, shell sort with sequences of jump sizes that are powers of one another doesn't do as well as one where most jump sizes are not multiples of others, mixing up the data more. In addition, the number of intervals must be increased as the size of the array to be sorted increases, which explains why we allow an *arbitrary* array of intervals to be specified.

Without too much explanation, we show how you can choose the intervals differently in an *improved* shell sort, where the intervals have been chosen so as not to be multiples of one another.

Donald Knuth has suggested a couple of methods for computing the intervals:

$$h_0 = 1$$
$$h_{k+1} = 3h_k + 1$$
$$t = \lfloor log_3 n \rfloor - 1$$

Here we are using notation for the *floor* function $\lfloor x \rfloor$ means the largest integer $\leq x$.

This results in a sequence 1, 4, 13, 40, 121.... You stop computing values in the sequence when $t = log_3 n - 1$. (So for n=50,000, you should have about 9-10 intervals.)

For completeness, we note that $log_3 n$ must be sufficiently large (and > 2) for this method to work. Our code ensures this by taking the *maximum* of $log_3 n$ and 1.

Knuth also suggests:

$$h_0 = 1$$
$$h_{k+1} = 2h_k + 1$$
$$t = \lfloor log_2 n \rfloor - 1$$

This results in a sequence 1, 3, 7, 15, 31....

Here is the improvement to our naive method that dynamically calculates the intervals based on the first suggestion of Knuth:

```csharp
/// Generates the intervals for Shell sort on a
/// list of length n via an algorithm from Knuth.
static int[] GenerateIntervals (int n)
{
    if (n < 2) {  // no sorting will be needed
        return new int[0];
    }
    int t = Math.Max (1, (int)Math.Log (n, 3) - 1);
    int[] intervals = new int[t];
    intervals [0] = 1;
    for (int i=1; i < t; i++)
        intervals [i] = 3 * intervals [i - 1] + 1;
    return intervals;
}

public static void IntArrayShellSortBetter (int[] data)
{
    int[] intervals = GenerateIntervals (data.Length);
    IntArrayShellSort (data, intervals);
}
```

Shell sort is a complex sorting algorithm to make "work well", which is why it is not seen often in practice. It is, however, making a bit of a comeback in embedded systems.

We nevertheless think it is a very cool algorithm to have heard of as a computer science student and think it has promise in a number of situations, especially in systems where there are limits on available memory (e.g. embedded systems).

10.4.6 Quicksort a.k.a. Partition Sort

This sort is a more advanced example that uses *recursion*. We list it because it is one of the best sorts for *random* data, having an *average* time behavior of $O(N \log N)$. Quicksort is a rather interesting case. While it has an excellent average behavior, it has a worst case performance of $O(N^2)$.

```csharp
/// Sort elements of data in index range [lowI, highI].
public static void IntArrayQuickSort (int[] data,
```

```
3                                                   int lowI, int highI)
4          {
5              int afterSmall = lowI, beforeBig = highI;
6              int pivot = data[(lowI + highI) / 2];
7              // in loop data[i] <= pivot if i < afterSmall
8              //         data[i] >= pivot if i > beforeBig
9              //         region with aftersmall <= i <= beforeBig
10             //             shrinks to nothing.
11             while (afterSmall <= beforeBig) {
12                while (data[afterSmall] < pivot)
13                   afterSmall++;
14                while (pivot < data[beforeBig])
15                   beforeBig--;
16                if (afterSmall <= beforeBig) {
17                   Exchange (data, afterSmall, beforeBig);
18                   afterSmall++;
19                   beforeBig--;
20                }
21             } // after loop: beforeBig < afterSmall, and
22             //      data[i] <= pivot for i <= beforeBig,
23             //      data[i] == pivot for i if beforeBig < i < afterSmall,
24             //      data[i] >= pivot for i >= afterSmall.
25             if (lowI < beforeBig) // at least two elements
26                IntArrayQuickSort (data, lowI, beforeBig);
27             if (afterSmall < highI) // at least two elements
28                IntArrayQuickSort (data, afterSmall, highI);
29          }
30
31          public static void IntArrayQuickSort (int[] data)
32          {
33             if (data.Length > 1)
34                IntArrayQuickSort (data, 0, data.Length - 1);
35          }
```

Though the initial call is to sort an entire array, this is accomplished by dealing with sections, so the main work is done in the version with the two extra parameters, giving the lowest and highest index considered.

It picks an arbitrary element as *pivot*, and then swaps elements with values above and below the pivot until the part of the array being processed is in three sections:

1. elements <= pivot;

2. possibly an element equal to pivot

3. elements >= pivot.

Though sections 1 and 3 are not sorted, there are no inversions *between* any two separate sections, so only the smaller sections 1 and 3 need to be sorted *separately*, and only then if they have at least two elements. They can be sorted by calling the *same* function, but with a smaller range of indices to deal with in each case. These *recursive* calls stop when a part is reduced to one element.

Another optional glimpse at an advanced topic: The outer `while` loop in lines 11-20 has fairly complicated logic. To prove it is correct overall, you can state and prove the simpler *loop invariant* expressed in the comments above the loop, lines 7-10. This allows the conclusion after the loop in comment lines 21-24.

10.4.7 Random Data Generation

Now it is time to talk about how we are going to check the performance in a real-world situation. We're going to start by modeling the situation when the data are in random order.

The following code generates a random array:

```
1      /// Fill data with pseudo-random data seeded by randomSeed.
2      public static void IntArrayGenerate (int[] data, int randomSeed)
3      {
4          Random r = new Random (randomSeed);
5          for (int i=0; i < data.Length; i++)
6              data [i] = r.Next ();
7      }
```

There are a few things to note in this code:

1. We use the random number generator option to include a *seed*. Random numbers aren't truly random. The particular sequence is just determined by a seed. The simplest way to create a Random object uses a seed taken from the system clock.

2. In order to regenerate a particular example, we actually need the random sequence to be consistent, so we know that each of the sorting algorithms is being tested using the same random data.

3. Because the sorting algorithms *modify* the data that are passed to it, we need to have a way of regenerating the sequence for the next test. Hence we specify the same seed each time. (We could also copy the data, but it is kind of a waste of memory.)

This completes the discussion of the `Sorting` class in sorting/sorting.cs.

10.4.8 Timing

Separate from the basic sorts is the idea of checking their performance. The rest of the code is in the driver class in sorting/sorting_demo.cs.

We need the ability to time the various sorting algorithms. Luckily, the .Net framework/library gives us a way of doing so through its `Stopwatch` class. This class supports methods that you would expect if you've ever used a stopwatch (the kind found in sports):

- Reset: Resets the elapsed time to zero. We need this so we can use the same Stopwatch for each sorting algorithm.

- Start: Starts the stopwatch. Will keep recording time until stopped.

- Stop: Stops the stopwatch.

- ElapsedMilliseconds: Not really a method but a property (like a variable). We'll use this to get the total time that has elapsed between pairs of Start/Stop events in milliseconds.

So let's take a look at the `Main()` method's code to see how we compare the sorting algorithms. The `Main()` method should be thought of as an *experiment* that tests the performance of each of the sorting algorithms. As all of the tests follow the same pattern, we're going to look at the basic variable setup first, and then one test.

```
int arraySize;
int randomSeed;
Stopwatch watch = new Stopwatch ();
int[] data;
```

The variables declared here are to set up the apparatus:

- `arraySize`: The size of the array where we wish to test the performance. We will use this to create an array with `arraySize` random values.

- randomSeed: This allows the user to vary the seed that is used to create the random array. We often want to do this to determine whether our performance results are stable when run a large number of times with different distributions. We won't go into too much detail here but consider it an important part of building good performance benchmarks.

- watch: The stopwatch object we're using to do the timings of all experiments.

- data: The array to be sorted.

```
if (args.Length < 2) {
    arraySize = UI.PromptInt("Please enter desired array size: ");
    randomSeed = UI.PromptInt(
                    "Please enter an initial random seed value: ");
} else {
    arraySize = int.Parse (args [0]);
    randomSeed = int.Parse (args [1]);
}
data = new int[arraySize];
```

This code is designed so we can accept the parameters arraySize and randomSeed from the command line or by prompting the user. When programmers design benchmarks, they often try to make it possible to run them with minimal user interaction. For the purposes of teaching, we wanted to make it possible to run it with or without command-line parameters.

```
TimeSetup(data, randomSeed, watch);
Sorting.IntArrayQuickSort(data); //this line varies by experiment
TimeResult("Quick Sort", watch);
```

This code fragment is one of the tests in Main. All of the tests set up for timing first, run the desired sorting algorithm (bubble sort in this case), and report the timing results. Since the timing setup and reporting is always done the same way, they actions are placed in helping methods:

```
/// Set up data and watch for sort timing
public static void TimeSetup (int[] data, int randomSeed,
                              Stopwatch watch)
{
    Sorting.IntArrayGenerate (data, randomSeed);
    watch.Reset();
    watch.Start();
```

```
      }

      /// Report sort timing results from watch
      public static void TimeResult (string sortType, Stopwatch watch)
      {
         watch.Stop ();
         double elapsedTime = watch.ElapsedMilliseconds/1000.0;
         Console.WriteLine (sortType + ": {0:F3}", elapsedTime);
      }
```

`TimeSetup`:

1. Creates the 'random' array of `data`, based on `randomSeed`.

2. Resets the `Stopwatch` object to zero.

3. Starts the `Stopwatch`.

After the sort, `TimeResult`:

1. Stops the `Stopwatch` and get the elapsed time (`watch.ElapsedMilliseconds`). The value is an integer (long) number of milliseconds (thousandths of a second).

2. Prints the performance results in seconds.

You can see all of the tests in sorting/sorting_demo.cs.

10.4.9 Running the Code

Here's the output of a trial run on one of our computers. The results will vary depending on your computer's CPU, among other factors.

```
bin/Debug$ mono Sorting.exe 1000 12
Quick Sort: 0.000
Naive Shell Sort: 0.000
Better Shell Sort: 0.000
Insertion Sort: 0.001
Selection Sort: 0.002
Bubble Sort: 0.003
bin/Debug$ mono Sorting.exe 1000 55
Quick Sort: 0.000
Naive Shell Sort: 0.000
```

```
Better Shell Sort: 0.000
Insertion Sort: 0.001
Selection Sort: 0.002
Bubble Sort: 0.003
bin/Debug$ mono Sorting.exe 10000 2
Quick Sort: 0.001
Naive Shell Sort: 0.019
Better Shell Sort: 0.002
Insertion Sort: 0.134
Selection Sort: 0.174
Bubble Sort: 0.321
bin/Debug$ mono Sorting.exe 50000 2
Quick Sort: 0.006
Naive Shell Sort: 0.441
Better Shell Sort: 0.015
Insertion Sort: 3.239
Selection Sort: 4.172
Bubble Sort: 8.028
bin/Debug$ mono Sorting.exe 100000 2
Quick Sort: 0.014
Naive Shell Sort: 1.794
Better Shell Sort: 0.034
Insertion Sort: 13.158
Selection Sort: 16.736
Bubble Sort: 31.334
```

At least based on randomly-generated arrays, the performance can be summarized as follows:

- Bubble Sort is rather unimpressive as expected. In fact, this algorithm is never used in practice but is of historical interest. Like the brute-force style of searching, it does way too much work to come up with the right answer!

- Selection Sort and Insertion Sort are also rather unimpressive on their own. Even though Selection Sort can in theory do a lot less data movement, it must make a large number of comparisons to find the minimum value to be moved. Again it is way too much work. Insertion Sort, while unimpressive, fares a bit better and turns out to be a nice building block (if modified) for the Shell Sort. Varying the interval size drastically reduces the amount of data movement (and the distance it has to move).

- Shell Sort does rather well, especially when we pick the right intervals. In

practice, the intervals also need to be adjusted based on the size of the array, which is what we do as larger array sizes are considered. This is no trivial task but a great deal of work has already been done in the past to determine functions that generate good intervals.

- The Quicksort is generally fastest on random data. It is by far the most commonly used sorting algorithm. Yet there are signs that Shell sort is making a comeback in embedded systems, because it concise to code and is still quite fast. See *[WikipediaShellSort]*, where it is mentioned that the *[uClibc]* library makes use of Shell sort in its `qsort()` implementation, rather than implementing the library sort with the more common quicksort.

10.5 Binary Searching

Binary search is an improvement over linear searching that works only if the data in the array are sorted beforehand.

Suppose we have the following array data shown under the array indices:

```
10   20   30   40   50   60   70   80   90   100 115 125 135 145 155 1 78 198
```

Binary search works by keeping track of the midpoint (mid) and the minimum (min) and maximum (max) index positions where the item *might be*.

If we are looking for a number, say, 115, here is a visual on how we might go about it. We display the indices over the data being considered. Here min and max are the smallest and largest index to still consider. A textual explanation follows the visual:

```
   0    1    2    3    4    5    6    7    8    9   10   11   12   13   14 | 15   16
  10   20   30   40   50   60   70   80   90  100  115  125  135  145  155 |178  198
min=0 max=16 mid=8
                                             100  115  125  135  145  155 |178  198
min=9 max=16 mid=12
                                             100  115  125
min=9 max=11 mid=10
Item 115 found at position 10
```

- We start by testing the data at position 8. 115 is greater than the value at position 8 (100), so we assume that the value must be somewhere between positions 9 and 16.

- In the second pass, we test the data at position 12 (the midpoint between 9 and 16). 115 is less than the value at position 12, so we assume that the value must be somewhere between positions 9 and 11.

- In the last pass, we test the value at position 10. The value 115 is at this position, so we're done.

So binary search (as its name might suggest) works by dividing the interval to be searched during each pass in half. If you think about how it's working here with 17 items. Because there is integer division here, the interval will not always be precisely half. it is the floor of dividing by 2 (integer division, that is).

With the data above, you see that the algorithm determined the item within 3 steps. To reduce to one element to consider, it could be 4 or 5 steps. Note that $4 < log_2 17 < 5$.

Now that we've seen how the method works, here is the code in binary_searching/binary_searching.cs that does the work:

```csharp
/// Return the index of item in a non-empty sorted array data,
/// or return -1 if item is not in the array.
public static int IntArrayBinarySearch(int[] data, int item)
{
    int min = 0, max = data.Length-1;
    while(min <= max) {
        int mid = (min+max) / 2;
        if (data[mid] == item)
            return mid;
        if (item > data[mid])
            min = mid + 1;
        else
            max = mid - 1;
    }
    return -1;
}
```

Here's a quick explanation, because it largely follows from the above explanation.

- Line 5: Initially item could be anywhere in the array, so minimum is at position 0 and maximum is at position N-1 (data.Length - 1).

- The loop to make repeated passes over the array begins on line 6. We can only continue searching if there is some data left to consider (`min <= max`).

10.5. Binary Searching

- Line 7 does just what we expect: It calculates the median position (mid).

- It is always possible that we've found the item, which is what we test on line 8, and return with our answer if we found it.

- Lines 10-13: If not, we continue. If the item is greater than the value at this mid position, we know it is in the "upper half". Otherwise, it's in the "lower half".

- Line 15: Otherwise the binary search loop terminates, and we return -1 (to indicate not found). The -1 value is a commonly-returned indicator of failure in search operations (especially on arrays, lists, and strings), so we use this mostly out of respect for tradition. It makes particular sense, because -1 is not within the *index set* of the array (which starts at 0 in C# and ends at `data.Length - 1`.

Of course we generally would be searching in an array with multiple elements. It is still important to check *edge cases*: Check that the code correctly returns -1 if the array has length 0 (a legal length).

Similar to linear searching, we provide a main program that tests it out. in binary_searching/binary_searching_demo.cs. It uses an elaboration of binary search that prints out the steps visually, as in the introduction to this section. It also references previous example projects: functions from files searching/extract_from_string.cs and sorting/sorting.cs.

```csharp
string input = UI.PromptLine(
    "Please enter some comma/space separated integers: ");
int[] data = ExtractFromString.IntsFromString(input);
Sorting.IntArrayShellSortBetter(data);
string prompt =
    "Please enter a number to find (empty line to end): ";
input = UI.PromptLine(prompt);
while (input.Length != 0) {
    int searchItem = int.Parse(input);
    int foundPos = BinarySearching.IntArrayBinarySearchPrinted(
                        data, searchItem);
    if (foundPos < 0)
        Console.WriteLine("Item {0} not found", searchItem);
    else
        Console.WriteLine("Item {0} found at position {1}",
                            searchItem, foundPos);
    input = UI.PromptLine(prompt);
```

```
 18        }
```

10.5.1 Elaborated Binary Search Exercise

Even if you do not find `item` in a binary search, it is sometimes useful to know
where `item` lies in relation to the array elements. It could be before the first
element, in between two elements, or after the last element. Suppose `N` is the
(positive) array length. Instead of just returning -1 if `item` is not in the array,
return

- -1 if `item < data[0]`

- `-(k+1)` if `data[k-1] < item < data[k]`

- `-(N+1)` if `data[N-1] < item`

Modify binary_searching/binary_searching.cs into `binary_searching2.cs`
so this extra information is returned (and indicated clearly in a main testing pro-
gram derived from binary_searching/binary_searching_demo.cs). This should *not*
require a change to the `while` loop, *nor* require any added loop.

10.6 Lab: Arrays

10.6.1 Overview

In this lab, we'll practice working with arrays. Arrays are fundamental to com-
puter science, especially when it comes to formulating various *algorithms*. We've
already learned a bit about arrays through the `string` data type. In many ways,
a character string reveals the secrets of arrays:

- each element of a string is a common type (char)

- we can use indexing to find any given character, e.g. `s[i]` gives us the
  character at position `i`.

- we know that the string has a finite length, e.g. `s.Length`.

So you've already learned these *concepts*. But practice is useful creating and
manipulating arrays with different kinds of data.

10.6.2 Goals

In this lab, we're going to practice:

- creating arrays that hold numerical data

- populating an array with data

- using the tools of loops and decisions to do something interesting with the data

- printing the data

10.6.3 Tasks

Copy the example file array_lab_stub/array_lab.cs to a new project of yours. Complete the body of a function for each main part, and call each function in `Main` several times with actual parameters chosen to test it well. To label your illustrations, make liberal use of the first function, `PrintNums`, to display and label inputs and outputs. Where several tests are appropriate for the same function, you might want to write a separate testing function that prints and labels inputs, passes the data on to the function being tested, and prints results.

Recall that you can declare an array in two ways:

```
int[] myArray1 = new int[10];
int[] myArray2 = { 7, 7, 3, 5, 5, 5, 1, 2, 1, 2 };
```

The first declaration creates an array initialized to all zeroes. The second creates an array where the elements are taken from the bracketed list of values. The second will be convenient to set up tests for this lab.

1. Complete and test the function with documentation and heading:

```
/// Print label and then each element preceeded by a space,
///  all across one line.  Example:
///  If a contains {3, -1, 5} and label is "Nums:",
///  print:   Nums: 3 -1 5
static void PrintInts(string label, int[] a)
{

}
```

This will be handy for labeling later tests. Note that you end on the same line, but a later label can start with \n to advance to the next line.

2. Complete and test the function with documentation and heading:

```
///  Prompt the user to enter n integers, and
///  return an array containing them.
///  Example:  ReadInts("Enter values", 3) could generate the
///  Console sequence:
///      Enter values (3)
///      1: 5
///      2: 7
///      3: -1
///  and the function would return an array containing {5, 7, -1}.
///  Note the format of the prompts for individual elements.
static int[] ReadInts(string prompt, int n)
{
    return new int[0]; // so stub compiles
}
```

This will allow user tests. The earlier input utility functions are included at the end of the class.

3. Complete and test the function with documentation and heading:

```
///  Return the minimum value in a.
///  Example: If a contains {5, 7, 4, 9}, return 4.
///  Assume a contains at least one value.
static int Minimum(int[] a)
{
    return 0; // so stub compiles
}
```

4. Complete and test the function with documentation and heading:

```
///  Return the number of even values in a.
///  Example: If a contains {-4, 7, 6, 12, 9}, return 3.
static int CountEven(int[] a)
{
    return 0; // so stub compiles
}
```

5. Complete and test the function with documentation and heading:

```
      /// Add corresponding elements of a and b and place them in sum.
      /// Assume all arrays have the same Length.
      /// Example: If a contains {2, 4, 6} and b contains {7, -1, 8}
      /// then at the end sum should contain {9, 3, 14}.
      static void PairwiseAdd(int[] a, int[] b, int[] sum)
      {


      }
```

To test this out, you'll need to declare and initialize the arrays to be added. You'll *also* need to declare a third array to hold the results. Make sure that the arrays all have the same dimensionality before proceeding.

This section is a warm-up for the next one. It is not required if you do the next one:

6. Complete and test the function with documentation and heading:

```
      /// Return a new array whose elements are the sums of the
      /// corresponding elements of a and b.
      /// Assume a and b have the same Length.
      /// Example: If a contains {2, 4, 6} and b contains {3, -1, 5}
      /// then return an array containing {5, 3, 11}.
      static int[] NewPairwiseAdd(int[] a, int[] b)
      {
         return new int[0]; // so stub compiles
      }
```

See how this is different from the previous part!

7. Complete and test the function with documentation and heading:

```
      /// Return true if the numbers are sorted in increasing order,
      /// so that in each pair of consecutive entries,
      /// the second is always at least as large as the first.
      /// Return false otherwise.  Assume an array with fewer than
      /// two elements is ascending.
      /// Examples: If a contains {2, 5, 5, 8}, return true;
      /// if a contains {2, 5, 3, 8}, return false.
      static bool IsAscending(int[] a)
      {
         return false; // so stub compiles
      }
```

This has some pitfalls. You will need more tests that the ones in the documentation! You can code this with a "short-circuit" loop. What do you need to find to be immediately sure you know the answer?

8. **20 % extra credit:** Complete and test the function with documentation and heading:

```
///  Print an ascending sequence from the elements
///  of a, starting with the first element and printing
///  the next number after the previous number
///  that is at least as large as the previous one printed.
///  Example: If a contains {5, 2, 8, 4, 8, 11, 6, 7, 10},
///  print:  5 8 8 11
static void PrintAscendingValues(int[] a)
{

}
```

9. **20 % extra credit:** Complete and test the function with documentation and heading:

```
///  Prints each ascending run in a, one run per line.
///  Example: If a contains {2, 5, 8, 3, 9, 9, 8}, print
///  2 5 8
///  3 9 9
///  8
static void PrintRuns(int[] a)
{

}
```

10. **20 % extra credit:** Given two arrays, a and b that represent vectors. Write a function that computes the vector dot product of these two floating point arrays. The vector dot product (in mathematics) is defined as the sum of a[i] * b[i] (for all i). Here's an example of how it should work:

```
double[] a = new double[] { 1.5, 2.0, 3.0 };
double[] b = new double[] { 4.0, 2.0, -1.0 };

double dotProduct = VectorDotProduct(a, b);
Console.WriteLine("The dot product is {0}", dotProduct);

// Should calculate 1.5 * 4.0 + 2.0 * 2.0 + 3.0 * -1.0 = 7.0
```

From here on, create your own headings.

11. **20 % extra credit:** Suppose we have loaded an array with the digits of an integer, where the digit for the highest power of 10 is kept in position 0, next highest in position 1, and so on. The ones position is always at position array.Length - 1:

```
int[] digits = { 1, 9, 6, 7 };
```

representing $1(10^3) + 9(10^2) + 6(10^1) + 7(10^0)$.

Without showing you the code, here is how you would convert a number from its digits to an integer efficiently, without calculating high powers for 10 separately:

```
num = 0
num = 10 * 0 + 1 = 1
num = 10 * 1 + 9 = 19
num = 10 * 19 + 6 = 196
num = 10 * 196 + 7 = 1967
done!
```

Write a function that converts the array of digits representing a base 10 number to its int value (or for really long integers, you are encouraged to use a long data type). Note that we only allow single digit numbers to be placed in the array, so negative numbers are not addressed.

12. **20 % extra credit:** Each digit represents a multiple of a *power* of the *base*. In the previous version the base is 10, but other bases are important. Now make the base a parameter. Here we consider bases no bigger than 10, so we can continue to use only digits for place value symbols. Write a function (or revise the previous solution) to return the int or long represented. For example if $\{1, 0, 0, 1, 1\}$ represents a base 2 number, $1(2^4) + 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = 19$ is returned. Base 2 is central to computer hardware.

10.7 Lab: Performance

In *Sorting Algorithms* we took advantage of a few ideas to show how to do basic benchmarking to compare the various approaches.

- using randomly-generated data

- making sure each algorithm is working with the same data

- making sure that we try a range of sizes to observe the effects of scaling

- using a timer with sufficiently high resolution (the `Stopwatch` gives us measurements in milliseconds).

In this lab, you get your chance to learn a bit more about performance by comparing *searches*. The art of benchmarking is something that is easy to learn but takes a lifetime to master (to borrow a phrase from the famous Othello board game).

Most of the algorithms we cover in introductory courses tend to be *polynomial* in nature. That is, the execution time can be bounded by a polynomial function of the data size n. A more accurate measure may also include a logarithm. Examples include but are not limited to:

- $O(1)$ is constant time, characterized by a calculation with a limited number of steps.

- $O(n)$ is linear time; often characterized by a single loop

- $O(n^2)$ is the time squared; often characterized by a nested loop

- $O(log\ n)$ is logarithmic (base 2) time; often characterized by a loop that repeatedly divides its work in half. The binary search is a well-known example.

- $O(n\ log\ n)$ is an example of a hybrid. Perhaps there is an outer loop that is linear and an inner loop that is logarithmic.

And there are way more than these shown here. As you progress in computing, you'll come to know and appreciate these in greater detail.

In this lab, we're going to look at a few different data structures and methods that perform searches on them and do *empirical* analysis to get an idea of how well each combination works. Contrasted with other labs where you had to write a lot of code, we're going to give you some code to do all of the needed work but ask you to write the code to do the actual analysis and produce a basic table.

10.7. Lab: Performance **335**

10.7.1 The Experiments

We're going to measure the performance of data structures we have been learning about (and *will* learn about, for lists and sets). For this lab, we'll focus on:

- Integer arrays using *Linear Searching* and *Binary Searching*

- *Lists* of integers with linear searching

- *Sets* of integers; checking if an item is contained in the set

In the interest of fairness, we are only going to look at the time it takes to perform the various search operations. We're not going to count the time to randomly-generate the data and actually *build* the data structure. The reasoning is straight-forward. We're interested in the search time, which is completely independent of other aspects that may be at play. We're not at all saying that the other aspects are unimportant but want to keep the assignment focus on search.

The experimental apparatus that we are constructing will do the following for each of the cases:

- create the data structure (e.g. new array, new list, new set)

- use a random seed `seed`, initialize a random generator that will generate `n` values.

- insert the random values into the data structure . For the case of sets, which eliminate duplicates, it is entirely possible you will end up with a tiny fraction of a percent fewer than `n` values.

- to measure the performance of any given search method, we need to perform a significant number of lookups (based on numbers in the random sequence) to ensure that we get an accurate idea of the *average* lookup time in practice. We'll call this parameter, `rep`. We will spread out the values looked for by checking data elements that have indices at a regular interval throughout the array. The separation is `m = n/rep` when `rep < n`. The separation is 1, and we wrap around at the end of the array if `rep > n`.

- We'll start a Stopwatch just before entering the loop to perform the lookups.

10.7.2 Starter Project

To make your life easier, we have put together a project that refers to all the code for all of the experiments you need to run. (That's right, we're giving you the code for the *experiments*, but you're going to write some code to run the various experiments and then run for varying sizes of n.) The stub file is performance_lab_stub/performance_lab.cs.

Recreate example project performance_lab_stub in your solution as performance_lab, so you have your own copy to modify. You can either

- copy into the lab project the files sorting/sorting.cs, searching/searching.cs, and binary_searching/binary_searching.cs. *If* you copy them into the lab project, *rename* the unused Main method from binary_searching.cs to something else (since Xamarin Studio allows only one Main method in a project).

- An alternative is to recreate their whole projects, and *reference* them from the lab project.

Here is the code for the first experiment, to test the performance of linear searching on integer arrays:

```
1  public static long ExpIntArrayLinearSearch (int n, int rep, int seed)
2  {
3      Stopwatch watch = new Stopwatch ();
4      int[] data = new int[n];
5      Sorting.IntArrayGenerate (data, seed);
6      int m = Math.Max(1, n/rep);
7      watch.Reset ();
8      watch.Start ();
9      // perform the rep lookups
10     for (int k=0, i=0; k < rep; k++, i=(i+m)%n) {
11         Searching.IntArrayLinearSearch (data, data [i]);
12     }
13     watch.Stop ();
14     return watch.ElapsedMilliseconds;
15 }
```

Let's take a quick look at how this experiment is constructed. We'll also take a look at the other experiments but these will likely be presented in a bit less detail, except to highlight the differences:

- On line 3, we create a `Stopwatch` instance. We'll be using this to do the timing.

- On lines 4-5, we are creating the data to be searched. Because we have already written this code in our sorting algorithms examples, we can refer to the Sorting class code in `sorting.cs`, as long as you made the lab project able to reference it. We use the `Sorting` class name to access the method `IntArrayGenerate()` within this class. We also take advantage of this in the other experiments.

- Line 6 converts the number of repetitions into the increment in index values for each time.

- Line 7 resets the stopwatch. It is not technically required; however, we tend to be in the habit of doing it, because we sometimes reuse the same stopwatch and want to make sure it is completely zeroed out. A call to `Reset()` ensures it is zero.

- Line 8 actually starts the stopwatch. We are starting here as opposed to before line 4, because the random data generation has nothing to do with the actual searching of the array data structure.

- Lines 10 through 12 are searching `rep` times for an item already known to be in the array.

- Line 13 stops the stopwatch.

- Line 14 returns the elapsed time in *milliseconds* between the `Start()` and `Stop()` method calls, which reflects the actual time of the experiment.

Each of the other experiments is constructed similarly. For linear search and binary search we use the methods created earlier. For the lists and the set we use the built-in `Contains` method to search. The list and set are directly initialized in their constructors from the array data. (More on that in later chapters.)

You need to fill in the `Main` method. The stub already has code to generate a random value for the `seed` for any run of the program. *Read* through to the end of the lab before starting to code. A step-by-step sequence is suggested at the end.

- Your code must parse command line `args` for the parameters `rep` and *any number* of values for `n`. For instance:

  mono PerformanceLab.exe 50000 1000 10000 100000

would generate the table shown below for 50000 repetitions for each of the values of `n`: 1000, 10000, and 100000.

- In the end you will want to run each experiment for `rep` repetitions and iterate through each different value of `n`.

- Present the result data in a nice printed right-justified table for all values of n, with a title including the number of repetitions. Print something like the following, with the number of seconds calculated.

```
Times in seconds with 50000 repetitions
     n     linear       list  binary      set
  1000  ????.???   ????.???  ??.???   ??.???
 10000  ????.???   ????.???  ??.???   ??.???
100000  ????.???   ????.???  ??.???   ??.???
```

The table would be longer if more values of n were entered on the command line. Note that the experiments return times in milliseconds, (1/1000 of a second) while the table should print times in *seconds*.

- Your final aim is to show your TA or instructor the results of a run with a table with at least three lines of data and with n being successive powers of 10, and *non-zero entries everywhere*. *Read on* for the major catch!

  You will need to *experiment* and adjust the repetitions and `n` choices. In order to *get all perceptible values* (nonzero), you will need a very large number of repetitions to work for the fastest searches. Our choice of 50000 in the example is not appropriate with these n values. The catch is that without further tweaking, you will only get nonzero values for all the fastest searches if the slower ones take *ridiculously long*.

  Because the range of speeds is so enormous, make an accommodation with the slow linear versions: If `rep >= 100` and `(long)n*rep >= 100000000`, then, for the linear and list columns *only*, time with `rep2 = rep/100` instead of `rep`, and then compensate by multiplying the resulting time by `(double)rep/rep2` to produce the final table value. (This multiplier is not necessarily just 100, since the integer division creating `rep2` may not be exact.)

Before making the modification for large numbers, be sure to test with small enough values (though some results will be 0). Once again, you are encouraged to develop this is steps, for example:

1. Make sure you can parse the command line parameters. In a testing version write code to print out `rep`, and *separate* code to print out all `n` values, for any number of `n` values.

2. Print out one *linear* test for `rep` and one value of `n`.

3. Print out the results for all tests for `rep` and one value of `n`. Keep `rep*n` small enough so the linear searches do not take too much time.

4. Do all values of `n`.

5. Make the printing be formatted as in the sample table.

6. Add the modification for large `rep*n`.

7. Experiment and get a table to show off!

10.8 Multi-dimensional Arrays

10.8.1 Rectangular Arrays (Two Dimensional)

You should be familiar with one dimensional arrays. The data in arrays may be any type. While a one dimensional array works for a sequence of data, we need something more for a two dimensional table, where data values vary over both row and column.

If we have a table of integers, for instance with three rows and four columns:

```
2 4   7 55
3 1   8 10
6 0 49 12
```

We could declare an array variable of the right size as

```
int[,] table = new int[3, 4];
```

Multiple indices are separated by commas inside the square brackets. In declaring an array type, no indices are included so the `[,]` indicates a two dimensional array. Where the `new` object is being created, the values inside the square brackets give each dimension.

In general the notation for a two dimensional array declaration is:

> **type** [,] **variableName**

and to create a new array with default values:

> new **type** [*intExpression1* , *intExpression2*]

where the expressions evaluate to integers for the dimensions.

To assign the 8 in the table above, consider that it is in the second row in normal counting, but we start array indices at 0, so there are rows 0, 1, and 2, and the 8 has row index 1. Again starting with index 0 for columns, the 8 is at index 2. We can assign a value to that position with

```
table[1, 2] = 8;
```

In fact a two dimensional array just needs two indices that could mean anything. For instance, it was just the standard convention, calling the left index the row. They could have been switched everywhere, and assume the notation `table[column, row]`:

```
int[,] table = new int[4, 3];
table[2, 1] = 8;
```

but we will *stick with* the original `[row, column]` model.

Data indexed by more than two integer indices can be stored in a higher dimension array, with more indices between the square braces. We will only consider two dimensional arrays in the examples here.

A shorthand for initializing all the data in the table, analogous to initializing one dimensional arrays is:

```
int[, ] table = { {2, 4,   7, 55},
                  {3, 1,   8, 10},
                  {6, 0, 49, 12} };
```

All rows must be the same length when using this notation.

Often two dimensional arrays, like one dimensional arrays, are processed in loops. Multiple dimension arrays are often processed in nested loops. We could print out this table using columns 5 spaces wide with the code:

```
for (int r = 0; r < 3; r++) {
    for (int c = 0; c < 4; c++) {
        Console.Write("{0, 5}", table[r, c]);
```

10.8. Multi-dimensional Arrays **341**

```
            }
            Console.WriteLine();
        }
```

If we wanted to make a more general function out of that code, we have a problem: the number of rows and columns were literal values that we knew. We need something more general. For one dimensional arrays we had the Length property, but now there are more than one lengths!

The following csharp sequence illustrates the syntax needed:

```
csharp> int[, ] table = { {2, 4,  7, 55},
      >                    {3, 1,  8, 10},
      >                    {6, 0, 49, 12} };
csharp> table.Length;
12
csharp> table.GetLength(0);
3
csharp> table.GetLength(1);
4
csharp> foreach (int n in table) {
      >    Console.WriteLine(n);
      > }
2
4
7
55
3
1
8
10
6
0
49
12
```

Note:

- The meaning for the Length property, is now the *total* number of elements, $(3)(4) = 12$.

- The separate method `GetLength` is needed to find the number of rows and columns. The entries in the list of array indices in the multi-dimensional

---

array notation are themselves indexed to provide the `GetLength` method parameter for each dimension. In this case index 0 gives the row length (left index of the array notation), and 1 gives the column length (right index of the array notation).

- The `foreach` statement behavior is consistent with the `Length` property, giving *all* the elements, row by row. More generally, the rightmost indices vary most rapidly as the `foreach` statement iterates through all elements.

Just replacing 3 and 4 by `table.GetLength(0)` and `table.GetLength(1)` in the table printing code would make it general.

A more elaborate table might include row and column sums:

```
  2    4    7   55 |   68
  3    1    8   10 |   22
  6    0   49   12 |   67
--------------------
 11    5   64   77 |  157
```

For example, the following function from example file print_table/print_table.cs, prints out a table of integers neatly, including row and column sums. It illustrates a number of things. It shows the interplay between one and two dimensional arrays, since the row and column sums are just one dimensional arrays.

Now that we are using arrays, we can easily look at the same collection of data repeatedly. It is possible to look at the data one time to just see its maximum width, and then go through again and print data using a column width that is just large enough for the longest numbers. When looking through the data for string lengths, the row and column structure is not important, so the code illustrates `foreach` loops to chug through all the data. We use the earlier trick in *Modular Multiplication Table*, creating a custom format string to make columns the right width.

The code refers once to the earlier `StringOfReps` in *Lab: Loops* for the row of dashes setting off the column sums:

```csharp
static void PrintTableAndSums(int[,] t)
{
    int[] rowSum = new int[t.GetLength(0)],
    colSum = new int[t.GetLength(1)];
    int sum = 0;
    for (int r = 0; r < t.GetLength(0); r++) {
```

10.8. Multi-dimensional Arrays **343**

```
            for (int c = 0; c < t.GetLength(1); c++) {
                rowSum[r] += t[r, c];
                colSum[c] += t[r, c];
            }
            sum += rowSum[r];
        }
        int width = (""+sum).Length;
        foreach (int n in t) {
            width = Math.Max(width, (""+n).Length);
        }
        foreach (int n in colSum) {
            width = Math.Max(width, (""+n).Length);
        }
        foreach (int n in rowSum) {
            width = Math.Max(width, (""+n).Length);
        }
        string format = "{0," + width + "} ";

        for (int r = 0; r < t.GetLength(0); r++) {
            for (int c = 0; c < t.GetLength(1); c++) {
                Console.Write(format, t[r, c]);
            }
            Console.WriteLine("| " + format, rowSum[r]);
        }
        string reps = StringOfReps("-",(width+1)*(t.GetLength(1)+1) + 1);
        Console.WriteLine(reps);
        for (int c = 0; c < t.GetLength(1); c++) {
            Console.Write(format, colSum[c]);
        }
        Console.WriteLine("| " + format, sum);
    }
```

Row and Column Numbering Exercise

Write a function that sets the values in a given rectangular array to 10 * (row index +1) + column index + 1, with the normal row and column indices, starting from 0. For example an array with two rows and five columns would end up with values below. Your method should set the values in the array, not print them out:

```
11 12 13 14 15
21 22 23 24 25
```

If there are no more than 9 rows or columns, this display gives row and column numbers neatly for the normal human counting system, starting from 1.

Varying Column Width Exercise

Copy the project file print_table/print_table.cs to a file print_varying_width_table.cs in a projet of yours. Edit it so that *each* column is only as wide as it needs to be: the width for the widest entry in that column. The earlier data would now print as:

```
 2  4   7 55 |  68
 3  1   8 10 |  22
 6  0  49 12 |  67
---------------
11  5 64 77 | 157
```

Calculate each of these widths *only once*. Hint: Create an array of widths and an array of format strings.

10.8.2 Advanced topic: Array of Arrays

Since you can have an array of any type, it is also possible to have an *array of arrays*.

You could create a table with shape like the initial example in *Multi-dimensional Arrays*

```
int[][] table2 = new array[3][4];
```

and refer to an element still by row and column index::

```
table2[1][2] = 4;
```

The notational difference so far is just that each index in enclosed in separate square brackets, with no commas.

10.8. Multi-dimensional Arrays   **345**

This takes somewhat more memory and is longer to access than the multidimensional arrays that we have been talking about with the comma-separated notation. For most uses, when you want to refer to a rectangular table of data, like above, this new version has little to offer.

There are a couple of reasons why you might want this format.

First you may have functions that operate on one dimensional arrays, and individual rows of `table2` are one dimensional arrays: `table2[1]` has type `int[]`. On the other hand, with the earlier `table` with type `int[,]`, references to part of the array must always include a comma, so `tabel[1]` would not refer to a row, but would be a syntax error.

Also in an array of arrays, since each row is an independent array, their lengths can *vary*. Here is code to make a doubly indexed "triangular" collection (of 0 values). Each row must be separately created as a new array:

```
int[][] tri = new int[4][]; //create four null int[] elements
for (int i = 0; i < tri.Length; i++) { // Length 4 (rows)
   tri[i] = new int[i+1];   // each row has a different length
}
```

With `tri` constructed as above:

- `tri[3, 3]` would given a compiler error: no changing `[][]` to `[, ]`.

- `tri[3][3]` would refer to an element.

- `tri[1][3]` would given a run-time out-of-bounds error. since `tri[1]` is an array of length 2.

10.9 Chapter Review Questions

1. When do you want to use an array rather than just a bunch of individually named variables?

2. Before writing a program, must you know the exact size of an array that you are going to create?

3. Before creating a new array in a program, must the program be able to calculate the proper size for the array?

4. After you have created the array, can you change the size of the original array object?

5. If I have the declaration

```
int[] vals = new int[5];
```

    (a) What is stored directly in the memory position for variable `vals`?

    (b) Does `vals[3]` then have a clear value? If so, what?

    (c) Can I later make `vals` refer to an array of a different size?

6. Comment on the comparison between these two snippets:

```
char[] a = {'n', 'o', 'w'};
a[0] = 'c';

string s = "now";
s[0] = 'c';
```

7. If I want to read or modify the first 100 elements of a 999 element array, would I use a `foreach` loop or a `for` loop? Explain.

8. If I want to modify all the elements of an array, would I use a `foreach` loop or a `for` loop? Explain.

9. If I want to read all the elements of an array, but not change the array, and I do not care about the exact position in the array of any member, would I use a `foreach` loop or a `for` loop?

10. Is this legal?

```
int[] a= {1, 2, 3, 4};
//...
a = new int[7];
```

11. The definition of a program's `Main` method may optionally include a parameter. What is the type? How is it used?

12. What is an alias? Why is understanding aliases important with arrays?

13. If I have a function declared

```
static void f(int num)
//...
```

and I call it from my `Main` function

```
int v = 7;
f(v);
Console.WriteLine(v);
```

Could `f` change the value of the variable `v`, so 1 is printed in `Main`? If so, write a one-line body for `f` that does it.

14. If I have a function declared

```
static void f(int[] nums)
//...
```

and I call it from my `Main` function

```
int[] v = {7, 8, 9};
f(v);
Console.WriteLine(v[0]);
```

Could `f` change the value of the variable `v[0]`, so 1 is printed in `Main`? If so, write a one-line body for `f` that does it.

15. What is printed by this snippet?

```
int[] a = {1, 2, 3};
int[] b = {4, 5, 6};
b[0] = 7;
a[1] = 8;
b[2] = 9;
Console.WriteLine("" + a[0] + a[1] + a[2]);
```

16. What is printed by this snippet? (Only the second line is changed.)

```
int[] a = {1, 2, 3};
int[] b = a;
b[0] = 7;
a[1] = 8;
b[2] = 9;
Console.WriteLine("" + a[0] + a[1] + a[2]);
```

17. If my only use for variable `temp` is to set up this call to `f`:

```
int[] temp = {1, 2, 3};
f(temp);
```

how could I rewrite it with an anonymous array?

18. After this line, what is the value of `a[2]`?

```
bool[] a = new bool[5];
```

19. This will cause a runtime error. Why?

```
string[] a = new string[5];
foreach(string s in a) {
    Console.WriteLine(s.Length);
}
```

20. If you get a data sequence from a `Random` object, is it really random?

21. Explain the significance of a *seed* for a `Random` object.

22. Suppose I create an object `table` of type `double[,]`, and I think of the first index as referring to a row and the second index as referring to a column.

    (a) Must each row be the same length?

    (b) Does each row have a type `double[]` ?

23. (Optional) Suppose I create an object `table` of type `double[][]`, and I think of the first index as referring to a row and the second index as referring to a column.

    (a) Must each each row be the same length?

    (b) Does each row have a type `double[]` ?

ELEVEN

LISTS

11.1 List Syntax

Arrays are fine if you know ahead of time how long your sequence of items is. Then you create your array with that length, and you are all set.

If you want a variable sized container, you are likely to want a `List`. As with arrays, you might want a collection of any particular type. Unfortunately, you cannot use the simple notation of arrays to specify the type of element in a `List`. Array syntax is *built into* the language. Lists are handled in the *library* of types provided by C# from the .Net framework. There are all sorts of situations where you might want a general idea to have a version for each of many kinds of objects. There is *not* a new syntax for *each* one.

11.1.1 Generics

Instead .Net 4.0 introduced one new form of syntax that can apply to all sorts of classes, *generics*.

The type for a list of strings is

```
List<string>
```

The type for an `int` list is

```
List<int>
```

In general the new generic syntax allows a type (or several, comma separated) in angle brackets after a class name. Lists are an example that depends on just one included type. We will see more shortly.

There is a namespace for the generics for collections, including List: System.Collections.Generic.

We will use several generic library classes, though we will not write the definitions of new generic classes ourselves.

11.1.2 List Constructors and Methods

We can play with some `List` methods in csharp. Note that csharp informally displays the value of a `List` with a list of elements inside braces. This is *not* a legal way to assign values to lists.

The blocks below are all from one csharp session, with our comments breaking up the sequence.

With the no-parameter constructor, the `List` is empty to start:

```
csharp> List<string> words = new List<string>();
csharp> words;
{   }
csharp> words.Count;
0
```

You can add elements, and keep count with the `Count` property as the size changes:

```
csharp> words.Add("up");
csharp> words;
{ "up" }
csharp> words.Add("down");
csharp> words;
{ "up", "down" }
csharp> words.Add("over");
csharp> words;
{ "up", "down", "over" }
csharp> words.Count;
3
```

You can reference and change elements by index, like with arrays:

```
csharp> words[0];
"up"
csharp> words[2];
"over"
csharp> words[2] = "in";
csharp> words;
{ "up", "down", "in" }
```

You can use `foreach` like with arrays or other sequences:

```
csharp> foreach (string s in words) {
        >     Console.WriteLine(s.ToUpper());
        > }
UP
DOWN
ON
```

Note: Unfortunately C# is not user-friendly if you try to use `Console.WriteLine` to print a `List` *object*:

```
csharp> Console.WriteLine(words)
System.Collections.Generic.List`1[System.Int32]
```

Next compare `Remove`, which finds the first matching element and removes it, and `RemoveAt`, which removes the element at a specified index. `Remove` returns whether the List has been changed:

```
csharp> words.Remove("down");
true
csharp> words;
{ "up", "in" }
csharp> words.Remove("around"); // no change
false
csharp> words.Add("out");
csharp> words.Add("on");
csharp> words;
{ "up", "in", "out", "on" }
csharp> words.RemoveAt(2); // "out" is at index 2
csharp> words;
{ "up", "in", "on" }
```

Removing does not leave a "hole" in the `List`: The list closes up, so the index decreases for the elements after the removed one:

```
csharp> words[2];
"on"
csharp> words.Count;
3
```

You can check for membership in a `List` with `Contains`:

```
csharp> words.Contains("in");
true
csharp> words.Contains("into");
false
```

You can also remove all elements at once:

```
csharp> words.Clear();
csharp> words.Count;
0
```

Here is a List containing `int` elements. Though more verbose than for an array, you can initialize a `List` with another collection, including an anonymous array, specified with an explicit sequence in braces:

```
csharp> List<int> nums = new List<int>(new[]{5, 3, 7, 4});
csharp> nums;
{ 5, 3, 7, 4 }
```

We have been using the explicit declaration syntax, but generic types tend to get long, so `var` is handy with them:

```
var stuff = new List<string>();
```

When initializing a generic object, you still need to remember both the angle braces around the type *and* the parentheses for the parameter list after that.

An aside on the `Remove` method: It both causes a side effect, changing the list, *and* it returns a value. If a function returns a value, we typically use the function call as an expression in a larger statement. This is not necessary, as described in *Not using Return Values*. In that section we discussed the *mistake* of not using return values. The `Remove` method illustrates that this is not always a mistake: If you just want the side effect, trying to remove an element, whether or not it is

in the list, then there is no need to check for the return value. This complete C#
statement is fine:

```
someList.Remove(element);
```

You should generally think carefully before *defining* a function that both has a
side effect and a return value. Most functions that return a value do not have a
side effect. If you see a function used in the normal way as an expression, it is
easy to forget that it was *also* producing some side effect.

11.1.3 Interactive List Example

Lists are handy when you do not know how much data there will be. A simple
example would be reading in lines from the user interactively:

```
/// Return a List of lines entered by the user in response
/// to the prompt.  Lines in the List will be nonempty, since an
/// empty line terminates the input.
List<string> ReadLines(string prompt)
{
   List<string> lines = new List<string>();
   Console.WriteLine(prompt);
   Console.WriteLine("An empty line terminates input.");
   string line = Console.ReadLine();
   while (line.Length > 0) {
      lines.Add(line);
      line = Console.ReadLine();
   }
   return lines;
}
```

11.2 .Net Library (API)

This book can only introduce so many classes and methods from the C# library.
You should browse the MSDN .Net Framework Class Library's online documen-
tation.

http://msdn.microsoft.com/en-us/library/gg145045.aspx

We mostly deal with classes in the namespaces System, System.IO, and System.Collections.Generic, and you can drill down to them.

One complication is that variations on these classes and methods are included for several Microsoft languages. Under the Syntax heading, make sure the **C# tab** is selected.

For example, you can click in the left column on System.Collections Namespaces, and then System.Collections.Generic, and then, for example, List(T). (In C# that is `List<T>`.)

The summary section separates constructors, properties, and methods. When you see one of these with promise, click on it to get the full details. For example, click on the first method in the Methods section, `Add`, or something new, like `IndexOf`, or `Reverse`, or `Sort`....

Classes also can be classified in several ways for browsing:

- Those you will want to be fairly familiar with pretty soon: string, List, Dictionary

- Those that might be useful, that you should be at least aware of.

- Those that may be useful eventually, but are not worth your time now.

You will also find methods in various categories as you browse:

- Methods that make sense and are useful right away

- Methods that take a little reading to absorb

- Features that we have yet to discuss

- Features that are well beyond what we have talked about - ask or wait or read a *lot*.

11.3 Chapter Review Questions

1. Distinguish the cases when you would want to use a list instead of an array, or the other way around.

2. What syntax is consistent between arrays and lists? What are comparable features, but with different syntax?

3. How is the type declaration for a generic type distinctive?

4. Here is one way to put five particular elements into a list:

```
var words = new List<string>();
string[] temp = {"a", "an", "the", "on", "of"};
foreach(string s in temp) {
    words.Add(s);
}
```

How can you do this all without a loop, and with only two statements? How about with a single statement, assuming you do not need `temp` again?

5. If we continue on from above, with the line:

```
var words2 = words;
```

Then what would be the difference in effect between these two possible next lines?

```
words.Clear()

words = new List<string>();
```

6. The constructors for collections like a List are all overloaded. What forms are allowed in general?

7. If you delete an element from the middle of a list, what happens to the spot where you removed the element?

DICTIONARIES

12.1 Dictionary Syntax

We have explored several ways of storing a collection of the same type of data:

- arrays: built-in syntax, unchanging size of the collection
- List: generic class type, allows the size of the collection to grow

Both approaches allow reference to data elements using a numerical index between square brackets, as in `words[i]`. The index provides an order for the elements, but there is no meaning to the index beyond the sequence order.

Often, we want to look up data based on a more meaningful key, as in a dictionary: given a word, you can look up the definition.

C# uses the type name `Dictionary`, but with greater generality than in nontechnical use. In a regular dictionary, you start with a word, and look up the definition. The generalization is to have some piece of data that leads you to (or *maps* to) another piece of data. The computer science jargon is that a *key* leads you to a *value*. In a normal dictionary, these are both likely to be strings, but in the C# generalization, the possible types of key and value are much more extensive. Hence the generic `Dictionary` type requires you to specify both a type for the key and a type for the value.

We can initialize an English-Spanish dictionary `e2sp` with

```
Dictionary<string, string> e2sp = new Dictionary<string, string>();
```

That is quite a mouthful! The C# `var` syntax is handy to shorten it:

```
var e2sp = new Dictionary<string, string>();
```

The general generic type syntax is

> Dictionary< **keyType, valueType** >

If you are counting the number of repetitions of words in a document, you are likely to want a `Dictionary` mapping each word to its number of repetitions so far:

```
var wordCount = new Dictionary<string, int>();
```

If your friends all have a personal list of phone numbers, you might look them up with a dictionary with a string name for the key and a `List` of personal phone number strings for the value. The type could be `Dictionary<string, List<string>>`. This example illustrates how one generic type can be built on another.

There is no restriction on the value type. There is one important technical restriction on the key type: it should be immutable. This has to do with the implementation referenced in *Dictionary Efficiency*.

Similar to an array or `List`, you can assign and reference elements of a `Dictionary`, using square bracket notation. The difference is that the reference is through a key, not a sequential index, as in:

```
e2sp["one"] = "uno";
e2sp["two"] = "dos";
```

Csharp displays dictionaries in its own special form, as a sequence of pairs {key, value}. Again, this is *special* to csharp. Here is a longer csharp sequence, broken up with our comments:

```
csharp> Dictionary<string, string> e2sp = new Dictionary<string, string>();
csharp> e2sp;
{}
csharp> e2sp["one"] = "uno";
csharp> e2sp["two"] = "dos";
csharp> e2sp["three"] = "tres";
csharp> e2sp.Count;
3
```

```
csharp> e2sp;
{{ "one", "uno" }, { "two", "dos" }, { "three", "tres" }}
csharp> Console.WriteLine("{0}, {1}, {2}...", e2sp["one"],
     > e2sp["two"], e2sp["three"]);
uno, dos, tres...
```

If you want to iterate through a whole `Dictionary`, you will want the syntax
below, with `foreach` and the property `Keys`:

```
csharp> foreach (string s in e2sp.Keys) {
     >     Console.WriteLine(s);
     > }
one
two
three
```

The documentation for `Dictionary` says that you cannot depend on the order
of processing with `foreach`, though the present implementation remembers the
order in which keys were added.

It is often useful to know if a key is already in a `Dictionary`: Note the method
`ContainsKey`:

```
csharp> e2sp.ContainsKey("seven");
false
csharp> e2sp.ContainsKey("three");
true
```

The method Remove takes a key as parameter. Like a `List` and other collections,
a `Dictionary` has a `Clear` method:

```
csharp> e2sp.Count;
3
csharp> e2sp.Remove("two");
true
csharp> e2sp.Count;
2
csharp> e2sp.Clear();
csharp> e2sp.Count;
0
```

12.2 Dictionary Efficiency

We could simulate the effect of a Dictionary pretty easily by keeping a List `keys` and a List `values`, in the same order. We could find the entry with a specified key with:

```
int i = keys.IndexOf(key);
return values[i];
```

Searching though a `List`, however, take time proportional to the length of the `List` in general, *linear order*. Through a clever implementation covered in data structures classes, a `Dictionary` uses a *hash table* to make the average lookup time of *constant order*. A hash table depends on the keys being immutable.

12.3 Dictionary Examples

12.3.1 Sets

In the next section we will have an example making central use of a dictionary. It will also make use of a set. The generic C# version is a `HashSet`, which models a mathematical set: a collection with no repetitions and no defined order. We use a `HashSet` for the words to be ignored. We use a `HashSet` rather than a `List` because the `Contains` method for a `List` has linear order, while the `Contains` method for a `HashSet` uses the same trick as in a `Dictionary` to be of constant order on average.

Here is a csharp session using the type `HashSet` of strings. The `Add` method, like the `Remove` method for Lists, returns true or false depending on whether the method changes the set:

```
csharp> var set = new HashSet<string>();
csharp> set;
{  }
csharp> set.Add("hi");
true
csharp> set;
{ "hi" }
csharp> set.Add("up");
```

```
true
csharp> set;
{ "hi", "up" }
csharp> set.Add("hi");  // already there
false
csharp> set;
{ "hi", "up" }
csharp> set.Contains("hi");
true
csharp> set.Contains("down");
false
csharp> var set2 = new HashSet<string>(new string[]{"a", "be", "see"});
csharp> set2;
{ "a", "be", "see" }
```

That lack of order for a `HashSet` means it cannot be indexed, but otherwise it has mostly the same methods and constructors that have been discussed for a `List`, including `Add` and `Contains` and a constructor that takes a collection as parameter.

12.3.2 Word Count Example

Counting the number of repetitions of words in a text provides a realistic example of using a `Dictionary`. With each word that you find, you want to associate a number of repetitions. A complete program is in the example file count_words/count_words.cs.

The central functions are excerpted below, and they also introduce some extra features from the .Net libraries.

This constructor pattern taking the elements of one collection and creating another collection, possibly of another type, is used twice: first to create a `HashSet` from an array, and later to create a `List` from a `HashSet`. The latter is needed so the `List` can be sorted in alphabetical order with its `Sort` method, used here for the first time. Our table contains the words in alphabetical order.

Also used for the first time are two string methods: the pretty clearly named `ToCharArray` and another variation on `Split`. An alternative to supplying a single character to split on, is to use a `char` array as parameter, and the string is split at an occurrence of any of the characters in the array. This allows a split on

---

all punctuation and special symbol characters, as well as a blank.

We separate the processing into two functions, one calculating the dictionary, and one printing a table. To reduce the amount of clutter in the `Dictionary`, the function `GetCounts` takes as a parameter a set of words to ignore.

```csharp
        /// Return a Dictionary of word:count pairs from parsing s,
        ///  excluding all strings in ignore.
        public static Dictionary<string, int> GetCounts(string s,
             HashSet<string> ignore)
        {
           char[] sep = "\n\t !@#$%^&*()_+{}|[]\\:\";<>?,./".ToCharArray();
           string[] words = s.ToLower().Split(sep);
           ignore.Add(""); //generated from consecutive splitting characters
           var wc = new Dictionary<string, int>();
           foreach (string w in words) {
              if (!ignore.Contains(w)) {
                 if (wc.ContainsKey(w)) { //increase count of word already see
                    wc[w]++;
                 }
                 else {                   // make a first entry
                    wc[w] = 1;
                 }
              }
           }
           return wc;
        }

        /// Print each word and its count, if the count is at least minCount.
        public static void PrintCounts(Dictionary<string, int> wc, int minCoun
        {
           List<string> words = new List<string>(wc.Keys);
           words.Sort();
           foreach (string w in words) {
              if (wc[w] >= minCount) {
                 Console.WriteLine("{0}: {1}", w, wc[w]);
              }
           }
        }
```

Look at the code carefully, and look at the whole program that analyses the Gettysburg Address.

12.4 Lab: File Data and Collections

12.4.1 Goals for this lab:

- Read a text file.

- Work with loops.

- Work with a Dictionary and a List.

- Retrieve a random entry.

Overview

Copy project dict_lab_stub to your own project.

This lab provides a replacement file fake_help.cs for an improved project. The project still needs some additions in a helper class.

Before we get there, open the comparison program fake_help_verbose/fake_help_verbose.cs and look at the methods `GetParagraphs()` and `GetDictionary()`. All the strings for the responses are pre-coded for you there, but if you were writing your own methods, it would be a pain. There is all the repetitious code to make multiline strings and then to add to the List and Dictionary. This lab will provide simple versatile methods to fill a `List<string>` or a `Dictionary<string, string>`: You only need you to write the string data itself into a text file, with the only overhead being a few extra newlines. Minor further adaptations could save time later in a project, too.

Look in your copy of fake_help.cs. It creates the `List guessList` and the `Dictionary responses` using more general functions that you need to fill in. The stubs for these new versions are put in the class `FileUtil` for easy reuse. `Main` calls these functions and chooses the files to read. The results will look the same as the original program to the user, but the second version will be easier for a programmer to read and generalize: It will be easier in other situations where you want lots of canned data in your program (like in a game you might write soon).

The stub should run as is (mostly saying things are not implemented). Test out your work at every stage!

You will need to complete very short versions of functions `GetParagraphs` and `GetDictionary` that have been moved to file_util.cs and now take a `StreamReader` as parameter. The files that they read will contain the basic data. You can look in the lab project at the first data file: help_not_defaults.txt, and the beginning is shown below:

```
Welcome to We-Give-Answers!
What do you have to say?

We-Give-Answers
thanks you for your patronage.
Call again if we can help you
with any other problem!

No other customer has ever complained
about this before.  What is your system
configuration?

That sounds odd. Could you describe
that problem in more detail?
```

You can see that it includes the data for the welcome and goodbye strings followed by all the data to go in the `List` of random answers.

One complication is that many of these strings take up several lines, in what we call a *paragraph*. We follow a standard convention for putting paragraphs into plain text: Put a blank line after a paragraph to mark its end. As you can see, that is how help_not_defaults.txt is set up.

12.4.2 Steps

All of the additions you need to make are in bodies of function definitions in the class `FileUtil`. Look back to `Main` in `FakeAdvise` to see how the functions from `FileUtil` are actually used: The `StreamReader` is set up to read from the right file. The the `FileUtil` functions `ReadParagraph`, `GetParagraphs`, and `GetDictionary` are used to provide the text data needed.

ReadParagraph

The first method to complete in file_util.cs is useful by itself and later for use in the `GetParagraphs` and `GetDictionary` that you will complete. See the stub:

```
/// Return a string consisting of a sequence of nonempty lines read
/// from reader. All the newlines at the ends of these lines are included.
/// The function ends after reading (but not including) an empty line.
public static string ReadParagraph(StreamReader reader)
```

The first call to `ReadParagraph`, using the file illustrated above, should return the following (showing the escape codes for the newlines):

```
"Welcome to We-Give-Answers!\nWhat do you have to say?\n"
```

and then the reader should be set to read the goodbye paragraph (the next time `ReadParagraph` is called).

To code, you can read lines one at a time, and append them to the part of the paragraph read so far. There is one thing to watch out for: The `ReadLine` function *throws away* the following newline (`"\n"`) in the input. You need to preserve it, so be sure to explicitly add a newline, back onto your paragraph string after each nonempty line is added. The returned paragraph should end with a single newline.

Throw away the empty line in the input after the paragraph. Make sure you stop after reading the empty line. It is very important that you advance the reader to the right place, to be ready to read the next paragraph.

Be careful of a pitfall with files: You can only read a given chunk once: If you read again, with the exact same syntax, you get the *next* line of the file. The `ReadLine` method has the *side effect* of advancing the reading position in the file.

Testing: This first short `ReadParagraph` function should actually be most of the code that you write for the lab! The program is set up so you can immediately run the program and test `ReadParagraph`: It is called to read in the welcome string and the goodbye string for the program, so if those come correctly to the screen, you can advance to the next two parts.

GetParagraphs

Since you have `ReadParagraph` at your disposal, you now only need to insert a *few remaining lines of code* to complete the next method `GetParagraphs`, that reads to the end of the file, and likely processes more than one paragraph.

```csharp
/// Read the remaining empty-line terminated paragraphs
/// from reader into a new list of paragraph strings,
/// and return the list.
/// The function reads all the way to the end of
/// the file attached to reader.
/// The file must end with two newlines in sequence: one at the
/// end of the last nonempty line followed by one for the empty line.
public static List<string> GetParagraphs(StreamReader reader)
{
    List<string> all = new List<string>();

    // REPLACE the next line with your lines of code to fill all
    all.Add("You have not coded GetParagraphs yet!\n");

    return all;
}
```

Look again at help_not_defaults.txt, to see how the data is set up.

This lab requires very few lines of code. Be sure to read the examples and instructions carefully (several times). A lot of ideas get packed into the few lines!

Testing: After writing `GetParagraphs`, the random responses in the lab project program should work as the user enters lines in the program.

GetDictionary

The last stub to complete in file_util.cs is `GetDictionary`. Its stub also takes a `StreamReader` as parameter. In `Main` this function is called to read from help_not_responses.txt. Here are the first few lines:

```
crash
Well, it never crashes on our system.
It must have something to do with your system.
Tell me more about your configuration.
```

```
slow
I think this has to do with your hardware.
Upgrading your processor should solve all
performance problems.
Have you got a problem with our software?

performance
Performance was quite adequate in all our tests.
Are you running any other processes in the background?
```

Here is the stub of the function to complete, reading such data:

```csharp
/// Return a new Dictionary, taking data for it from reader.
/// Reader contains key-value pairs, where each single-line key is
/// followed by a possibly multi-line paragraph value that is terminated
/// by an empty line. The file must end with two newlines in sequence:
/// one at the end of the last nonempty line followed by one for the
/// empty line.
public static Dictionary<string, string> GetDictionary(StreamReader reader)
{
    Dictionary<string, string> d = new Dictionary<string, string>();

    // add your lines of code to fill d here!

    return d;
}
```

Testing: When you complete this function, the program should behave just like the earlier verbose version with the hard-coded data, using a dictionary value if it finds the right key, or choosing a random response if there is no key match.

Be careful to distinguish the data file help_not_responses.txt from help_not_responses2.txt, used in the extra credit option.

This should also be an extremely short amount of coding! Think of following through the data file, and get the corresponding sequence of instructions to handle the data in the exact same sequence.

Show the program output to a TA (after the extra credit if you like).

Extra credit

1. (20%) Modify ReadParagragh so it *also* works if the paragraph ends at the end of the file, with no blank line after it, or if the line after the paragraph only has whitespace characters. Both changes are good to bullet-proof the code, since the added or removed whitespace is hard to see in print.

2. (20%) The crude word classification scheme would recognize "crash", but not "crashed" or "crashes". You could make whole file entries for each key variation, repeating the value paragraph. A concise approach is to use a data file like help_not_responses2.txt. Here are the first few lines:

```
crash crashes crashed
Well, it never crashes on our system.
It must have something to do with your system.
Tell me more about your configuration.

slow slowly
I think this has to do with your hardware.
Upgrading your processor should solve all
performance problems.
Have you got a problem with our software?

performance
Performance was quite adequate in all our tests.
Are you running any other processes in the background?
```

The line that used to have one key now may have several blank-separated keys.

Here is how the documentation for GetDictionary should be changed:

```
/// Return a new Dictionary, taking data for it from reader.
/// Reader generates key-value pairs, where one or more space
/// separated keys on a line are followed by a possibly multi-line
/// paragraph value that is terminated by an empty line.  Each
/// key on the line is mapped to the same paragraph that follows.
/// The file must end with two newlines in sequence:  one at the end
/// of the last nonempty line followed by one for the empty line.
```

Modify the lab project to use this file effectively: Find "help_not_responses.txt" on line 22 in Main. Change it to

"help_not_responses2.txt" (inserting '2'), so `Main` reads it.

In your test of the program, be sure to use several of the keys that apply to the same response, and show to your TA.

12.5 Chapter Review Questions

1. Which is true for a `Dictionary`: is it mutable or immutable?

2. Though for some collections, like arrays and lists, you can fairly easily replace a `foreach` loop with a `for` loop, that is not the case if you want to iterate through a `Dictionary`. How do you go through all the keys in a `Dictionary`?

3. What syntax is there for a `Dictionary` that matches that for a `List`?

4. How is a `Dictionary` like an array? How is it different?

5. `Dictionary` values are of arbitrary type. What is the restriction on key types?

6. How is a `HashSet` different than a `List`?

7. What syntax is shared between a `List` and a `HashSet`?

8. Which is more efficient in general: searching for an element of a list or finding the value given a key in a dictionary?

CLASSES AND OBJECT-ORIENTED PROGRAMMING

13.1 A First Example of Class Instances: Contact

13.1.1 Making a Datatype

C# comes with lots of built-in datatypes, but not everything we might want to use. We start with a very simple example of building your own new type of object: Contact information for a person involves several pieces of data, and they are all unified by the fact that they are for one person, so we would like to store them together as a unit. For simplicity, let us just consider the contact information to be name, phone number, and email address.

You could always keep three independent string variables, but conceptually the main idea is *the* contact. It just happens to have parts.

In order to treat a contact as *one* entity, we create a `class`, `Contact`. This way we can have a single variable refer to a `Contact` object. Such an object is an *instance* of the class.

It is important to distinguish between a class and an instance of a class: A class provides a template or instructions to make new instance objects. A common comparison is that a class is like a cookie cutter while an instance of the class is like a cookie. You might consider constructor parameters as being for different

decorations on different cookies, so not all cookies must end up completely the same.

Later we will see an example for rational numbers, *The Rational Class*, where the parts of the class are more tightly integrated, but that is more complicated, so we defer it.

We have already considered built-in types with internal state, like a `List`: Each `List` can contain different data, and the internal data can be changed.

The idea of creating a new type of object opens new ground for managing data. Thus far we have stored data as local variables, and we have called functions, with two ways to get information in and out of functions:

1. In through parameters and out through returned data.

2. Directly via the user: in through the keyboard and out to the screen.

We have stored and passed around built-in types of object using this model.

We have alternatives for storing and accessing data in the methods within a new class that we write. Now we have the idea of an object that has *internal* state (like a contact with a name, phone, and email). We shall see that this state is *not* stored in local variables and does *not* need to be passed through parameters for methods *within* the class. Pay careful attention as we introduce this new location for data and the new ways of interacting with it.

This is quite a shift. *Do not take it lightly.*

We can create a new object with the `new` syntax. We can give parameters defining the initial state of the new object. In our example the obvious thing to do is supply parameters giving values for the three parts of the object state, so we can plan that

```
Contact c = new Contact("Marie Ortiz", "773-508-7890", "mortiz2@luc.edu");
```

would create a new Contact storing the data. A Contact object, created with `new` is an *instance* of the class Contact.

Like with built-in types, we can have the natural operations on the type as *methods*. For instance we can

- look up individual pieces of the contact data with methods `GetName`, `GetPhone` and `GetEmail`

- print it all out together, labeled, with method `Print`.

Thinking ahead to what we would like for our Contact objects, here is the testing code of contact1/test_contact1.cs:

```csharp
using System;

namespace IntroCS
{
   public class TestContact
   {
      public static void Main()
      {
         Contact c1 = new Contact("Marie Ortiz", "773-508-7890",
                                  "mortiz2@luc.edu");
         Contact c2 = new Contact("Otto Heinz", "773-508-9999",
                                  "oheinz@luc.edu");
         Console.WriteLine("Marie's full name: " + c1.GetName());
         Console.WriteLine("Her phone number: " + c1.GetPhone());
         Console.WriteLine("Her email: " + c1.GetEmail());
         Console.WriteLine("\nFull contact info for Otto:");
         c2.Print();
      }
   }
}
```

When running this testing code, we would like the results:

```
Marie's full name: Marie Ortiz
Her phone number: 773-508-7890
Her email: mortiz2@luc.edu

Full contact info for Otto:
Name:  Otto Heinz
Phone: 773-508-9999
Email: oheinz@luc.edu
```

We are using the same object oriented notation that we have for many other classes: *Calls to instance methods are always attached to a specific object.* That has always been the part through the `.` of

> *object* . *method* (...)

So far we have been thinking and illustrating how we would like objects in this Contact class to look like and behave from the *outside*. We could be describing

---

13.1. A First Example of Class Instances: Contact   **375**

another library class. Now, for the first time, we start to delve inside, to the code and concepts needed to make this happen. We start with the most basic parts. First we need a `class`:

13.1.2 Class Syntax

Our code is nested inside

```
public class Contact
{

    // ... fields, constructor, code for Contact omitted

}
```

This is the same sort of wrapper we have used for our Main programs! *Before*, everything inside was labeled `static`. Now we see what happens with the `static` keyword *omitted*....

13.1.3 Instance Variables

A Contact has a name, phone number and email address. We must remember that data. Each individual Contact that we use will have its own name, phone number and email address.

We have used some static variables before in classes, with the keyword `static`, where there is just one copy for the whole class. If we omit the `static` we get an *instance variable*, that is the particular data for an *individual* Contact, for an individual instance of the class. This is our new place to store data:

We declare these *in* the class and *outside* any method declaration. (This is in the same place as we would store *Static Variables*).

They are *fields* of the class. As we will discuss more in terms of safety and security, we add the word "private" at the beginning:

```
public class Contact
{
    private string name;
    private string phone;
```

```
    private string email;

    // ... constructor, code for Contact omitted

}
```

You also see that we are lazy in this example, and abbreviate the longer descriptions fullName, phoneNumber and emailAddress.

It is important to distinguish *instance* variables of a class and *local* variables. A local variable is only accessible inside the block (surrounded by braces) where it was declared, and is destroyed at the end of the execution of that block. However the class fields `name`, `phone` and `email` are remembered by C# as long as the Contact object is in use.

The *lifetime* of a variable is from when it is first created until it is no longer accessible by the program. We repeat:

Note: *Instance* variable have a completely different lifetime and scope from *local* variables. An object and its instance variables, persist from the time a new object is created with `new` for as long as the object remains referenced in the program.

We need to get values into our field variables. They describe the state of our Contact.

We have *used* constructors for built-in types. Now for the first time we *create* one.

13.1.4 Constructors

The constructor is a slight variation on a regular method: Its name is the same as the kind of object constructed, so here it is the class name, Contact. It has *no return type* (and *no* `static`). Implicitly you are creating the kind of object named, a **Contact** in this case. The constructor can have parameters like a regular method. We will certainly want to give a state to our new object. That means giving values to its fields. Recall we are want to store this state in instance variables `name`, `phone` and `email`:

```
    public Contact(string fullName, string phoneNumber, string emailAddress)
    {
        name = fullName;
```

13.1. A First Example of Class Instances: Contact   **377**

```
        phone = phoneNumber;
        email = emailAddress;
    }
```

While the local variables in the formal parameters disappear after the constructor terminates, we want the data to live on as the state of the object. In order to remember state after the constructor terminates, we must *make sure the information gets into the instance variables* for the object. This is the basic operation of most constructors: Copy desired formal parameters in to initialize the state in the fields. That is all our simple code above does.

Note that `name`, `phone` and `email` are *not* declared as local variables. They refer to the *instance* variables, but we are *not* using full object notation: an object reference and a dot, followed by the field.

So far always we have always been referring to a built-in type of object defined in a different class, like `arrayObject.Length`. The constructor is *creating* an object, and the use of the bare instance variable names is understood to be giving values to the instance variables in this Contact object that is being constructed. Inside a constructor and also inside an instance method (discussed below) C# allows this shorthand notation without `someObject..`

13.1.5 Instance Methods

The instance variable names and method names used without an object reference and dot refer to the *current* instance. Whenever a constructor or non-static method in the class is called, there is *always* a current object:

1. In a constructor, referring to the object being created. In execution, a static method like `Main` must create the first object.

2. When some instance method `methodName` is called with explicit dot notation, `someObject.methodName()`, then it is acting on the current object `someObject`. In any program's execution the first call to an instance method must either be in this explicit form or from within a constructor for a new object.

3. If that constructor or instance method calls a further instance method inside the same class, without using dot notation, then the further method has the *same* current object.... We will see examples of this as we go along.

Again, this means that in execution, whenever an instance method is called, there *is* a *current specific object*. This is the object associated with any instance variable or method referred to in that method, if there is not an explicit prefix in the `someObject.` form. This will take practice to get used to.

13.1.6 Getters

In instance methods you have an extra way of getting data in and out of the method: Reading or setting instance variables. (As we have just pointed out, in execution there will always be a current object with its specific state.) The simplest methods do nothing but reading or setting instance variables. We start with those:

The `private` in front of the field declarations was important to keep code outside the class from messing with the values. On the other hand we do want others to be able to *inspect* the name, phone and email, so how do we do that? Use **public methods**.

Since the fields are accessible anywhere *inside* the class's instance methods, and public methods can be used from *outside* the class, we can simply code

```csharp
public string GetName()
{
    return name;
}

public string GetPhone()
{
    return phone;
}

public string GetEmail()
{
    return email;
}
```

These methods allow one-way communication of the name, phone and email values out from the object. These are examples of a simple category of methods: A *getter* simply returns the value of a part of the object's state, without changing the object at all.

Note again that there is no static in the method heading. The field value for the *current* Contact is returned.

A standard convention that we are following: Have getter methods names start with "Get", followed by the name of the data to be returned.

In this first simple version of Contact we add one further method, to print all the contact information with labels.

```csharp
    public void Print()
    {
        Console.WriteLine (@"Name:  {0}
Phone: {1}
Email: {2}", name, phone, email);
    }
```

Again, we use the instance variable names, plugging them into a format string. Remember the @ syntax for multiline strings from *String Special Cases*.

You can see and see the entire Contact class in contact1/contact1.cs.

This is our first complete class defining a new type of object. Look carefully to get used to the features introduced, before we add more ideas:

13.1.7 This Object

We will be making an elaboration on the Contact class from here on. We introduce new parts individually, but the whole code is in contact2/contact2.cs.

The current object is *implicit* inside a constructor or instance method definition, but it can be referred to *explicitly*. It is called this. In a constructor or instance method, this is automatically a legal local variable to reference. You usually do not need to use it explicitly, but you could. For example the current Contact object's name field could be referred to as either this.name or the shorter plain name. In our next version of the Contact class we will see several places where an explicit this is useful.

In the first version of the constructor, repeated here,

```csharp
    public Contact(string fullName, string phoneNumber, string emailAddress
    {
        name = fullName;
```

```
        phone = phoneNumber;
        email = emailAddress;
    }
```

we used different names for the instance variables and the formal parameter names that we used to initialize the instance variables. We chose reasonable names, but we are adding extra names that we are not going to use later, and it can be confusing. The most obvious names for the formal parameters that will initialize the instance variables are the *same* names.

If we are not careful, there is a problem with that. An instance variable, however named, and a local variable are not the same. This is nonsensical:

```
public Contact(string name, string phone, string email)
{
    name = name; // ????
    ...
```

Logically we want this pseudo-code in the constructor:

> instance variable `name` = local variable `name`

We have to disambiguate the two uses. The compiler always looks for *local* variable identifiers *first*, so plain `name` will refer to the local variable `name` declared in the formal parameter list. This local variable identifier *hides* the matching instance variable identifier. We have to do something else to refer to the instance variable. The explicit `this` object comes to the rescue: `this.name` refers to a part of this object. It must refer to the instance variable, not the local variable. Our alternate constructor is:

```
public Contact(string name, string phone, string email)
{
    this.name = name;
    this.phone = phone;
    this.email = email;
}
```

13.1.8 Setters

The original version of Contact makes a Contact object be *immutable*: Once it is created with the constructor, there is no way to change its internal state. The

only assignments to the private instance variables are the ones in the constructor. In real life people can change their email address. We might like to allow that with our Contact objects. Users can read the data in a Contact with the *getter* methods. Now we need *setter* methods. The naming conventions are similar: start with "Set". In this case we must supply the new data, so setter methods need a parameter:

```csharp
public void SetPhone(string newPhone)
{
    phone = newPhone;
}
```

In `SetPhone`, like in our original constructor, we illustrate using a *new* name for the parameter that sets the instance variable. For comparison we use the alternate identifier matching approach in the other setter:

```csharp
public void SetEmail(string email)
{
    this.email = email;
}
```

Now we can alter the contents of a Contact:

```csharp
Contact c1 = new Contact("Marie Ortiz", "773-508-7890", "mortiz2@luc.edu");
c1.SetEmail ("maria.ortiz@gmail.com");
c1.SetPhone("555-555-5555");
c1.Print();
```

would print

```
Name:  Marie Ortiz
Phone: 555-555-5555
Email: maria.ortiz@gmail.com
```

13.1.9 ToString Override

We created the `Print` method for a Contact, and it is helpful to assemble all the data for display *and* print it. The issue there is that the method does two *separate* clear things: combining the string data and printing it. You might want the same string but put in a file or used some other way. Our `Print` method will not help.

A good design decision is to separate the different actions: the first is to generate the 3-line string showing the full state of the object. Once we have this string, we can easily print it or write it to a file or .... Hence we want a method to generate a descriptive string.

Think more generally about string representations: All the built-in types can be concatenated into strings with the '+' operator, or displayed with `Console.Write`. We would like that behavior with our custom types, too. How can the compiler know how to handle types that were *not invented* when the compiler was written?

The answer is to have common features among all objects. Any object has a `ToString` method, and that method is used implicitly when an object is used with string concatenation, and also for `Write`. The default version supplied by the system is not very useful for an object that it knows nothing about! You need to define your own version, one that knows how you have defined your type, with its own specific instance variables. You need to have that version used *in place of* the default: You need to *override* the default. To emphasize the *change* in meaning, the word `override` *must* be in the heading:

```csharp
public override string ToString()
{
   return string.Format (@"Name:   {0}
{1}
{2}", name, phone, email);
}
```

See what the method does: it uses the object state to create and *return* a single string representation of the object.

For any kind of new object that you create and want to be able to implicitly convert to a string, you need a `ToString` method with the *exact* same heading as the `ToString` for a Contact.

A more complete discussion of `override` would lead us into class hierarchies and inheritance, which we are not emphasizing in this book.

We still might like to have a convenience method `Print`. It now can be written much more easily, using our latest `ToString`.

We want one instance method, `Print` to call another instance method `ToString` for the *same* object. How does this work? It is like when instance method `GetName` refers to an instance variable `name` without using dot notation.

Then `name` is assumed to refer to this object associated with the call to `GetName`. We can change our `Print` definition to:

```
public void Print()
{
   Console.WriteLine(ToString());
}
```

Here `ToString()` is a method called without dot notation explicitly attaching it to an object. As with instance variables, it is implicitly attached to this object, the one attached to the call to `Print`.

Again, the whole code for the elaborated Contact is in example con-tact2/contact2.cs.

New testing code is in contact2/test_contact2.cs. Run the project and check that it does what you would expect. There are several new features illustrated in the testing code:

```
public static void Main()
{
   Contact c1 = new Contact("Marie Ortiz", "773-508-7890",
                            "mortiz2@luc.edu");
   Contact c2 = new Contact("Otto Heinz", "773-508-9999",
                            "oheinz@luc.edu");
   Console.WriteLine("Marie's full name: " + c1.GetName());
   Console.WriteLine("Her phone number: " + c1.GetPhone());
   Console.WriteLine("Her email: " + c1.GetEmail());
   Console.WriteLine("All together:\n{0}", c1);
   Console.WriteLine("Full contact info for Otto:");
   c2.Print();
   c1.SetEmail("maria.ortiz@gmail.com");
   c2.SetPhone("123-456-7890");
   Contact c3 = new Contact("Amy Li", "847-111-2222",
                            "amy.li22@yahoo.com");
   Console.WriteLine("With changes and added contact:");
   var allc = new List<Contact>(new Contact[] {c1, c2, c3});
   foreach(Contact c in allc) {
      Console.WriteLine("\n"+c);
   }
}
```

Contact is now a type we can use with other types. `Main` ends creating a

List<Contact> and an array of Contacts, and processes Contacts in the List with a foreach loop.

We mentioned that this particular signature in the ToString heading means that the system recognizes it in string concatenation and in substitutions into a Write or WriteLine format string. Find both in Main.

The ToString override also means that the body of our Print definition in the Contact class could have been even shorter, using the object this:

```csharp
public void Print()
{
    Console.WriteLine(this);
}
```

When we use Console.WriteLine on this current object, which is *not* already a string, there is an automatic call to ToString.

13.1.10 Local Variables Hiding Instance Variables

A common error is for students to try to declare the instance variables twice, once in the regular instance variable declarations, *outside of any constructor or method* and then again *inside* a constructor, like:

```csharp
public Contact(string fullName, string phoneNumber, string emailAddress)
{
    string name = fullName;       // LOGICAL ERROR!
    string phone = phoneNumber;   // LOGICAL ERROR!
    string email = emailAddress; // LOGICAL ERROR
}
```

This is deadly. It is worse than redeclaring a local variable, which at least will trigger a compiler error.

> **Warning:** Instance variable *only* get declared outside of all functions and constructors. Same-name local variable declarations hide the instance variables, but *compile just fine*. The local variables disappear after the constructor ends, leaving the instance variables *without* your desired initialization. Instead the hidden instance variables just get the default initialization, null for an object or 0 for a number.

There is a related strange compiler error. This is not likely to happen frequently, but thinking through its logic (or illogic) could be helpful in understanding local and instance variables: Generally when you get a compiler error, the error is at or *before* the location the error is referenced, but with local variables covering instance variables, the real cause can come later in the text of the method. Below, when you first refer to r in Badnames, it appears to be correctly referring to the instance variable r:

```csharp
class ForwardError
{
    private int r = 3;

    // ...

    void BadNames(int a, int b)
    {
        int n = a*r + b; // legal in text *just* to here; instance field r
        //...
        int r = a % b; // r declaration makes *earlier* line wrong
        //...
    }
```

The compiler scans through *all* of BadNames, and sees the r declared locally in its scope. The error may be marked on the earlier line, where the compiler then assumes r is the later declared local int variable, not the instance variable. The error it sees is a local variable used before declaration.

This is based on a real student example. This example points to a second issue: using variable names that that are too short and not descriptive of the variable meaning, and so may easily be the same name as something unrelated.

Lifetime and Scope Exercise/Example

Be careful to distinguish lifetime and scope. Either a local variable or an instance variable can be temporarily out of scope, but still be alive. Can you construct an example to illustrate that? One of ours is lifetime_scope/lifetime_scope.cs.

13.2 Class Instance Examples

13.2.1 More Getters and Setters

As an exercise you could write a simple class Example, with

1. `int` instance variable n and `double` instance variable d

2. a simple constructor with parameters to set instance variables n and d

3. getters and setters for both instance variables (4 methods in all)

4. a ToString that prints a line with both instance variables labeled.

Compare yours to example_class/example_class.cs.

We include a testing class at the end of this file.

```
class ExampleTest
{
   public static void Main()
   {
      Example e = new Example(1, 2.5); // use constructor
      // this creates the first Example object with reference e
      Console.WriteLine("e.n = {0}", e.GetN()); // prints 1
      Console.WriteLine("e.d = {0}", e.GetD()); // prints 2.5
      Console.WriteLine(e); // prints Example: n = 1, d = 2.5

      e.SetN(25);
      e.SetD(3.14159);
      Console.WriteLine("e.n = {0}", e.GetN()); // prints 25
      Console.WriteLine("e.d = {0}", e.GetD()); // prints 3.14159
      Console.WriteLine(e); // prints Example: n = 25, d = 3.14159

      Example e2 = new Example(3, 10.5);
      // this creates the second Example object with reference e2
      Console.WriteLine(e2); // prints Example: n = 3, d = 10.5

      e = e2; // now both e and e2 reference the second object
      // the first Example object is now no longer referenced
      // and its memory can be reclaimed at runtime if necessary
      Console.WriteLine(e); // prints Example: n = 3, d = 10.5
```

```
      e2.SetN(77);  // symbolism uses e2, not e
      Console.WriteLine(e2); // prints Example: n = 77, d = 10.5
      Console.WriteLine(e); // prints Example: n = 77, d = 10.5
      // but e is the same object - so its fields are changed
   }
} // end of class ExampleTest
```

Besides the obvious tests, we also use the fact that an Example object is mutable to play with *References and Aliases*: In the last few lines of Main, after e becomes an alias for e2, we change an object under one name, and it affect the alias the same way. Check this by running the program!

Make sure you can follow the code and the output from running.

Beyond Getters and Setters

Thus far we have had very simple classes with instance variables and getter and setter methods, and maybe a ToString method. These classes were designed to introduce the basic syntax for classes with instances. The classes have just been containers for data that we can read back, and change if there are setter methods - pretty boring and limited. Many objects have more extensive behaviors, so we will take a small step and imagine a somewhat more complicated Averager class:

- A new Averager starts with no data acknowledged.

- Be able to enter data values one at a time (method AddDatum). We will use double values.

- At any point be able to return the average of the numbers entered so far (method GetAverage). The average is the sum of all the values divided by number of values. With double values we assume a double average. This does not make sense if there are no values so far, but with double type we can use the value NaN (Not a Number) in this case.

- Be able to return the number of data elements entered so far (method GetDataCount)

- Be able to clear the Averager, going back to no data elements considered yet, like in a new Averager (method Clear)

- It is good to have a ToString method. We choose to have it label the number of data items and the average of the items.

The object starts from a fixed state (no data) so we do not need any constructor parameters.

We can imagine a demonstration class `AveragerDemo` with a `Main` method containing

```
Averager a = new Averager();
Console.WriteLine("New Averager: " + a);
foreach (double x in new[] {5.1, -7.3, 11.0, 3.7}) {
    Console.WriteLine ("Next datum " + x);
    a.AddDatum (x);
    Console.WriteLine("average {0} with {1} data values",
                      a.GetAverage(), a.GetDataCount());
}
a.Clear();
Console.WriteLine("After clearing:");
Console.WriteLine("average {0} with {1} data values",
                  a.GetAverage(), a.GetDataCount());
```

It should print

```
New Averager: items: 0; average: NaN
Next datum 5.1
average 5.1 with 1 data values
Next datum -7.3
average -1.1 with 2 data values
Next datum 11
average 2.93333333333333 with 3 data values
Next datum 3.7
average 3.125 with 4 data values
After clearing:
average NaN with 0 data values
```

Now that we have a clear idea of the proposed outward behavior, we can consider how to implement the `Averager` class.

We could store a list of all the data values entered in any instance, requiring a large amount of memory for a long list. However, this functionality was built into early calculators, with very limited memory. How can we do it without remembering the whole list? Consider a concrete example:

If I have entered numbers 2.1, 4.5 and 5.4, and want the average, it is

$$(2.1 + 4.5 + 5.4)/3 = 12.0/3 = 4.0$$

13.2. Class Instance Examples 389

If I want to include a further number 5.0, the average becomes

$$(2.1 + 4.5 + 5.4 + 5.0)/4 = 17.0/4 = 4.25$$

Note the relationship to the previous average expression:

$$= ((2.1 + 4.5 + 5.4) + 5.0)/4 = (12.0 + 5.0)/(3 + 1)$$

In the numerator we have added the latest value to the previous sum, and in the denominator the count of data items is increased by one. All we need to remember to go on to include the next item is the sum of values so far and the number of values so far. We can just have instance variables sum and dataCount.

You might think how to create this class....

The full Averager code follows:

```csharp
using System;
namespace IntroCS
{
/// a class that is more than a container
   class Averager
   {
      private int dataCount;
      private double sum;

      /// new Averager with no data
      public Averager()
      {
         Clear();
      }

      public void AddDatum(double value)
      {
         sum += value;
         dataCount++;
      }

      public int GetDataCount()
      {
         return dataCount;
      }

      /// Gets the average of the data
```

```
    ///    or NaN if no data.
    public double GetAverage()
    {
       return sum/dataCount;  // is NaN if dataCount is 0
    }

    public void Clear()
    {
       sum = 0.0;
       dataCount = 0;
    }

    public override string ToString ()
    {
       return string.Format("items: {0}; average: {1}",
                            GetDataCount(), GetAverage());
    }
  }
}
```

Several things to note:

- We show that a constructor, like an instance method, can include a call to a further instance method. Though we illustrate this idea, the constructor code is actually unneeded. See the *Unneeded Constructor Code Exercise* below.

- We have methods that are not ToString or mere getters or setters of instance variables. The logic of the class requires more.

- The GetAverage method does return data obtained by reading instance variable, but it does a calculation using the instance variables first. See *Alternate Internal State Exercise*.

The code for both classes is in project averager.

Statistics Exercise

Modify the project averager so the Averager class is converted to Statistics. Besides having methods to calculate the count of data items and average, also calculate the standard deviation with a method

---

`StandardDeviation`. It turns out that the only other instance variable needed is the sum of the squares of the data items, call it `sumOfSqr`. Before calculating the standard deviation, suppose we assign the current average to a local variable `avg`. Then the handiest form of expression for the standard deviation is

```
Math.Sqrt((sumOfSqr - avg*avg)/dataCount)
```

Modify the demonstration program to show the standard deviation, too.

Unneeded Constructor Code Exercise

Recall that objects are always initialized. Each instance variable has a default value assigned before a constructor is even run. The default value for numeric instance variables is 0, so the call to `Clear` in the constructor could be left out, leaving the body of the constructor empty! Try commenting that line out and test that the behavior of demo program is the same.

Emphasizing the fact that you are thinking about the initial values of instance variables is not a bad idea. Hence a common practice is to explicitly assign even the default values in the constructor, as we did initially with the call to `Clear` inside the constructor.

If no constructor definition is explicitly provided at all, the compiler implicitly creates one with no parameters and an empty body. This means that the entire constructor in `Averager` could be omitted. Comment the whole constructor out and check.

There is a defensive programming reason to provide even the do-nothing constructor explicitly: If you use the implicit constructor and then decide to add a constructor with parameters, the implicit constructor is no longer defined by the compiler, so any remaining call to it in your code becomes an error.

Alternate Internal State Exercise

The way we represent the internal state for an `Averager` is the best probably, but if it turns out that the `GetAverage` method is called a lot more often than a method that changes the state, we could avoid repeated division by saving the average as an instance variable. We could keep that *instead of* `sum` (and still keep `dataCount`). We can manage to keep the same public interface to the methods.

With these alternate instance variables how would you change the implementation code and not change the method headings or meanings? If we keep the assumption that the average of 0 items is `double.Nan`, you will need to treat adding the first datum as a special case. The code is simpler if we change the outward assumptions enough to consider the average of 0 items to be 0. Try it either way.

In the version with NaN you can avoid testing for NaN, but if you choose to test for NaN, you need the boolean `Double.IsNaN` function, because the expression `double.NaN == double.NaN` is *false*!

13.2.2 Converting A Static Game To A Game Instance

For a comparison of procedural and object-oriented coding, consider converting *Number Guessing Game Lab* so that a GuessGame is an object, an instance of the GuessGame class.

While our earlier example, Contact, is a simple but practical use of object-oriented programming, GuessGame is somewhat more artificial. We create it hoping that highlighting the differences between procedural and object-oriented presentation is informative. Also, we will see with *Interfaces* that there are C# language features that require an object rather than just a function and data. In *IGame Interface Exercise* you can use a Game object.

Here is a procedural game version, example file static_version/static_version.cs

```csharp
public class Game
{
    private static Random r = new Random();

    public static void Main()
    {
        int big = UI.PromptInt("Enter a secret number bound: ");
        Play(big);
    }

    public static void Play(int big)
    {
        int secret = r.Next(big);
        int guesses = 1;
        Console.WriteLine("Guess a number less than {0}.", big);
        int guess = UI.PromptInt("Next guess: ");
```

```
        while (secret != guess) {
          if (guess < secret) {
             Console.WriteLine("Too small!");
          } else {
             Console.WriteLine("Too big!");


          }
          guess = UI.PromptInt("Next guess: ");
          guesses++;
       }
       Console.WriteLine("You won on guess {0}!", guesses);
    }
  }
```

The project also refers to the library class UI, with the functions we use for safe keyboard input. It is all static methods.

Is there any reason to make this UI class have its own own instances?

No. There is no state to remember between UI method calls. What comes in through the keyboard goes out through a return value, and then you are completely done with it. A simple static function works fine each time. *Do not get fancy for nothing*.

What state would a game hold? We might set it up so the user chooses the size of the range of choices just once, and remember it for possibly multiple plays of the GuessGame. The variable was big before, we can keep the name. If we are going to remember it inside our GuessGame instance, then big needs to become an instance variable, and it will be something we can set in a constructor.

What actions/methods will this object have? Only one - playing a GuessGame. The GuessGame could be played multiple times, and that action, play, makes sense as a method, Play, which will look a lot like the current static function.

In the procedural version there are several other important variables:

- Random rand: That was static before, for good reason: We only need one Random number generator for the whole time the program is running, so one static variable makes sense.

- The central number in the procedural Game and our future Play method is secret. Should that be an instance variable? It would work, but it would be unhelpful and misleading: Secret is reset every time the game is played,

and it has no meaning after a Play function would be finished. There is nothing to remember *between* time you Play. This is the perfect place for a local variable *as we have now*.

A common newbie error is to make things into instance variables, just because you can, when an old-fashioned local variable is all that you need. It is good to have variables leave the programmer's consciousness when they are no longer needed, as a local variable does. An instance variable lingers on, leaving extra places to make errors.

This introductory discussion could get you going, making a transformation. Go ahead and make the changes as far as you can: create project GuessGame inside the current solution. Have a class GuessGame for the GuessGame instance, with instance variable `big` and method `Play`.

You still need a static `Main` method to first create the GuessGame object. You could prompt the user for the value for `big` to send to the constructor. Once you have an object, you can call *instance method* `Play`. What about parameters? What needs to change from the procedural version?

There is also a video for this section that follows all the way through the steps. A possible final result is in instance_version/guess_game.cs.

13.2.3 Animal Class Lab

Objectives: Complete a simple (silly) class, with constructor and methods, including a `ToString` method, and a separate testing class.

Make an animal_lab project in your solution, and copy in the files from the example project animal_lab_stub. Then modify the two files as discussed below.

1. Complete the simple class Animal in your copy of the file animal.cs. The bullets below name and describe the instance variables, constructor, and methods you need to write:

   - An Animal has a `name` and a `gut`. In our version the `gut` is a List of strings describing the contents, in the order eaten. A newly created Animal gets a `name` from a parameter passed to the constructor, while the `gut` always starts off *empty*.

   - An Animal has a `Greet` method, so an animal named "Froggy" would say (that is, print)

> Hello, my name is Froggy.

- An Animal can `Eat` a string naming the food, adding the food to the `gut`. If Froggy eats "worm" and then "fly", its `gut` list contains "worm" and "fly".

- An Animal can `Excrete` (removing and printing what was *first* in the gut List). Recall the method `RemoveAt` in *List Syntax*. Print the empty string, "", if the `gut` *was already empty*. Following the Froggy example above, Froggy could `Excrete`, and "worm" would be printed. Then its `gut` would contain only "fly".

- A `ToString` method: Remember the `override` keyword. Make it return a string in the format shown below for Froggy, including the Animal's name:

  > "Animal: Froggy"

  Try this first, and note the extra credit version below.

- All the methods that print should be void. Which need a parameter, of what type?

2. Complete the file test_animal.cs with its class `TestAnimal` containing the `Main` method, testing the class Animal: Create a couple of Animals and visibly test all the methods, with enough explanation that someone running the test program, but *not* looking at the code of either file, can see that everything works.

3. 20% EXTRA CREDIT: Elaborate `ToString` so if Froggy had "worm", "fly" and "bug" in the gut, the string would be:

   > "Animal: Froggy ate worm, fly and bug"

with a comma separated list of the gut contents, except use proper English, so the last separator is " and ", not ", ". If the gut has nothing in it, list the contents as "nothing":

> "Animal: Froggy ate nothing"

13.2.4 Clock Example

Consider the logic for a digital 24 hour clock object, type `Clock`, that shows hours and minutes, so 03:45 would be three forty-five. Note that there is no AM or PM: The hours go from 00, starting at midnight, through hour 23, the 11PM hour, so 23:59 would be a minute before midnight, and 13:00 would be 1PM.

Assume there is some attached circuit to signal when a new minute starts.

This class could have just a few methods: `Tick`, called when a new minute is signaled, and `GetTimeString` to return the time in the format illustrated above, and `SetTime` specifying new values for the hours and minutes. We can start from a constructor that just sets the clock's time to midnight.

We can imagine a demonstration class `ClockDemo` with a `Main` method containing

```
Clock c = new Clock();
Console.WriteLine("Midnight " + c.GetTimeString());
c.SetTime(23, 58);
Console.WriteLine("Before midnight " + c.GetTimeString());
for (int i = 0; i < 4; i++) {
    c.Tick ();
    Console.WriteLine ("Tick " + c.GetTimeString());
}
```

It should print

```
Midnight 00:00
Before midnight 23:58
Tick 23:59
Tick 00:00
Tick 00:01
Tick 00:02
```

A `Clock` object will need instance variables. One obvious approach would be to have `int` instance variables for the hours and minutes. Both can be set and can advance and will need ot be read.

These actions are common to both the hours and minutes, so we might think how we can avoid writing some things twice. There is one main difference: The minutes roll over at 60 while the hours roll over at 24. Though the limits are different, they are both numbers, so we can store the limit for each, 60 or 24. Then the

same code could be used to advance each one, just using a different value for the rollover limit.

How would we neatly code this in a way that reuses code? The most significant thing to notice is that dealing with minutes involves data (the current count and the limit 60) and associated actions: being set, advanced and read. The same is true for the hours. The combination of *data and tightly associated actions*, particularly used in more than one situation, suggests a new class of objects, say RolloverCounter.

Notice the shift in this approach: The instance variables for hours and minutes would become instances of the RolloverCounter class. A RolloverCounter should know how to advance itself. Hence the logic for advancing a counter, sometimes rolling it over, would not be directly in the Clock class, but in the RolloverCounter class.

So let's think more about what we would want in the RolloverCounter class. What instance variables? Of course we have the current count, and since we want the same class to work for both minutes and hours, we also need to have the rollover limit. They are both integers.

The limit should just be set once for a particular counter, presumably when the object is created. For simplicity we can just assume the count is 0 when a RolloverCounter is first created. Of course we must have a method to let the count advance, rolling over back to 0 when the limit it reached.

Throw in a getter and a setter for the count and we can have the following class:

```
using System;
namespace IntroCS
{
/// class used twice in Clock
   class RolloverCounter
   {
      private int limit, count;

      public RolloverCounter(int limit)
      {
         this.limit = limit;
         count = 0;   //for clarity - this is the default value
      }

      public int GetCount()
```

```
      {
         return count;
      }

      public void SetCount(int count)
      {
         this.count = count;
      }

      /// advance by one time tick
      /// eventually roll over at limit
      public void Advance()
      {
         count = (count + 1) % limit;
      }
   }
}
```

Note how concise the Advance method is! With the remainder operation, we do not need an if statement. Check examples by hand if this seems strange.

Finally we introduce the Clock class itself. We display the entire code first, and follow it with comments about a number of new features.

```
using System;
namespace IntroCS
{
/// class with instance variables of another user class
   class Clock
   {
      private RolloverCounter hours, minutes;

      /// new Clock set to midnight
      public Clock()
      {
         hours = new RolloverCounter(24);
         minutes = new RolloverCounter(60);
      }

      ///  new Clock set to specified time
      public Clock(int nHours, int nMinutes)
      {
```

```
        hours = new RolloverCounter(24);
        minutes = new RolloverCounter(60);
        SetTime (nHours, nMinutes);
    }

    public void SetTime(int nHours, int nMinutes)
    {
        hours.SetCount(nHours);
        minutes.SetCount(nMinutes);
    }

    /// advance by one time tick
    public void Tick()
    {
        minutes.Advance();
        if (minutes.GetCount() == 0) {
            hours.Advance();
        }
    }

    /// Always 2 digits for both hours and minute with colon in middle
    public string GetTimeString()
    {
        return string.Format("{0:D2}:{1:D2}",
            hours.GetCount(), minutes.GetCount());
    }
  }
}
```

1. First the principal reason for this example: We illustrate writing a class where the instance variables are objects of a different user-defined type. Because the instance variables `hours` and `minutes` are objects, we must initialize them using the `new` syntax.

2. Skip over the *second* constructor for now, and see the `SetTime` method: We call the appropriate method to update the individual `RolloverCounter` instances.

3. Now go back to the second constructor. This is not really necessary: With the first constructor the calling code could just have one more `SetTime` line any time you want to to create a clock with a time other than midnight. We can make a case for this being so common, that we want to do it in

just one line, with a constructor that sets a specified time. However, the main excuse was really to illustrate that constructors can be *overloaded*, like methods: You can have separate constructors with distinct signatures. In this case versions with no parameters vs. two `int` parameters.

4. The `Tick` method has a bit of logic to it: while the minutes always advance, the hours only advance when the minutes roll over to 0.

5. Finally the `GetTimeString` method illustrates a new integer string formatting mode: The D2 format specifier applies to an integer, and displays it as a minimum of 2 digits, padding on the left with 0's as necessary. This is just what we want here. In general the 2 could be replaced by another literal integer, so D6 would force at least 6 digits: With the D6 format specifier 12 would be formatted as 000012, and the longer 1234567 would add no extra 0's: still 1234567.

The code for all the classes is in project clock.

Admittedly, with this exact functionality and such a concise line to advance a count, it would actually have shorter to have done everything inside the `Clock` class, with no `RolloverCounter`, but we were looking for a simple illustration of combining user-defined types this way, and a `RolloverCounter` is a clear unified concept that can be used in other situations. See an upcoming exercise.

Alternate Clock Constructor Exercise

Make a small change to clock/clock_demo.cs, so the second constructor is tested.

Clock With Seconds Exercise

Modify the project clock, assuming the Tick is for each second, and the time also show the seconds, like 55 seconds before midnight would be 23:59:05.

Twelve Hour Time Exercise

Modify the project clock so a `GetTimeString12` method returns the 12 hour time with AM or PM, like 11:05PM or 3:45AM. (The hours do not have a leading 0 in this format.) This could be done modifying a lot of things: keeping the actual hours and minutes that you will display and remembering AM or PM (with

---

the hours being more complicated, not starting at 0). We suggest something else instead:

This is a good place to note a very useful pattern for programming, called *model-view-controller*. The *model* is the way chosen to store the state internally. The *controller* has the logic to modify the model as it needs to evolve. A *view* of a part of the model is something shown to the user that does not need to be in the exact same form as the model itself: A view just needs to be something that can be *easily calculated* from the model, and presumably is desired by the user.

In this case a simple (and already coded!) way to store and control the time model data is the minutes and up to 23 hours that do happen to directly correspond to the 24 hour clock view.

The main control is to advance the time, and with just two 0-based counts we have the very simple remainder formulas.

So the suggestion is to keep the *internal* data the same way as before. Just in the method to create the desire 12-hour view have the logic to do the *conversion* of the internal 24-hour model data.

You could leave in the method to provide the time in the 24 hour format, giving the `Clock` class user the option to use either view of the shared model data. To be symmetrical in the naming, you might change the original name `GetTimeString` to `GetTimeString24`.

13.3 The Rational Class

Like other numbers, we think of a rational number as a unit, but all rational numbers can be most easily represented as a fraction, a ratio of two integers. We can create a class Rational, so

```
Rational r = new Rational(2, 3);
```

would create a new Rational number for the mathematical expression, 2/3.

Our previous simple class examples have mostly been ways of collecting related data, with limited methods beyond getters and setters. Rational numbers also have lots of obvious operations defined on them. Our Rational example we will use most of the concepts for object so far, and add a few more, to provide a rich practical class with a variety of methods.

Thinking ahead to what we would like for our rational numbers, here is some testing code. Hopefully the method names are clear and reasonable. in the illustration we operate on a single rational number, and do calculation with a pair, and parse string literals. The code is from test_rational/test_rational.cs:

```csharp
using System;

namespace IntroCS
{
    public class TestRational
    {
        public static void Main()
        {
            Rational f = new Rational(6, -10);
            Console.WriteLine("6/(-10) simplifies to {0}", f);
            Console.WriteLine("reciprocal of {0} is {1}", f, f.Reciprocal());
            Console.WriteLine("{0} negated is {1}", f, f.Negate());
            Rational h = new Rational(1, 2);
            Console.WriteLine("{0} + {1} is {2}", f, h, f.Add(h));
            Console.WriteLine("{0} - {1} is {2}", f, h, f.Subtract(h));
            Console.WriteLine("{0} * {1} is {2}", f, h, f.Multiply(h));
            Console.WriteLine("({0}) / ({1}) is {2}", f, h, f.Divide(h));
            Console.WriteLine("{0} > {1} ? {2}", h, f, (h.CompareTo(f) > 0));
            Console.WriteLine("{0} as a double is {1}", f, f.ToDouble());
            Console.WriteLine("{0} as a decimal is {1}", h, h.ToDecimal());
            foreach (string s in new[] {"-12/30", "123", "1.125"}) {
                Console.WriteLine("Parse \"{0}\" to Rational: {1}",
                                    s, Rational.Parse(s));
            }
        }
    }
}
```

One non-obvious method is `CompareTo`. This one method allows all the usual comparison operators to be used with the result. We will discuss it more in *Rationals Revisited*.

The results we would like when running this testing code:

```
6/(-10) simplifies to -3/5
reciprocal of -3/5 is -5/3
-3/5 negated is 3/5
-3/5 + 1/2 is -1/10
```

```
-3/5 - 1/2 is -11/10
-3/5 * 1/2 is -3/10
(-3/5) / (1/2) is -6/5
1/2 > -3/5 ? true
-3/5 as a double is -0.6
1/2 as a decimal is 0.5
Parse "-12/30" to Rational: -2/5
Parse "123" to Rational: 123
Parse "1.125" to Rational: 9/8
```

A Rational has a numerator and a denominator. We must remember that data. Each individual Rational that we use will have its own numerator and denominator, which we store as the instance variables (and which we abbreviate since we are lazy):

```csharp
public class Rational
{
    private int num;
    private int denom;
    // ...
```

We could have a very simple constructor that just copies in values for the numerator and denominator. However, there is an extra wrinkle with rational numbers: They can be represented many ways. You remember from grade school being told to "reduce to lowest terms". This will keep our internal representations unique, and use the smallest numbers.

Intermediate operations and initial constructor parameters will not always be in lowest terms. To reduce to lowest terms we need to divide the original numerator and denominator by their *Greatest Common Divisor*. We include a *static* GCD method taken from that section, and make the adjustment to lowest terms in a helping method, `Normalize`, that is called by the constructor:

```csharp
    /// Force the invarient: in lowest terms with a positive denominator.
    private void Normalize()
    {
        if (denom == 0) { // We *should* force an Exception, but we won't.
            Console.WriteLine("Zero denominator changed to 1!");
            denom = 1;
        }
        int n = GCD(num, denom);
        num /= n;          // lowest
```

```
      denom /= n;        //     terms
      if (denom < 0) {   // make denom positive
         denom = -denom;//    double negation:
         num = -num;     //    same mathematical value
      }
   }
```

There are several things to note about this method:

- It is *private*. It is only used as a helping method, called from inside of Rational. It is not a part of the public interface used from other classes.

- We need to deal with a 0 denominator somehow. We should be causing an *exception*, but that is an advanced topic, so we wimp out and just change the denominator to 1.

- There is one other technical issue in getting a unique representation: The denominator could start off being negative. If that is the case, we change the sign of both the numerator and denominator, so we always end up with a positive denominator. We will use this fact in several places.

- It calls a static method of the class, GCD. Classes can have both instance and static methods. It is fine for an instance method like Normalize to call a static method: The instance variables cannot be accessed. Here GCD is passed all its data explicitly through its parameters.

The complete constructor, using Normalize, is below. Note that by the time we are done constructing a new Rational, it is in this normalized form: lowest terms and positive denominator:

```
   /// Create a fraction given numerator and denominator.
   public Rational(int numerator, int denominator)
   {
      num = numerator;
      denom = denominator;
      Normalize();
   }
```

The call to the Normalize method is another place where we have a call without dot notation, since it is acting on the same object as for the constructor.

We mentioned that instance method Normalize calls static method GCD, and this is fine. The reverse is not true:

> **Warning:** Inside a `static` method there is *no* current object. A common compiler error is caused when you try to have a static method call an instance method without dot notation for a specific object. The shorthand notation without an explicit object reference and dot cannot be used, because there is no current object to reference implicitly:
>
> ```
> public void AnInstanceMethod()
> {
>         ...
> }
>
> public static void AStaticMethod()  // no current object
> {
>         AnInstanceMethod();  // COMPILER ERROR CAUSED
> }
> ```
>
> On the other hand, there is no issue when an instance method calls a static method. (The instance variables are just inaccessible inside the static method.)

The Rational class has the usual getter methods, to access the obvious parts of its state:

```
public int GetNumerator()
{
   return num;
}

public int GetDenominator()
{
   return denom;
}
```

We certainly want to be able to display a Rational as a string version:

```
/// Return a string of the fraction in lowest terms,
/// omitting the denominator if it is 1.
public override string ToString()
{
   if (denom != 1)
      return string.Format("{0}/{1}", num, denom);
   return ""+num;
```

```
        }
```

Note that we simplify so you would see "3" rather than "3/1". This is also a place where the normalization to have a positive denominator comes in: a negative Rational will always have a leading "-" as in "-5/9" rather than "5/-9"

With a Rational, several other conversions make sense: to `double` and `decimal` approximations.

```csharp
/// Return a double approximation to the fraction.
public double ToDouble()
{
    return ((double)num)/denom;
}

/// Return a decimal approximation to the fraction.
public decimal ToDecimal()
{
    return ((decimal)num)/denom;
}
```

So far we have returned built-in types. What if we wanted to generate the reciprocal of a Rational? That would be another Rational. It is legal to return the type of the class that you are defining! How do we make a new Rational? We have a constructor! We can easily use it. The reciprocal swaps the numerator and denominator. It is also easy to negate a Rational:

```csharp
/// Return a new Rational which is the reciprocal of this Rational.
public Rational Reciprocal()
{
    return new Rational(denom, num);
}

/// Return a new Rational which is this Rational negated.
public Rational Negate()
{
    return new Rational(-num, denom);
}
```

Static methods are still useful. For example, in analogy with the other numeric types we may want a static `Parse` method to act on a string parameter and return a new Rational.

The most obvious kind of string to parse would be one like `"2/3"` or `"-10/77"`, which we can split at the `'/'`. Integers are also rational numbers, so we would like to parse `"123"`. Finally decimal strings can be converted to rational numbers, so we would like to parse `"123.45"`.

See how our `Parse` method below distinguishes and handles all the cases. It constructs integer strings, `parts[0]` and `parts[1]`, for both the numerator and denominator, and then parses the integers. Note that the method *is* `static`. There is no Rational being referred to when it starts, but in this case the method *returns* one.

That last case is the trickiest. For example `"123.45"` becomes 12345/100 (before being reduced to lowest terms). Note that there were originally *two* digits after the decimal point and then the denominator gets *two* zeroes to have the right power of 10:

```
/// Parse a string of an integer, fraction (with /) or decimal (with .
///  and return the corresponding Rational.
public static Rational Parse(string s)
{
    s = s.Trim();         // will adjust numerator and denominator parts
    string[] parts = {s, "1"}; // for an int string, this is correct
    if (s.Contains("/")) {     // otherwise correct num, denom parts
        parts = s.Split('/');
    } else if (s.Contains(".")) {
        string[] intFrac = s.Split('.'); // integer, fractional digits
        parts[0] = intFrac[0]+intFrac[1];  // "shift" decimal point
        parts[1] = "1";                    // denom will have as many 0's
        foreach( char dig in intFrac[1]) {  //    as digits after '.'
            parts[1] += "0";                //    to compensate
        }
    }
    return new Rational(int.Parse(parts[0].Trim()),
                        int.Parse(parts[1].Trim()));
}
```

13.3.1 Method Parameters of the Same Type

We can deal with the current object without using dot notation. What if we are dealing with *more than one* Rational, the current one *and* another one, like the parameter in Multiply:

```
      /// Return a new Rational which is the product of this Rational and f.
      public Rational Multiply(Rational f)
```

We can mix the shorthand notation for the current object's fields and dot notation for another *named* object: `num` and `denom` refer to the fields in the *current* object, and `f.num` and `f.denom` refer to fields for the other `Rational`, the parameter `f`.

```
      /// Return a new Rational which is the product of this Rational and f.
      public Rational Multiply(Rational f)
      {  // end Multiply heading chunk
         return new Rational(num*f.num, denom*f.denom);
      }
```

We do not refer to the fields of `f` through the public methods `GetNumerator` and `GetDenominator`. Though `f` is not the same *object*, it is the same *type*:

Note: Private members of *another* object of the *same* type are accessible from method definitions in the class.

There are a number of other arithmetic methods in the source code for Rational that return a new Rational result of the arithmetic operation. They *do* review your knowledge of arithmetic! They do *not* add further C# syntax.

The whole code for Rational is in rational_nunit/rational.cs. The testing code we started with, in test_rational/test_rational.cs uses all the methods. We will see more advance ways to test Rational in *Testing*.

There is also a more convenient version of Rational, using advanced concepts, in *Defining Operators (Optional)*.

13.3.2 Pictorial Playing Computer

Let us start pictorially playing computer on `test_rational.cs`, as a review of much of the previous sections. We explicitly show a local variable `this` to identify the current object in an instance method or constructor.

The first line of `Main`,

```
Rational f = new Rational(6, -10);
```

creates a new Rational, so it calls the constructor. At the very beginning of the constructor, a prototype `Rational` is already created as the current object, so immediately, there is a `this`. The parameters 6, and -10 are passed, initializing the explicit local variables `numerator` and `denominator`. The figure illustrates the memory state at the beginning of the constructor call:
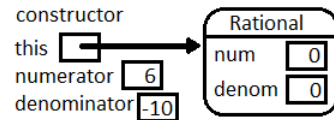
Note the immediate value assigned to the numeric instance variables is zero: This is as discussed in *Default Initializations*. Of course we do not want to keep those default values: The constructor finds the value of the local variable `numerator`, and needs to assign the value 6 to a variable `num`. The compiler has looked first for a local variable `num`, and found none. Then it looked *second* for an instance variable in the object pointed to by `this`. It found `num` there. Now it copies the 6 into that location. Similarly for `denominator` and `denom`:
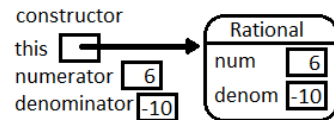
Then the constructor calls `Normalize`. Since `Normalize` is also an instance method, a reference to `this` is passed implicitly. While illustrating the memory state for more than one active method, we separate each one with a horizontal segment.
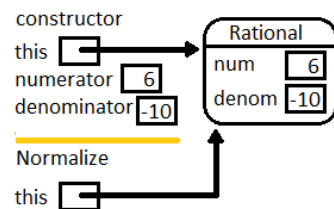
Later `Normalize` calls `GCD`. Since `GCD` is static, note that the local variables for `GCD` do *not* contain a reference to `this`.

At the end of GCD the int 2 is returned and initializes n in the calling method Normalize. Then Normalize modifies the instance variable pointed to by this, and finishes.

That is the same object this in the constructor. Just before the constructor completes we have:

Then in Main the constructor's this is the reference to the new object initializing f.

Consider the next line of Main:

```
Console.WriteLine("6/(-10) simplifies to {0}", f);
```

We omit the internals of the WriteLine call, except to note that it must convert the reference f to a string. As with any object, it does this by calling the `ToString` method for f, so the implicit `this` in the call to `ToString` refers to the same object as f:

`ToString` returns "-3/5", and it gets printed as part of the line generated by `WriteLine`....

We skip the similar details through two more `WriteLine` statements and the initialization of h:

```
Rational h = new Rational(1,2);
```

The `WriteLine` statement after that needs to evaluate `f.Add(h)`, generating a call to `Add`. The next figure shows the two local variables in `Main`, f and h, each pointing to a `Rational` object. The image shows the situation in the call to `Add`, just before the end of the return statement, when the new Rational is being constructed.

In the local variables for the method `Add` see what the implicit `this` refers to, and what the (local to `Add`) variable f refer to. As the figure shows, this use of a local variable f is independent of the f in `Main`:

Since the return statement in Add creates a new object, the figure shows a call to the constructor from inside Add. We do not go through the details of another constructor call, but this in the constructor points to the Rational shown and returned by Add:

which gets sent to the WriteLine statement and gets printed in Main as in the earlier code.

Make sure you see how the pictures reinforce these important ideas:

- Keeping track of the this with constructors and instance methods (but not static methods).

- The aliasing of Rational objects used as parameters explicitly or implicitly (this).

We have played computer before in procedural programming, following individually explicitly named variables. This has allowed us to follow loops clearly after the code is written. The pictorial version with multiple object references and method calls is also useful for checking on code that is written with many object references.

When first *writing* code with object references that you are manipulating, a picture of the setup showing the references in your data is also helpful. New object-oriented programmers often have a hard time referring to the data they want to work with. If you have a picture of the data relationships you can point with a finger to a part that you want to use. For example in the call to Add, one piece of data you need for your arithmetic is the num field in f. Then you must be carful to note that *only local variables can be referenced directly* (including the implicit this). If you want to refer to data that is not a local variable, you must follow the reference path arrow that leads *from a local variable* to an instance field that you want to reference.
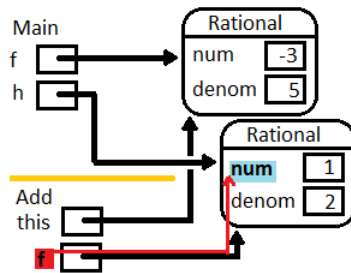
Then use the proper object-oriented notation to refer to the path. In the example, it takes one step, from local variable f to its field num, referred to as f.num. Similarly the current object's num is connected through this, but C# shorthand allows this. to be omitted. And so on, for f.denom and denom.

Visually following such paths will be even more important later, when we construct more complex types of objects, and you need to follow a path through *several* references in sequence.

13.3.3 ForceMatch Exercise

Suppose we have a class:

```
class Pair
{
   private int x,y;

   public Pair(int x, int y)
   {
       this.x = x; this.y = y;
   }

   ///Mutate the parameter so its instance variables match this object
   public void ForceMatch(Pair p)
   {
       // need code ...
   }

   public override string ToString()
   {
       return string.Format("({0}, {1})", x, y);
   }
}
```

A test would be code in another class:

```
Pair first = new Pair(3, 7);
Pair second = new Pair(1, 9)
Console.WriteLine(second);  // prints (1, 9)
first.ForceMatch(second);
Console.WriteLine(second);  // prints (3, 7)
```

1. Would this code work? If not, explain why not:

```
public void ForceMatch(Pair p)
{
   p = new Pair(x, y);
}
```

   If you do not see it, do a graphical play-computer like in the last section.

2. Complete the body of ForceMatch correctly.

13.4 Planning A Class Structure

The Console input/output interchange below illustrates an idea for a skeleton of
a text (adventure?) game. It could be the basis of a later group project. It does
not have much in it yet, but it can be planned in terms of classes. Classes with
instances correspond to nouns you would be using, particularly nouns used in
more than one place with different state data being remembered. Verbs associated
with nouns you use tend to be methods. Think how you might break this down,
looking at what is happening in the sequence below.

The parts appearing after the '>' prompt are entries by the user. Other lines are
computer responses:

```
Welcome to Loyola!
This is a pretty boring game, unless you modify it.
Type 'help' if you need help.

You are outside the main entrance of the university that prepares people for
extraordinary lives.  It would help to be prepared now....
Exits: east south west
> help
You are lost. You are alone.
You wander around at the university.

Your command words are:
   help go quit

Enter
       help command
for help on the command.
> help go
Enter
       go direction
to exit the current place in the specified direction.
The direction should be in the list of exits for the current place.
> go west
You are in the campus pub.
Exits: east
> go east
You are outside the main entrance of the university that prepares people for
extraordinary lives.  It would help to be prepared now....
```

```
Exits: east south west
> go south
You are in a computing lab.
Exits: north east
> go east
You are in the computing admin office.
Exits: west
> bye
I don't know what you mean...
> quit
Do you really want to quit? yes
Thank you for playing.  Good bye.
```

Think and discuss how to organize things first....

The different parts of a multi-class project interact through their public methods. Remember the two roles of writer and consumer. The consumer needs good documentation of how to use (not implement) these methods. These methods that allow the interaction between classes provide the *interface* between classes. Unfortunately "interface" is used in more than one way. Here it means publicly specified ways for different parts to interact.

As you think how to break this game into parts (classes), also think how the parts interact (public methods). This is a good place for the start of a class discussion.

If the plan is to discuss it in class, *wait* before looking at the code that generated the exchange above, in the project folder cs_project1.

The code uses many of the topics discussed so far in this book.

We will add some features from another meaning of *Interfaces*, and discuss the revision in project csproject_stub (no 1). You *might* use this version as a basis of a project.

13.5 Classes And Structs

C# has an alternate syntax to a class: a *struct*. Everything we have said so far about classes such an `Rational` applies to structs also! In fact you could change `class` into `struct` in the heading for Rational, and it would become a `struct`, with no further code changes in any of the code we have written!

```
public struct Rational
{
    // ...
}
```

So why the distinction? We have mentioned that new objects created in a class are accessed indirectly via a reference, as with an array. As a general category, they are called *reference objects*. We distinguished the types int and double and bool, where the actual value of the data is stored in the space for a variable of the type. They are *value types*. A struct is also a value type. In practice this is efficient for small objects of a fixed size. We made Rational a class because you have already seen the class construct with static entries, and classes are more generally useful. In fact being a struct would be a good choice for Rational, since it only contains two integers. Its size is no more than one double.

The behavior of a Rational is the same either way, because it is immutable. If we allowed mutating methods, then a class version and a struct version would not behave the same way, due to the fact the reference types can have aliases, and value types cannot.

There are some more complicated situations where there are further distinctions between classes and structs, but we shall not concern ourselves with those fine advanced points in this book.

13.6 Defining Operators (Optional)

13.6.1 Operator Overloading

The Rational class is a fine example of a useful utility class. Still, to an experienced user, it has a striking deficiency: A Rational is a *number* and we are used to doing arithmetic with standard operators. We would like to replace the mouthful frac1.Multiply(frac2) by our common symbolism for multiplication, frac1*frac2. This can be coded in C#, using *operator overloading* to give new meanings to the operator *. The C# syntax is illustrated in the variant of the Rational class in rational_ops_stub/rational.cs. This class also contains code discussed in the next section, *Casts in User-Defined Classes*. Here are operator overload declarations for * and others:

```
/// * binary multiplication operator
public static Rational operator *(Rational f1, Rational f2)
{
    return new Rational(f1.num*f2.num, f1.denom*f2.denom);
}

/// – unary negation operator
public static Rational operator – (Rational r)
{
    return new Rational(-r.num, r.denom);
}

/// binary == operator
public static Boolean operator ==(Rational f1, Rational f2)
{
    return f1.num == f2.num && f1.denom == f2.denom;
}

/// binary != operator
public static Boolean operator !=(Rational f1, Rational f2)
{
    return f1.num != f2.num || f1.denom != f2.denom;
} // or extra call, but clearly consistent: return !(f1 == f2)
```

All operator overload headings have the special form

```
public static returnType operator opSymbol ( parameters
)
```

Here opSymbol can be any arithmetic or comparison operator, or some other operators that we have not discussed. So something like `operator *` or `operator –` replaces the method name. Binary operations like multiplication require two operands, and hence the method has two parameters. The method computes and returns the named return type in the normal fashion. In general at least one of the parameters must be of the type of the class being defined.

(We could have directly defined four further overloads of `*`, with the first or second parameter being an `int` or a `double`, but we will avoid that by also adding methods to provide implicit *Casts in User-Defined Classes*.)

The – symbol is special, since it can be used either as a unary operator for negation, or as a binary operator for subtraction. Since we include only one parameter

above, we are defining the unary version.

The operator does not need to produce a result of the same type. We included ==
and != as examples (returning a Boolean).

(These methods do not cause compiler errors, but warnings are generated: We
have not added further more advanced overrides of Equals and HashCode meth-
ods, that are ideally in sync with the meaning of ==. You should see a discussion
of these methods in a data structures course, like Loyola's Comp 271.)

The class is a stub, and *Operator Overloading Exercise* invites you to add further
operator overloads.

Precedence: Note that the operator overloading method definitions include noth-
ing about *Precedence of Operators*. That is because the precedence of operators
is fixed across the whole language. Unary – has higher precedence than * ... no
matter what the types involved.

An example testing class also uses the new casting syntax of next section:

13.6.2 Casts in User-Defined Classes

We have discussed casts before. We know that an int can also be represented as
a double with an integer value, and the cast from int to double is done im-
plicitly when needed: An expression like 3.2 * 2 is processed by the compiler,
implicitly casting the 2 to double 2.0, and then doing a double multiplication.
The same idea makes sense with an int n and a Rational f. We only defined
the operator overload * for two Rationals, so in our code so far, f * n does not
make sense. Mathematically an integer is rational, so mathematically, it should
make sense. We bridge this difference by defining an implicit cast of an int to a
Rational, so the compiler will take f * n and see the need to implicitly cast n to
a Rational. The definition below will also allow explicit casts if you choose, like
f * (Rational) n:

```
/// Code to cast an int to a Rational implicitly when needed.
public static implicit operator Rational(int n)
{
   return new Rational(n, 1);
}
```

Again it is the heading that takes a special form, starting with public static

implicit operator followed by the type being cast to, like Rational, while the parameter is the starting type, like int. This is not like a regular method with its return type and method name. Here it looks something like a constructor with a type in place of a method name, but a constructor would not start with static implicit operator!

Now consider a double d and a Rational f. We would like to allow an expression like d * f. Again, the operator overload for * does not allow this directly, so consider implicit casts: Since a double is only an approximation, in general, it would not be wise to implicitly convert a double to a Rational, but it does make sense to approximate a Rational by a double before use with a double:

```
/// Code to cast to a double implicitly when needed.
public static implicit operator double(Rational f)
{
    return (double)f.num/f.denom;
}
```

The general format of such an implicit cast in a user-defined class is:

> public static implicit operator *resultType* (*source-Type paramName*)

One of the two types should be the type of the containing class. We have illustrated both combinations.

Finally, you need to be *very careful where you declare implicit casts*, to make sure you are not being overly general, and maybe allowing trouble in a form that may be very hard to debug: It is much harder to foresee and trace implicit actions than explicit actions. You are *safe*, but more *verbose*, if you *only allow explicit* casts. For example, we have already seen these required for a cast from double to int. To only allow an explicit cast with your type, replace implicit by explicit in the cast method heading.

```
/// Code to cast to a decimal with an explicit cast.
public static explicit operator decimal(Rational f)
{
    return (decimal)f.num/f.denom;
}
```

The decimal type: Though we have not used the decimal type before, we use it here for contrast to illustrate a cast from Rational to decimal that can only be used explicitly, as in (decimal) f:

Example rational_ops_stub/test_ops.cs tests all of the operator overloads and casts shown for a Rational. Look at the source code and run it. Note where overloaded operators are used and where implicit and explicit casts to or from a Rational are used.

The example also illustrates a special feature of the `decimal` type. While a `double` is encoded with a power of 2, so 0.1 is *not* stored accurately, a `decimal` is encoded with a power of 10, so exact decimal values with up to 28 digits can be stored and manipulated. (This is important for *monetary calculations*, so a `decimal` literal has **m** for money appended, like `5.99m`, representing the mathematical quantity 5.99 *exactly*.)

Operator Overloading Exercise

The classes discussed above from example project rational_ops_stub are incomplete. Add the overloaded binary operators /, +, -, <, >, <= and >= to the Rational class, and extend the TestOps class to test them.

13.7 Chapter Review Questions

1. Where in a class are instance variables declared?

2. For most instance variables, what is the modifier used that does not appear at the beginning of a local variable declaration?

3. What is the lifetime of an instance variable: When does it come into existence, and how long does it last?

4. Why do we generally make an instance variable `private`?

5. In what code can an instance variable be seen and used?

6. Must instance variables and methods always be preceded by an explicit object reference and `.`?

7. Can we refer to an instance variable in a part of the code where there is no current object?

8. In what kind of method in a class definition are instance variables never accessible?

9. What is the purpose of a constructor?

10. How is the heading of a constructor different from a regular method?

11. How are parameters to a constructor generally used?

12. If you do not explicitly assign a value to an instance variable in a constructor, does the instance variable have a value?

13. If we want users to be able to see the value of a private instance variable from outside of the class, how do we do it?

14. What is the general name of the category of methods that return instance state values?

15. Instance variables are usually visible from inside instance methods for the class. What is the exception? In the exceptional case, what is the workaround to allow access to the instance variable?

16. Sometimes you need to refer explicitly to the current object. How do you do it?

17. Sometimes you want to let users outside the class modify the value of a private instance variable. How do you do it?

18. What is the general name of the category of public methods whose sole purpose is to set a part of instance state to a new specified value?

19. If a class has one or more setter methods, is the object type immutable?

20. What is the return type for a setter method?

21. If you want to set an instance variable in a method, should you declare that instance variable in the method?

22. A method with what signature allows you to control how the string concatenation operate (+) generates a string from the object?

23. If you write an override the `ToString` method in a class, should the method print the string? If not, what should it do with the resulting string?

24. What is `this`?

25. Can aliased objects cause problems when created for an immutable object? Mutable object?

---

26. In a class with instance methods you can always design the class so variables are instance variables and not local variables. When should you use local variables instead?

27. If an instance method has a formal parameter of the same type as the class being defined, can you refer to a private instance variable in the parameter object? May you change it? How do you distinguish an instance variable for the current object from the corresponding instance variable for the parameter object?

FOURTEEN

TESTING

Now that we have learned a bit about classes, we're going to use the same feature to support *unit testing*. Unit testing is a concept that will become part of just about everything you do in future programming-focused courses, so we want to make sure that you understand the idea and begin to make use of it in all of your work.

The notion of unit testing is straightforward in principle. When you write a program in general, the program comprises what are properly known as units of development. Each language has its own definition of what units are but most modern programming languages view the *class* concept as the core unit of testing. Once we have a class, we can test it and all of the parts associated with it, especially its methods.

We will be introducing parts of file rational_nunit/rational_unit_tests.cs.

14.1 Assertions

A key notion of testing is the ability to make a logical assertion about something that generally must hold *true* if the test is to pass.

Assertions are not a standard language feature in C#. Instead, there are a number of classes that provide functions for assertion handling. In the framework we are using for unit testing (NUnit), a class named Assert supports assertion testing.

In our tests, we make use of an assertion method, `Assert.IsTrue()` to determine whether an assertion is successful. If the variable or expression passed to this method is *false*, the assertion fails.

Here are some examples of assertions:

- `Assert.IsTrue(true)`: The assertion is trivially successful, because the boolean value `true` is true.

- `Assert.IsTrue(false)`: The assertion is not successful, because the boolean value `false` is not true!

- `Assert.IsFalse(false)`: This assertion is successful, because `false` is, of course, false.

- `Assert.IsTrue(5 > 0)`: Success

- `Assert.IsTrue(0 > 5)`: Failure

There are many available assertion methods. In our tests, we use `Assert.IsTrue()`, which works for everything we want to test. Other assertion methods do their magic rather similarly, because every assertion method ultimately must determine whether what is being tested is true or false.

14.2 Attributes

Besides assertions, a building block of testing (in C# and beyond) comes in the form of attributes. Attributes are an additional piece of information that can be attached to classes, variables, and methods in C#. There are two attributes of interest to us:

- [TestFixture]: This indicates that a class is being used for testing purposes.

- [Test]: This indicates that a method is one of the methods in a class being used for testing purposes.

Without these annotations, classes and methods will *not* be used for testing purposes. This allows a class to have some methods that are used for testing while other methods are ignored.

In the remainder of this section, we're going to take a look at the strategy for testing the Rational class. In general, your goal is to ensure that the entire class is tested. It is easier said than done. In later courses (Software Engineering) you would learn about strategies for *coverage* testing.

Our strategy will be as follows:

- Test the constructor and make sure the *representation* of the rational number is sound. If the constructor isn't initializing an instance properly, it is likely that little else in the class will work properly.

- Then test the rest of the class. Whenever possible, group the tests in some logical way. In the case of the Rational class, there are three general categories (and one rather special one): arithmetic operations, comparisons, and conversions. In addition, there is the parsing test, which ensures that we can convert strings representing fractions into properly initialized (and reduced) rational numbers.

Let's get started.

14.3 Testing the Constructor

```
1    [Test()]
2    public void ConstructorTest()
3    {
4        Rational r = new Rational(3, 5);
5        Assert.IsTrue(r.GetNumerator() == 3);
6        Assert.IsTrue(r.GetDenominator() == 5);
7        r = new Rational(3, -5);
8        Assert.IsTrue(r.GetNumerator() == -3);
9        Assert.IsTrue(r.GetDenominator() == 5);
10       r = new Rational(6, 10);
11       Assert.IsTrue(r.GetNumerator() == 3);
12       Assert.IsTrue(r.GetDenominator() == 5);
13       r = new Rational(125, 1);
14       Assert.IsTrue(r.GetNumerator() == 125);
15       Assert.IsTrue(r.GetDenominator() == 1);
16   }
```

Testing the constructor is fairly straightforward. We essentially test three basic cases:

- Test whether a basic rational number can be constructed. In the above, we test for 3/5, 3/-5, 6/10, and 125. Per the implementation of the Rational class (how we defined it), these should result in fractions with numerators of 3, -3, 3, and 12; and denominators of 5, 5, 5, and 1, respectively.

- As you can observe from the code, we perform basic assertion testing to ensure that the numerators and denominators are what we expect. For example:

```
    Assert.IsTrue(r.GetNumerator() == 3)
```

Tests whether the newly minted rational number, Rational(3, 5), actually has the expected numerator of 3.

- If we are able to get through the entire code of the `ConstructorTest()` method, our constructor test is a success. Otherwise, it is a failure.

We'll look at how to actually run our tests in a bit but let's continue taking a look at how the rest of our testing is done.

14.4 Testing Rational Comparisons

```
1        [Test()]
2       public void BasicComparisonTests() {
3           Rational r1 = new Rational(-3, 6);
4           Rational r2 = new Rational(2, 4);
5           Rational r3 = new Rational(1, 2);
6           Assert.IsTrue(r1.CompareTo(r2) < 0);
7           Assert.IsTrue(r2.CompareTo(r1) > 0);
8           Assert.IsTrue(r2.CompareTo(r3) == 0);
9        }
```

It is pretty well established by now that the ability to compare is of fundamental importance whenever we are talking about data. Everything we do, especially when it comes to searching (finding a value) and sorting (putting values in order) depends on comparison.

In this test, we construct a few Rational instances (r1, r2, and r3) and perform at least one test for each of the essential operators (>, <, and =). Recall from our earlier discussion of the Rational class that the CompareTo method return a value < 0 when one Rational is *less than* another. It returns a number > 0 for *greater than*, and == 0 for *equal to*.

If any one of these comparisons fails, this means that we cannot rely on the ability to compare Rational numbers. This will likely prevent other tests from working,

such as the arithmetic tests, which rely on the ability to test whether a *computed result* matches an *expected result* (e.g. 1/4 + 2/4 == 3/4).

14.5 Testing Rational Arithmetic

```
1    [Test()]
2    public void BasicArithmeticTests() {
3        Rational r, r1, r2;
4        r1 = new Rational(47, 64);
5        r2 = new Rational(-11, 64);
6
7        r = r1.Add(r2);
8        Assert.IsTrue(r.CompareTo(new Rational(36, 64)) == 0);
9
10       r = r1.Subtract(r2);
11       Assert.IsTrue(r.CompareTo(new Rational(58, 64)) == 0);
12
13       r = r1.Multiply(r2);
14       Assert.IsTrue(r.CompareTo(new Rational(47 * -11, 64 * 64)) == 0);
15
16       r = r1.Divide(r2);
17       Assert.IsTrue(r.CompareTo(new Rational(47, -11)) == 0);
18
19       r = r1.Reciprocal();
20       Assert.IsTrue(r.CompareTo(new Rational(64, 47)) == 0);
21
22       r = r1.Negate();
23       Assert.IsTrue(r.CompareTo(new Rational(-47, 64)) == 0);
24   }
```

Testing of arithmetic is a fairly straightforward idea. For all of these tests, we create a couple of rational numbers (47/64 and -11/64) and then call the various methods to perform addition, subtraction, multiplication, division, reciprocal, and negation.

The key to testing arithmetic successfully in the case of a Rational number is to know know what the result *should be*. As a concrete example, the result of adding these two rational numbers should be 36/64. So the testing strategy is to use the `Add()` method to add the two rational numbers and then test whether the result of the addition is equal to the *known* answer of 36/64.

As you can observe by looking at the code, the magic occurs by checking whether the *computed* result matches the *constructed* result:

```
Assert.IsTrue(r.CompareTo(new Rational(36, 64)) == 0);
```

Because we have *separately* tested the constructor and comparison methods, we can assume that it is ok to rely upon comparison methods as part of this arithmetic test.

And it is in this example where we begin to see the *art of testing*. You can write tests that assume that other tests of features you are using have *already passed*. In the event that your assumption is wrong, you'd be able to know that this is the case, because all of the tests you assumed to pass would not have passed.

Again, to be clear, the arithmetic tests we have done here *assume* that we can rely on the constructor and the comparison operation to determine equality of two rational numbers. It is entirely possible that this is not true, so we'll be able to determine this when examining the test output (we'd see that not only the arithmetic test fails but possibly the constructor and/or comparison tests as well).

The remaining tests are fairly straightforward. We'll more or less present them as is with minimal explanation as they are in many ways variations on the theme.

14.6 Testing Rational Conversions (to other types)

```
1      [Test()]
2      public void ConversionTests() {
3          Rational r1 = new Rational(3, 6);
4          Rational r2 = new Rational(-3, 6);
5          Rational r3 = new Rational(10, -2);
6
7          Assert.IsTrue(r1.ToDecimal() == 0.5m);
8          Assert.IsTrue(r2.ToDecimal() == -0.5m);
9          Assert.IsTrue(r1.ToDouble() == 0.5); //.5 is stored exactly
10         Assert.IsTrue(r2.ToDouble() == -0.5);
11         Assert.IsTrue(r2.ToString() == "-1/2");
12         Assert.IsTrue("" + r3 == "-5");  //implicit use of ToString
13
14     }
```

In this test, we want to make sure that Rational objects can be converted to floating point and decimal types (the built-in types of the C# language).

For example, Rational(3/6) is 1/2, which is 0.5 (both in its floating-point and decimal representations.

14.7 Testing the Parsing Feature

```csharp
[Test()]
public void ParseTest()
{
    Rational r;
    r = Rational.Parse("-12/30");
    Assert.IsTrue(r.CompareTo(new Rational(-12, 30)) == 0);
    r = Rational.Parse("123");
    Assert.IsTrue(r.CompareTo(new Rational(123, 1)) == 0);
    r = Rational.Parse("1.125");
    Assert.IsTrue(r.CompareTo(new Rational(9, 8)) == 0);
    Assert.IsTrue(r.ToString().Equals("9/8"));
}
```

The parsing test tests whether we can convert the string representation of a rational number into an actual (reduced) rational number. We test three general cases:

- The ability to take a fraction and convert it into a rational number. This fraction may or may not have a "-" sign in it. For example -12/30 should be equivalent to constructing a Rational(-12, 30).

- The ability to take a whole number and get a proper Rational, e.g. 123 is equal to Rational(123)

- The ability to take a textual representation (1.125) and get a proper Rational(9, 8) representation. In this case, we are also getting an extra test to ensure the result is reduced.

14.8 Running the NUnit Tests

1. In Xamarin Studio, select the rational_nunit project.

2. In the main Xamarin Studio menu click "Run" and select "Run Unit Tests"

A test pad should appear and show something like
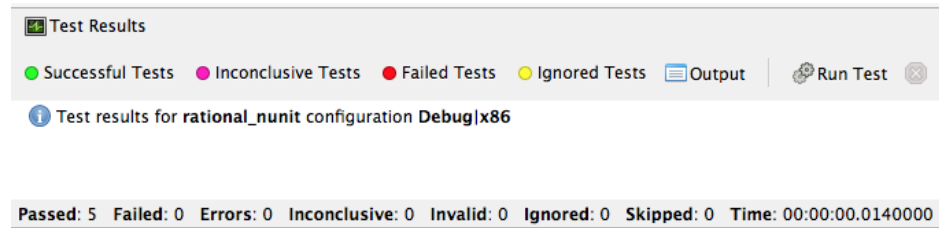
This likely just shows the overall results in the summary line at the bottom. You can show details by clicking on one or more of headings at the top of the pad. In particular, if you click Successful Test, Failed Tests, and Output (and likely drag the top of the pad to make it large enough to see everything), you should see something like
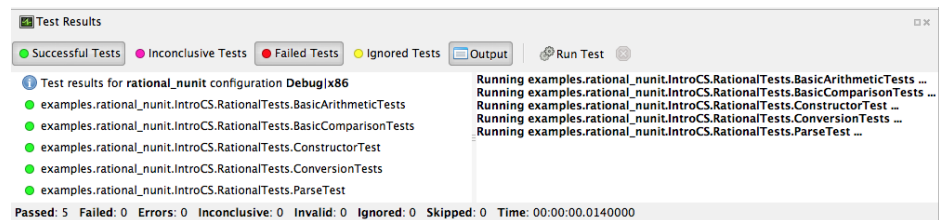
As you can see in the above displays, all of the tests in `RationalTests` get executed, and they all pass. There are no failed tests to see, but that part would be the most important details if any were there! The output just details the sequence of execution.

Xamarin Studio remembers the headings selected, so next time you run tests, the same details will show.

The testing file is using NUnit.Framework. There is a little more to this. If you edit the references, selecting All sources, and entering nunit in the search box, you see
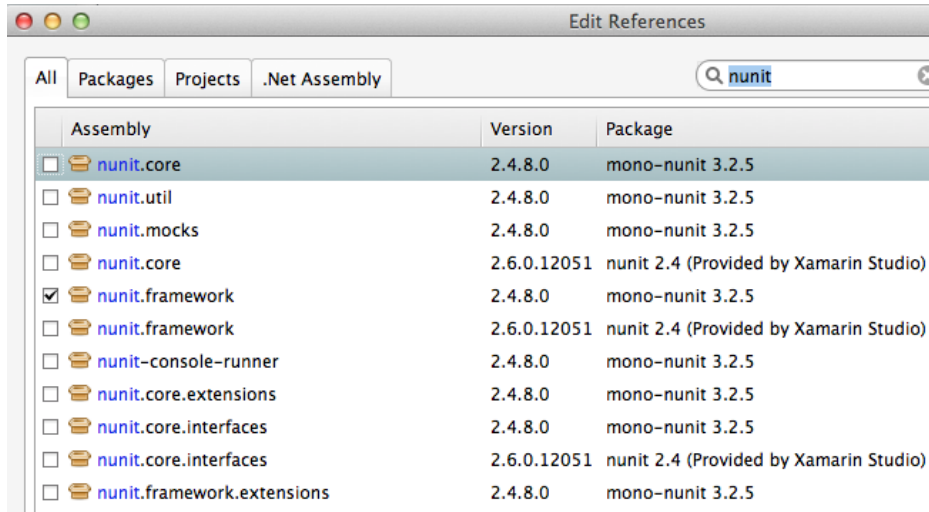
Note there are more than one version of nunit.framework. On a Mac, the selected version worked directly, but the logical looking one, provided by Xamarin, did *not* work directly. Either worked if we select the context menu in the Solutions pad for the nunit.framework version added to the references, and make it look like

with the "Require Specific Version" item toggled so it is *not* checked.

We waited until now to discuss unit testing, because the test classes are coded with *instance* methods, unlike the static methods that we started out with.

A test can also call `Main` of a program, with specified parameters that would normally come from the command line. See the small project cmdline_to_file.

14.8.1 String Replace NUnit Test Exercise

If you completed the program from the stub string_manip_stub/string_manip.cs, then add a testing class using NUnit that tests `ReplaceFirst` with the same parameters as used in the original file's `Main`. If you did the elaboration of `ReplaceFirst` that just returns the original string when the target is not found, then add tests for that, too. Remember the necessary library reference and try it out.

14.8.2 Grade File NUnit Test Exercise

Add a testing class using NUnit to *Homework: Grade File*. Remember the necessary library reference. Sample data files and files for testing the results are included in the project. Just test using command line parameters (so there is no Console input). Test with both comp170 and comp150.

INTERFACES

15.1 Rationals Revisited

C# has a built-in method to sort a List. `List` is a generic type, however, so how does C# know how to do comparisons for all different types? Is this specially programmed in for built-in types, or can it be extended to user-defined types?

In fact it can be extended to user defined types, such as our Rational. To sort objects, you only need to be able to do one thing: indicate which object comes before another. We can do that. The `CompareTo` method already does that. If Rational r1 is less than Rational r2, then

```
r1.CompareTo(r2) < 0
```

The single CompareTo method is very versatile: Just by varying the comparison with 0, you vary the corresponding comparison of Rationals:

    `r1.CompareTo(r2) < 0` means r1 < r2
    `r1.CompareTo(r2) <= 0` means r1 <= r2
    `r1.CompareTo(r2) > 0` means r1 > r2
    `r1.CompareTo(r2) >= 0` means r1 >= r2
    `r1.CompareTo(r2) == 0` means r1 is equal to r2
    `r1.CompareTo(r2) != 0` means r1 is not equal to r2

None of the other methods for Rationals make any difference for sorting: Just this one method is needed. Of course the comparison of strings or doubles are done with totally different implementations, but they have methods with the same name,

CompareTo, and with the same abstract meaning. Still C# is strongly typed and we are talking about totally different types.

An *interface* allows us to group diverse classes under one interface type. An interface just focuses on the commonality of behavior in one or more methods among the different classes. Interface types, like classes can also be generic. For sorting we are only concerned with one method, CompareTo. We want it to be able to compare to another object of the same type.

C# defines a generic interface IComparable<T>. A type T can satisfy this interface if if has a public instance method with signature:

```
public int CompareTo(T other);
```

There is one more step before we can use a library method to sort: Although the signature shown above for CompareTo is the one that C# requires to be able to satisfy the Icomparable<T> interface, it does not automatically assume that this is your *intention*. You must explicitly say that you *want* your class to be considered to satisfy this interface. For instance for Rational, we need to change the class heading to:

```
public class Rational : IComparable<Rational>
```

In general one or more interface names can be listed after the class name and a colon, and before the opening brace of the class body. This particular interface is defined in System.Collections.Generic, so we need to be using that namespace.

The project interfaces has the modified rational.cs and test_rational_sort.cs to test this with a list of Rationals:

```csharp
using System;
using System.Collections.Generic;

namespace IntroCS
{
/// Use IComparable<Rational> interface to sort Rationals.
   class TestRationalSort
   {
      public static void Main(string[] args)
      {
         List<Rational> nums = new List<Rational>();
         nums.Add(new Rational(1, 2));
         nums.Add(new Rational(11, 3));
```

```
        nums.Add(new Rational(-1, 10));
        nums.Add(new Rational(2, 5));
        nums.Add(new Rational(2, 3));
        nums.Add(new Rational(1, 3));
        Console.WriteLine("Before sorting: " + ListString(nums));
        nums.Sort();
        Console.WriteLine("After sorting:  " + ListString(nums));
    }

    public static string ListString(List<Rational> list)
    {
        string s = "";
        foreach (Rational r in list) {
            s += r + " ";
        }
        return s;
    }
  }
}
```

which prints:

```
Before sorting: 1/2 11/3 -1/10 2/5 2/3 1/3
After sorting:  -1/10 1/3 2/5 1/2 2/3 11/3
```

Interfaces are very handy for dealing with the *common* abstract behavior of different objects and different underlying class, but if an object is declared to be of interface type the compiler no longer sees the attributes that are *not* common to the interface. A silly example, legal as far as it goes:

```
IComparable<Rational> r1 = new Rational(2,5),
                      r2 = new Rational(1, 2);
Console.WriteLine(r1.CompareTo(r2) > 0); // prints false
```

The declarations are legal because a Rational does have interface type `IComparable<Rational>`. The use of `CompareTo` is legal because that is the one method that this interface type guarantees.

However, if we add this extra line:

```
Console.WriteLine(r1.Multiply(r2)); // compiler error!
```

Even though r1 and r2 are actually Rational underneath, where the Multiply

---

method is legal, their declaration as only their interface type *hides* this extra functionality from the compiler. [1]

15.1.1 Interface Syntax Examples

In the previous section we showed a realistic application of an existing interface. It was in a fairly sophisticated class with a lot of other things included. Now you can look at examples that are designed to highlight syntax for interfaces without distractions. They are very simple and artificial.

There is a comment at the bottom of each file explaining the new features introduced. Like all our other examples, they compile and run as given. After running an example, see if the notes include instructions to delete or comment out parts or uncomment lines. If so, follow instructions and try to compile again. Check that the change causes a compiler error.

Look through and process these examples in order:

interface_syntax1/interface_demo1.cs, interface_syntax2/interface_demo2.cs, interface_syntax3/interface_demo3.cs, interface_syntax4/interface_demo4.cs, interface_syntax5/interface_demo5.cs, interface_syntax6/interface_demo6.cs, interface_syntax7/interface_demo7.cs, interface_syntax8/interface_demo8.cs

After looking at those simple bare examples illustrating the syntax, go on to the next section *Csproject Revisited*, where a useful, more sophisticated user-defined interface is introduced....

Example Class Sorting - Worked Exercise

In *More Getters and Setters* we introduced example example_class/example_class.cs.

Elaborate the code for the `Example` class, adding a `CompareTo` method. The rules for comparison are:

1. An object with a larger `n` value is considered larger.

---

[1] It is possible to deal with the actual underlying class type, but this gets more complicated. It is better discussed in a course that more fully explores *inheritance*.

2. If the `n` values are the same, then the object with the larger `d` value is considered larger.

3. If they completely match, then they are equal, of course.

Modify the `Main` driver to merely test sorting: create and sort a list of `Example` elements. Show the before and after sequence in the list.

You can compare your solution to ours: example_class2/example_class2.cs.

15.2 Csproject Revisited

The cs_project1 project skeleton was set up with the different commands in different classes, keeping related things together.

On the other hand they had high level structure in common. Similar names were consciously used for methods:

• Each command needed to Execute

• Each command needed to have Help for the user.

The corresponding names made it somewhat easier to follow the part of the Game constructor with the additions to the `helpDetails` Dictionary. Also there is repetitive logic in the crucial `proccessCommand` method.

In a game with more possible commands, the code would only get more repetitious! You would like to think of having a loop to replace the repetitious code.

A major use of a C# interface will allow this all to work in neat loops. For the first time we define our own interface, and use that interface as a type in a declaration.

While we are at this we can refactor our code further: classes that give a response to a command all obviously have their Execute and Help methods. They also have a command word to call them. We can further encapsulate all data for the response by having the classes themselves be able to announce the command that calls them. We add a string property CommandWord to each of them.

We will add an extra convenience feature of C# here. Thus far we have used private instance variables and public getter methods. We can use a public instance variable declaration with a similar effect as in:

```
public string CommandName {get; private set;}
```

The extra syntax in braces says that users in a another class may freely *get* (read)
the variable, but setting the variable is still *private*: it may only be done inside
the class. This is more concise than using a getter method: No getter needs to
be declared; referencing the data is shorter too, since it is a property, no method
parentheses are needed.

Note the unusual syntax: the declaration does not end with a semicolon. The
only semicolons are inside the braces. You will not be required to code with this
notation, but it surely is neater than using a getter method!

Now we can define our own interface taking all of these common features together.
Since each is a response to a command, we will call our interface Response:

```
namespace IntroCS
{
/// Object that responds to a command
   public interface Response
   {
      /// Execute cmd.
      /// Return true if the game is over; false otherwise
      bool Execute(Command cmd);

      /// Return a Help string for the command
      string Help();

      string CommandName {
         get;
      }
   }
}
```

Things to note:

- The heading has the reserved word `interface` instead of class.

- All the common method headings and the property declaration are listed.

- See what is missing! In place of each method body is just a semicolon.

- Everything in an interface is public. The part of the property about private
  access is merely omitted.

We are going to need a collection if we want to simplify the code with loops. We could use code like the following, assuming we already declared the objects helper, goer, and quitter:

```
Response[] resp = {helper, goer, quitter};
```

See how we use Response as a declaration type! Each of the objects in the declaration list *is* in fact a Response.

Now that we can process with this collection and `foreach` loops, we do not need the object names we gave at all: We can just put new objects in the initialization sequence!

Now that we can think of these different objects as being of the same type, we can see the processCommand logic, with its repetitive `if` statement syntax is just trying to match a command word with the proper Response, so a Dictionary is what makes sense!

In fact all the logic for combining the various Responses is now moved into CommandMapper, and the the CommandMapper constructor creates the Dictionary used to look up the Response that goes with each command word. Here is the whole code for ResponseMapper, taking advantage of the Dictionary in other methods, too.

```csharp
using System;
using System.Collections.Generic;
namespace IntroCS
{

/// Map commands names to commands.
   public class CommandMapper
   {
      public string AllCommands {
         get;
         private set;
      }
      private Dictionary<string, Response> responses; //responses to commands

      /// Initialize the command response mapping
      ///  game The game being played.
      public CommandMapper(Game game)
      {
         responses = new Dictionary<string, Response>();
```

```csharp
        Response[] resp = {
            new Quitter(),
            new Goer(game),
            new Helper(responses, this)
            // add new Responses here!
        };
        AllCommands = "";
        foreach (Response r in resp) {
            responses[r.CommandName] = r;
            AllCommands += r.CommandName + " ";
        }
    }

    /// Check whether aString is a valid command word.
    /// Return true if it is, false if it isn't.
    public bool isCommand(string aString)
    {
        return responses.ContainsKey(aString);
    }

    /// Return the command associated with a command word.
    ///  cmdWord The command word.
    /// Return the Response for the command.
    public Response getResponse(string cmdWord)
    {
        return responses[cmdWord];
    }
}
}
```

There is even more to recommend this setup: The old setup had references in
multiple places to various details about the collection of Responses. That made
it harder to follow and definitely harder to update if you want to add a new com-
mand. Now after writing the new class to respond to a new command, the *only*
thing you need to do is add a new instance of that class to the array initializer in
the CommandMapper constructor!

Note: Interfaces never say anything about constructors. Classes satisfying an
interface can have totally different constructors.

In the code above, `Quitter` has a very simple constructor. (A `Quitter` has no

---

individualized data, and only has an instance because an interface can only work with an instance, not with a static class.) On the other hand the `Goer` and `Helper` objects need to reference more data to work properly, so they have parameters.

The revised Xamarin Studio project is csproject_stub (no 1 this time).

See how the Game class is simplified, too.

Talking about adding commands - these classes could be the basis of a game project for a small group. Have any ideas? See *Group Project*.

There are further examples of defining and using Interfaces in the starting code for the exercises at the end of the next section.

15.2.1 Cohesion, Coupling, and Separation of Concerns

This section is motivated by the revisions to the project, not specifically about Interfaces, though good use of them helps.

There are three important ideas in organizing your code into classes and methods:

Cohesion of code is how focused a portion of code is on a unified purpose. This applies to both individual methods and to a class. The higher the cohesion, the easier it is for you to understand a method or a class. Also a cohesive method is easy to reuse in combination with other cohesive methods. A method that does several things is not useful to you if you only want to do one of the things later.

Separation of concerns allows most things related to a class to take place in the class where they are easy to track, separated from other classes. Data most used by one class should probably reside in that class. Cohesion is related to separation of concerns: The data and methods most used by one class should probably reside in that class, so you do not need to go looking elsewhere for important parts. Also, if you need to change something, where concerns are separated into cohesive classes, with the data and logic in one place, it is easier to see what to change, and the changes are likely to be able to be kept internal to the class, affecting only its internal implementation, not its public interface.

Some methods are totally related to the connection between classes, and there may not be a clear candidate for a class to maximize the separation of concerns. One thing to look at is the number of references to different classes. It is likely that the most referred to class is the one where the method should reside.

Coupling is the connections between classes. If there were no connections to a class, no public interface, it would be useless except all by itself. The must be some coupling between classes, where one class uses another, but with increased cohesion and strong separation of concerns you are likely to be able to have looser coupling. Limiting coupling makes it easier to follow your code. There is less jumping around. More important, it is easier to modify the code. There will be less interfacing between classes, so if you need to change the public interface of a class, there are fewer places in other classes that need to be changed to keep in sync.

Aim for strong cohesion, clear separation of concerns, and loose coupling. Together they make your code clearer, easier to modify, and easier to debug.

IGame Interface Exercise

On a much smaller scale than the project, this exercise offers you experience writing classes implementing and using an interface.

1. Copy project stub igame_stub to your own project, and modify it as discussed below.

2. Look at the IGame interface in i_game.cs. Then look at addition_game.cs, that implements the interface. See how a new `AdditionGame` can be added to list of `IGame`'s. Run `play_games.cs`. Randomly choosing a game when there is only one to choose from is pretty silly, but it gives you a start on a more elaborate list of games. The `PopRandom` method is a good general model for choosing, removing, and returning a random element.

3. Write several very simple classes implementing the IGame interface, and modify Main in play_games.cs to create and add a new game of each type. (Test adding one at a time.)

   One such game to create with little more work would be a variation on instance based Guessing Game instance_version/guess_game.cs. You need to make slight modifications. You could make Play return the opposite of the number of guesses, so more guesses does generate a worse score. Note that you could not use the original static game version: Only objects can satisfy an interface.

Bisection With Function Interface Exercise

See bisection_method/bisection_method.cs. Identify the interface. See how two new classes satisfy the interface, while the rest of the code in these classes is quite different: One has an instance variable and an explicit constructor, while the other does not. See that the `Bisection` function now has a parameter for an object containing the mathematical function to use for root finding: `Bisection` is now a generally useful method, not just tied to one hard-coded function that might have a root.

Add a new class satisfying the `Function` interface; add a test in `Main`. Try a function with multiple roots in the original interval and see what happens. Then, using *distinct* intervals in different tests, find different roots of the same function.

15.3 Chapter Review Questions

For these review questions, assume `Foo` is an interface type that you have access to.

1. Suppose you are writing a new class that you want to satisfy an existing interface:

    (a) Is matching the signatures required by the interface enough?

    (b) What else do you need in the new class heading?

2. With interface type `Foo` can you:

    (a) Declare a formal parameter of type `Foo`?

    (b) Declare a list of type `List<Foo>`?

    (c) Declare an array of type `Foo[]`?

    (d) Create a new object with `new Foo()`?

3. If you have an array of type `Foo[]`, then each element is an object which has an underlying class type. Must each element have the *same* underlying class type?

4. An interface type `Foo` has what heading (simplest)?

5. An interface type definition includes the signatures of all the methods for that interface. What is included after each signature?

6. May an interface definition include a constructor?

7. May a class satisfying an interface have further methods not listed in the interface?

8. Identify the legal interface declarations, and say what is wrong with any other code.

```csharp
public interface A
{
    void f(List<int> L)
    {
        L.Add(7);
    }
}
```

```csharp
public interface B
{
    void f(List<int> L);
}
```

```csharp
public interface C
{
    void f(List<int> L);
    int g(int x);
}
```

```csharp
public interface D
{
    D(int a);
    void f(List<int> L);
}
```

```csharp
public class E
{
    int g(int x);
}
```

9. Which of these class definitions make the class implement the legal interface C above? Explain the problem with any that do not.

```
public class CA
{
    public void f(List<int> L)
    {
        L.Add(7);
    }

    public int g(int x)
    {
        return x*x;
    }
}
```

```
//  Same as CA, except with heading
public class CB : C
// ...
```

```
public class CC : C
{
    public void f(List<int> L)
    {
        L.Add(7);
    }

    public void g(int x)
    {
        Console.WriteLine(x*x);
    }
}
```

```
public class CD : C
{
    public void f(List<int> L)
    {
        L.Add(7);
    }
}
```

```
public class CE : C
{
    private int a;
```

```csharp
        public CE(int a)
        {
            this.a = a;
        }

        public void f(List<int> L)
        {
            L.Add(a);
        }

        public void h(int x)
        {
            Console.WriteLine(a*x*x);
        }

        public int g(int x)
        {
            return x*a;
        }
    }
```

SIXTEEN

RECURSION

We are looking forward to a data structures course. Recursion is an important topic. It is not a topic we require in the introductory course.

More later.

DATA STRUCTURES

We have discussed some basic data structures that are available in C#: Array, List, Set, Dictionary. There are many more with many implementations. Further study leads you into a data structures course.

We may add some optional, forward looking material here.

APPENDIX

18.1 Development Tools

18.1.1 About Software Development Kits (SDKs)

A software development kit (SDK) is a set of tools for developing in a particular programming language (in our class, C#). Developing in a language means everything from compiling to running and (when things go wrong) to debugging programs.

The Microsoft SDK is the proprietary implementation of .Net. It runs only on Windows and is the primary development framework for all things Microsoft.

The Mono Project SDK <http://mono-project.com> is the free/open source equivalent implementation of the Microsoft SDK. It runs on all major platforms (including Windows) and is needed in situations where you want to develop .Net applications on non-Windows platforms.

As an interesting aside, the company whose developers lead the work on the Mono SDK are working on commercial tools that allow you to develop/run applications written in .Net on Apple iOS and Android mobile devices (phones and tablets).

18.1.2 Editing and Building Tools

Early programs were written with rudimentary text editors, more primitive than Windows Notepad. Gradually tools got better. Now there are editors that are

highly optimized for editing code.

After code is edited, it has to be converted into an executable program. That may involve several files and libraries and other dependencies. Streamlining and automating this process was a big deal. There are a variety of building tools that can be used with, or built into an SDK: make, ant, and now NAnt for .net.

Many developers use an a la carte approach, using their favorite editor along with their favorite building tool.

18.1.3 About Integrated Development Environments (IDE)

There are also all-in-one tools that combine an editor and build tools. These are also used by many developers.

There are two major IDEs for .Net development, which we explain briefly below:

- Visual Studio is the Microsoft IDE that interfaces directly to the Microsoft SDK.

- Xamarin Studio is the free/open source IDE for developing applications using the Mono SDK on Windows and all other platforms (in particular, Linux and OS X). The project started as MonoDevelop. Now Xamarin is both a major contributor to the code and has commercial versions for iOS development. *The name on the software is now* **Xamarin Studio**, though you may see references to MonoDevelop instead.

In addition, there is another Windows-specific IDE, SharpDevelop, that inspired the creation of Xamarin Studio. It is still actively maintained and provides a somewhat "lighter weight" alternative to Visual Studio for Windows users. Like Xamarin Studio, it is aimed at developers who would prefer a more free/open source "friendly" version.

18.1.4 Our Approach

In the interest of providing a consistent experience for our students who use various operating systems on their own machines, we will be using the multi-platform Mono (the SDK).

We find the IDE Xamarin Studio convenient to integrate everything for a beginner, and it is a powerful tool at a more advanced level. Hence we start off introducing

and using Xamarin Studio. Later we will look at some of the underlying tools that are obscured by the use of Xamarin Studio.

Mono has an extra advantage in the tool csharp, for immediate testing of small snippets of code. We will use it extensively as we introduce bits of syntax.

As there is significant evolution of both the Microsoft and Mono *toolchains*–a fancy word we want you to know and a more elegant way of saying SDK–we'll issue updates to this book.

Everything is free, but there are a number of steps. Follow them carefully.

18.1.5 Installing Mono and Xamarin Studios

Because the Mono Project web page is known to change frequently, these instructions are designed to be as generic as possible. If you have any questions, you should contact the instructors immediately or seek tutoring help.

18.1.6 OS X

> **Warning:** Xamarin Studio needs at least version 10.8 of OSX. If you have an older version, you can upgrade the operating system, or possibly use an older version of Xamarin Studio. In that case, ask for help.

There are two downloads to get and install in order. Mono first:

1. Go to <http://mono-project.com>.

2. Look for the Mono downloads link. Link on OS-X. You want to get the latest *stable* version of Mono for OS X. For this class, you need version 2.10 or later, though preferably 3.2.4 or later. Choose the MRE version. It installs directly. Administrative privileges are required to run the installer, so if you do not know this information, please stop here.

   Do *not* download Xamarin Studio from this site. This version of Xamararin Studio bugs you with emails.

Here is how to do a quick sanity check of your Mono setup:

1. Go to Applications -> Utilities and launch the Terminal application, or quicker: enter terminal in Spotlight. (Terminal is how you get to a command-line shell in OS X.)

2. You'll see a prompt that looks like this `computername:folder user$`. This means that Terminal is ready for input.

3. Type `which csharp` and hit enter/return. You should see `/usr/bin/csharp` as output. `csharp` is the C# interpreter.

4. Type `which mcs` and hit enter/return. You should see `/usr/bin/mcs` as output. `mcs` is one of the interfaces to the C# compiler.

Xamarin Studio Installation - OSX

1. Make sure Mono is installed first.

2. Now go to http://monodevelop.com. **Note**: Do *not* use a version that is linked to the mono-project.com site. Getting the suggested open-source version from http://monodevelop.com should not lead to a prompt for your email address....

3. As with Mono, we need to look for the downloads link. You should download the *stable* version.

4. For OS X, the Xamarin Studio SDK is distributed as a DMG disk image. You'll need to download this image and double-click it. Open the image and run the installer. Administrative privileges are required to run the installer.

5. This time, you will see an App for Xamarin Studio, which you can drag and drop into the Applications folder.

6. If the preceding steps were successful, you can launch Xamarin Studio by double-clicking the icon in your Applications folder. (You won't know what to do with it yet, but at least you can verify that it launches correctly and then use Command-Q to exit.)

18.1.7 Windows

There are four packages, so this takes a while. Mono first:

Dr. Yacobellis has a video showing Windows installation. https://connect.luc.edu/p4hmzk2kbmt/ There may be further changes to the system.

1. Go to <http://mono-project.com>.

2. Look for the Mono downloads link. You want to get the latest *stable* version of Mono for Windows. For this class, you need version 2.10 or later, preferably 3.2.3 or later.

3. Choose the link: Mono for Windows, Gtk#, and XSP, and download the installation package

4. It is a self-extracting executable, so you will need to double click it to install. For Windows 7 users, you may need to check your taskbar to see whether the installer is being held up by Microsoft's enhanced security, UAM, that makes sure you really want to install something you downloaded from the internet.

Here is how to do a quick sanity check of your Mono setup:

Mono Command Prompt

1. Open the Windows Start Menu and type "mono" in the text field at the bottom. You should see a short list of places "mono" appears.

2. Click on the choice that says "Mono ... Command prompt". (This is probably faster than going to the Start Menu, finding the Mono folder, expanding it, and clicking on the Mono Command Prompt.)

If it comes up, you are all set for an initial installation check. This will be the first step later, when you want to run the handy csharp program or compile and run your own programs. When working, you can just leave this window open, saving it for later use, (or close and reopen later....)

Xamarin Studio Installation - Windows

1. Have Mono installed first.

2. Now go to http://monodevelop.com. **Note**: Do *not* use a version that is linked to the mono-project.com site. Getting the suggested open-source

version from http://monodevelop.com should not lead to a prompt for your email address....

3. As with Mono, we need to look for the downloads link, click on the Windows icon. You should click the link for the download of the requirements for the *stable* version. That should be at least numbered 4.2.2. **Do not install it yet.**

   Note however, that you will next install two support packages:

   - .Net Framework 4.0 first. The link takes you to a Microsoft download site. Do not click the top Download button - that gives you much more than you need. Further down in Popular download 01 is Microsoft .NET Framework 4 (Web Installer). Click on that and follow the default sequence.

   - GKT# The GKT@ download directly downloads the GKT installer. Again follow the default installation sequence.

   - install Xamarin Studio **last**. The Download link gets you the installer directly. Install it following the default steps.

4. If the preceding steps were successful, you can launch Xamarin Studio by double-clicking the icon on the Desktop or using the Start Menu. (You won't know what to do with it yet, but at least you can verify at it launches correctly and then close the window.)

18.1.8 Linux

We only provide instructions for Debian-based Linux distributions such as Ubuntu.

1. Using the command-line `apt-get` tool, you can install everything that you need using `apt-get install monodevelop`. This should be run as the **root** user (using the `sudo` command).

2. You can test the sanity of your setup by following the instructions under OS X.

Xamarin Studio releases on Linux tend to lag behind the official stable release.

This page, https://launchpad.net/~keks9n/+archive/monodevelop-latest, describes how to update your Xamarin Studio setup if it is not version 2.8 or later as we'll

need for this course.

We wish to stress that Linux is recommended for students who already have a bit of programming experience under their belts. It can take a significant amount of energy to get a Linux setup up and running and to tweak it to your liking. While it has gotten ever so much easier since the 1990s when it first appeared, we encourage you to set it up perhaps a bit later in the semester or consider running it using virtualization software (on Mac or Windows) such as VirtualBox or VMware.

18.2 Xamarin Studio

Several sections have given documentation for Xamarin Studio:

- *Xamarin Studio Installation - Windows*
- or *Xamarin Studio Installation - OSX*
- *Lab: Editing, Compiling, and Running with Xamarin Studio*
- *Indentation Help*
- *Running our Xamarin Studio Examples Solution*
- *Xamarin Studio Reminders and Fixes*
- *Library Projects in Xamarin Studio (Optional)*
- *Running the NUnit Tests*

18.3 Command Line Introduction

Sometimes we will be directing you to use a command window or terminal to compile and run C# programs. [1]

Reasons to use the command line:

- The command line precedes the graphical user interface (GUI) used in modern operating systems and provides a simpler interface for input and output that is very flexible and powerful for *knowledgeable* users.

---

[1] Thanks to Dr. Robert Yacobellis for elaborations to this section.

- Input comes from the keyboard as typed characters (no mouse processing). Commands are only processed once you press `Enter`.

- Output goes to the monitor as textual information (no window processing).

- In C# these input/output mechanisms are called Console processing.

- Input from and output to files is done in a very similar way, simplifying learning.

- Many software development organizations use command line processing to automate creating, compiling ("building"), and running or executing software programs.

  - Command line "scripts" can be created to automate routine tasks.

  - Command line scripts are similar to C# and other computer programs.

  - Serious software developers should be familiar with the command line.

The most direct way to access the command line (often called a *command shell*):

- On Windows, to have easy access to Mono tools, press the Windows key (lower left or lower right on the keyboard) and type Mono, then select the Mono Command Prompt and press Enter.

  Alternately, if you do *not need Mono tools for sure*, the general way to get a command window is to press the Windows key and R (lower or upper case) together, then type *cmd* and press the Enter or Return key – this brings up the basic command processing program, cmd.

- On a Mac just open a Terminal window – this is fine for the Mono SDK commands.

Mac OSX, Linux and other Unix variants work basically the same once you get to a terminal, so we will only distinguish Windows and Mac OS-X.

18.3.1 Navigating Directories

First make sure you are familiar with *Path Strings*.

In a command shell there is always a *current working directory*, usually shown in the prompt for the next command. When you open a Mono Command Prompt or Terminal window you will see a prompt that tells you what folder or directory the command shell has started in: If you directly open a terminal as in the previous section, in Windows this is typically C:\Windows\System32, and on a Mac it is typically /Users/*yourLogin*.

Particularly on windows, this is an annoying folder. There are several ways to get to a better location.

If you can get to a parent folder of a folder that you want in a Windows Explorer window (by right clicking on Start) or Mac Finder, there are shortcuts to opening a terminal with the current directory being one shown in the graphical window:

Windows

1. Note: this approach does *not* give your a Mono command prompt.

2. In Windows explorer navigate to the parent folder, showing the folder you want as a subfolder.

3. Hold down the shift key and *right* click on the desired folder. A popup menu appears.

4. Click on "Open a Command Window here".

Mac

1. In the Finder navigate to the parent folder, showing the folder you want as a subfolder.

2. Hold down the control key and click on the desired folder. A popup menu appears.

3. Click on the bottom item Services, to get a submenu.

4. In the submenu click on "New Terminal at Folder" (likely the bottom entry).

Files in the current working directory can to referred to by their simple names, e.g., *myfile.txt*. You can list all the files in the directory with the simple command `dir` (short for directory) in Windows or `ls` (short for list) on a Mac.

You need to refer to files not in the current directory via a relative or absolute path name.

After starting in one folder, you may well want to change the current folder without opening a new terminal window. This is particularly true if you start with a Windows Mono command prompt. You will see below that you can change the current directory with the *cd* command.

On a Mac, the file system is unified in one hierarchy. On Windows there may be several drives, and you need to start a path reference with a drive, like C:, if it is not the current drive.

When you run cmd or start a Mono command prompt in Windows, you are likely to want to get to your home directory (where the Mac users start automatically).

Windows 7 or 8 users enter the command below (substituting your login ID) to get to your home directory:

```
cd C:\Users\yourLoginId
```

The cd is short for "Change Directory", changing the current directory.

> **Warning:** Windows only: The cd command does not work the way you are likely to think about it on a Windows system with more than one drive (like C: and flash drive E: that you have plugged in). Windows remembers a separate current directory for each separate drive. It also *separately* remembers a *current drive*. *You do not change the current drive with the cd command.* The command to change the current drive is just the name of the drive with a colon after it. For example the command
>
> ```
> E:
> ```
>
> sets the current drive to E:, and the active directory is the current directory on E:.
> However, if the current drive is C:, and you enter the command
>
> ```
> cd E:\comp170
> ```
>
> then you change the current directory on E:, but *the current drive remains C:.*

We described above how you can use a Windows Explorer/Finder folder to open a *new* terminal. If you just want to change directory in an existing terminal, there is also a shortcut to copy a long folder name, given a Windows Explorer/Finder folder:

Windows

- Depending on the setup of your options, in the address bar you may *not* see a clear path with a drive and backslashes. In that case generally clicking to the right of any directory in the path converts the view to the version we use on the command line.

- When you see a full absolute path, you can just note it and manually copy it, or else select it all and copy it and follow the instructions in *Copy and Paste* to later paste in the command window.

- In any case click in the terminal window, type *cd* and a space, then type or paste the path.

- Of course, you can also go the other way – if you see the current directory name in the Windows prompt, type that into an Explorer address bar to see its contents in a GUI window.

On a Mac there is an easier shortcut:

- Type *cd* and a space to start the command in the terminal

- Locate the directory you want as a subfolder in the Finder (not opening the directory).

- Drag the directory icon to the terminal. The path gets pasted! Press return.

18.3.2 Common Commands

The command shell waits for you to type in a *command* (a short name that the shell recognizes) followed by 0 or more *parameters* separated by spaces (and Enter). Note that if a parameter contains spaces you must surround the parameter value with matching single or double quotes – you'll see an example later.

We are going to mention some of the simplest uses of basic commands. More advanced documentation would include more options.

Some commands are common between the Windows and Mac shells:

dir (Windows) or ls (Mac) to list all the files a in the current directory or a named directory.

cd stands for *Change Directory* – you can use this command to change the current working directory to a different one.

You can use this command to change to directories where your C# program source files are located, if different from the initial directory.

On Windows, suppose you created a directory C:\COMP170\hello; to change to that, type *cd C:\COMP170\hello* and press Enter – the shell prompt will change to show this new directory location and programs like *mcs* and *mono* will be able to access files there, directly by name. If the Comp170 directory was you current directory, it would be shorter to use relative paths and just `cd hello`. Remember if you want a different Windows drive, you must first use a *drive change command*.

On a Mac you can also use either an absolute or a relative path with `cd`.

If you included a space in one or more of the directory names, for example C:\COMP 170\hello (a space between COMP and 170) you should enclose that part(s) in quotes like so: *cd C:\"COMP 170"\hello*

Mac Note: if you type just *cd* and press Enter you will change back to your home directory. There is also a shorthand name for your home directory in command paths: tilde (~), often shifted backquote on the keyboard. Sorry, no such thing with Windows.

mkdir stands for make directory – you can use *mkdir* to create a new empty directory in the current directory.

For example, on a Mac with current directory /Users/*YourLogin*, type *mkdir hello* and press Enter – this will create a new directory /Users/*yourLogin*/hello if it did not exist before; you can now create a C# source file in that directory and enter *cd hello* in the command shell.

An optional Windows abbreviation is *md*.

rmdir removes an *empty* directory that you give as parameter, e.g.,

> rmdir hello

With Mono installed (and for Windows, with a Mono command window), the programs associated with Mono can be used:

mcs compiles one or more listed C# source files without using Xamarin Studio.

csharp is the interactive C# statement testing program.

Other useful commands, with different names for Windows and Mac, are listed next by generic function, with general Windows syntax first and Mac second, and

then often examples in the same order:

Display the contents of a text file in the command window. The Unix/Mac name origin: a more complicated use of cat is to con**cat**enate files.

> type *textFileName*
> cat *textFileName*

> type my_program.cs
> cat my_program.cs

Make a copy of a file. Caution: If the second file already exists, you wipe out the original contents!

> copy *originalFile copyName*
> cp *originalFile copyName*

> copy prog.cs prog_bak.cs
> cp prog.cs prog_bak.cs

Erase or remove a file:

> erase *fileToKill*
> rm *fileToKill*

> erase poorAttempt.cs
> rm poorAttempt.cs

Another Windows equivalent is `del` (short for delete).

Help on a command:

> help *commandName*;
> *commandName* –help

Note the double dash above: This sometimes works for concise help on a Mac while you can generally get immensely detailed help overload on a Mac from

> man *commandName*

18.3.3 Scripts

This is not a subject of this course, but commands can be combined into script files.

Scripting languages are in fact whole new specialized programming languages, that include many of the types of programming statements found in C#.

18.3.4 Copy and Paste

Copying or pasting with a Mac is is the same with a terminal as in other editing: Use the same Apple Command key with C or P, and you can select with the mouse.

In Windows it is more complicated to use a command window: You can paste into the current command line by *right* clicking on the Command Window Title bar, and select edit and then paste.

By default a Windows command window is not sensitive to the mouse. You can change so that it is sensitive for select and copy: Right click in the title bar, select defaults, and make sure the check boxes under edit options are *all* checked. (The last two are explained in the next section.) Click OK. Then you can select with mouse and press Enter for the selection to be remembered in the copy buffer.

18.3.5 Command Line Shortcuts

Both Windows and Mac (with the right options selected, like the Windows check boxes in the last section), allow you to reduce typing:

You can bring back a previous command for the history of commands that are automatically remembered: Use the up and down arrows. This makes it very easy to run the same command again, or to make slight edits.

Both Windows and OS-X can see what files are in any directory being referred to. If you just start to type a file or folder name and then press the Tab key, both

Windows and OS-X will do *file completion* to complete the name if there is no ambiguity. If there is ambiguity, they work differently:

- Windows will cycle through all the options as you keep pressing Tab.

- On the first tab OS-X will do nothing but give a sound if there is ambiguity, but the second tab will list all the options. Then you need to type enough more to disambiguate the meaning.

18.4 Precedence of Operators

Earlier lines have higher precedence. Only operators used in this book are included:

```
obj.field  f(x)  a[i]  n++  n--  new
+  -  ! (Type)x (Unary operators)
* / %
+ - (binary)
< > <= >=
== !=
&&
||
=  *=  /=  %=  +=  -=
```

All symbols are listed at the beginning of the index.

Parentheses for grouping are encouraged with less common combinations, even if not strictly necessary.

18.5 Homework: Grade Calculation

Create a program file `grade_calc.cs` for this assignment. You are going to be putting together your first programming assignment where you will be taking the various concepts we have learned thus far from class and to put together your first meaningful program on your own.

This program will incorporate the following elements:

- Prompt a user for input.

- Perform some rudimentary calculations.

- Make some decisions.

- Produce output.

As we've mentioned earlier in class, our focus is going to be on learning how to write computer programs that start with a Main() function and perhaps use other functions as needed to *get a particular job done*. Eventually, we will be incorporating more and more advanced elements, such as classes and objects. For now, we would like you to organize your program according to the guidelines set forth here.

18.5.1 Program Summary

Our first program is based on a common task that every course professor/instructor needs to do: make grades. In any given course, there is a *grading scale* and a set of *categories*.

Here is sample output from two runs of the program. The only data entered by the user are show in **boldface** for illustration here.

One successful run with the data used above:

> Enter weights for each part as an integer
> percentage of the final grade:
> Exams: **40**
> Labs: **15**
> Homework: **15**
> Project: **20**
> Participation: **10**
>
> Enter decimal numbers for the averages in each part:
> Exams: **50**
> Labs: **100**
> Homework: **100**
> Project: **100**
> Participation: **5**

Your grade is 70.5%
Your letter grade is C-.

A run with bad weights:

Enter weights for each part as an integer
percentage of the final grade:
Exams: **30**
Labs: **10**
Homework: **10**
Project: **10**
Participation: **10**

Your weights add to 70, not 100.
This grading program is ending.

18.5.2 Details

Make your program file have the name `grade_calc.cs`.

This is based on the idea of Dr. Thiruvathukal's own legendary course syllabus. We're going to start by assuming that there is a fixed set of categories. As an example we assume Dr. Thiruvathukal's categories.

In the example below we work out for Dr. Thiruvathukal's weights in each category, though your program should prompt the user for these integer percentages:

- exams - 40% (integer weight is 40)

- labs - 15% (weight 15)

- homework - 15% (weight 15)

- project - 20% (weight 20)

- participation - 10% (weight 10)

Your program will prompt the user for each the weights for each of the categories. These weights will be entered as integers, which must add up to 100.

If the weights do not add up to 100, print a message and end the program. You can use an `if-else` construction here. An alternative is an `if` statement to test

for a bad sum. In the block of statements that go with the `if` statement, you can put not only the message to the user, but also a statement:

```
return;
```

Recall that a function ends when a return statement is reached. You may not have heard that this can also be used with a `void` function. In a `void` function there is no return value in the `return` statement.

Assuming the weights add to 100, then we will use these weights to compute your grade as a `double`, which gives you the best precision when it comes to floating-point arithmetic.

We'll talk in class about why we want the weights to be integers. Because floating-point mathematics is not 100% precise, it is important that we have an accurate way to know that the weights *really add up* to 100. The only way to be assured of this is to use *integers*. We will actually use floating-point calculations to compute the grade, because we have a certain tolerance for errors at this stage. (This is a fairly advanced topic that is covered extensively in courses like COMP 264/Systems Programming and even more advanced courses like Numerical Analysis, Comp 308.)

We are going to pretend that we already know our score (as a percentage) for each one of these categories, so it will be fairly simple to compute the grade.

For each category, you will define a weight (int) and a score (double). Then you will sum up the weight * score and divide by 100.0 (to get a double-precision floating-point result).

This is best illustrated by example.

George is a student in COMP 170. He has the following averages for each category to date:

- exams: 50%
- labs: 100%
- homework: 100%
- project: 100%
- participation: 5%

The following session with the `csharp` interpreter shows the how you would declare all of the needed variables and the calculation to be performed:

```
csharp> int exam_weight = 40;
csharp> int lab_weight = 15;
csharp> int hw_weight = 15;
csharp> int project_weight = 20;
csharp> int participation_weight = 10;

csharp> double exam_grade = 50.0;
csharp> double lab_grade = 100;
csharp> double homework_grade = 100;
csharp> double project_grade = 100;
csharp> double participation_grade = 5;
```

This is intended only to be as an example though. Your program must ask the user to enter each of these variables.

Once we have all of the weights and scores entered, we can calculate the grade as follows. This is a long expression: It is continued on multiple lines. Recall all the > symbols are csharp prompts are not part of the expression:

```
csharp> double grade = (exam_weight * exam_grade +
      > homework_weight* homework_grade +
      > lab_weight * lab_grade + project_weight * project_grade +
      > participation_weight * participation_grade) / 100.0;
```

Then you can display the grade as a percentage:

```
csharp> Console.WriteLine("Your grade is {0}%", grade);
Your grade is 70.5%
```

Now for the fun part. We will use `if` statements to print the letter grade. You will actually need to use multiple `if` statements to test the conditions. A way of thinking of how you would write the logic for determining your grade is similar to how you tend to think of the *best* grade you can *hope for* in any given class. (We know that we used to do this as students.)

Here is the thought process:

- If my grade is 93 (93.0) or higher, I'm getting an A.
- If my grade is 90 or higher (but less than 93), I am getting an A-.

- If my grade is 87 or higher (but less than 90), I am getting a B+.

- And so on...

- Finally, if I am less than 60, I am unlikely to pass.

We'll come to see how *logic* plays a major role in computer science–sometimes even more of a role than other mathematical aspects. In this particular program, however, we see a bit of the best of both worlds. We're doing *arithmetic* calculations to *compute* the grade. But we are using *logic* to determine the grade in the cold reality that we all know and love: the bottom-line grade.

This assignment can be started after the data chapter, because you can do most all of it with tools learned so far. Add the parts with `if` statements when you have been introduced to `if` statements. (Initially be sure to use data that makes the weights actually add up to 100.)

You should be able to write the program more concisely and readably if you use functions developed in class for the prompting user input.

18.5.3 Grading Rubric

> **Warning:** As a general rule, we expect programs to be complete, compile correctly, run, and be thoroughly tested. We are able to grade an incomplete program but will only give at most 10/25 for effort. Instead of submitting something incomplete, you are encouraged to complete your program and submit it per the late policy. Start early and get help!

25 point assignment broken down as follows:

- Enter weights, with prompts **[3]**

- End if the weights do not add to 100: **[5]**

- Enter grades, with prompts: **[3]**

- Calculate the numerical average and display with a label: **[5]**

- Calculate the letter grade and display witha label: **[5]**

- Use formatting standards for indentation: **[4]**

    - Sequential statements at the same level of indentation

- Blocks of statements inside of braces indented
- Closing brace for a statement block always lining up with the heading before the start of the block.

18.5.4 Logs and Partners

You may work with a partner, following good pair-programming practice, sharing responsibility for all parts.

Only one of a pair needs to submit the actual programming assignment. However *both* students, *independently*, should write and include a log in their Homework submission. Students working alone should also submit a log, with fewer parts.

Each individual's log should indicate each of the following clearly:

- Your name and who your partner is (if you have one)
- Your approximate total number of hours working on the homework
- Some comment about how it went - what was hard ...
- An assessment of your contribution (if you have a partner)
- An assessment of your partner's contribution (if you have a partner).

Just omit the parts about a partner if you do not have one.

Note: Name the log file with the exact file name: "log.txt" and make it a plain text file. You can create it in a program editor or in a fancy document editor. If you use a fancy document editor, be sure to a "Save As..." dialog, and select the file format "plain text", usually indicated by the ".txt" suffix. It does not work to save a file in the default word processor format, and then later just change its name (but not its format) in the file system.

18.6 Homework: Grade Calculation from Individual Scores

In the previous assignment, we calculated grades based on a *memorized* overall grade within each of the categories below, as in this example:

- exams - 40% (integer weight is 40)

- labs - 15% (weight 15)

- homework - 15% (weight 15)

- project - 20% (weight 20)

- participation - 10% (weight 10)

In this assignment, we are going to change the specification slightly to make the program a bit smarter. Instead of someone having to remember what their average grade was for each category, we will prompt the user for the number of items within each category (e.g. number of exams, number of labs, etc.), have the user enter individual grades, and have the program calculate the average for the category.

As usual, we will begin by specifying *requirements*. User responses are shown **bold faced**.

18.6.1 Functional Requirements

1. Instead of bombing out if the weights don't add up to 100, use *Do-While Loops* to prompt the user again for all of the weights until they do add up to 100. A `do { ... } while` loop is the right choice here, because you can test all of the weights at the end of the loop, after each time they have been entered in the loop.

2. Write a function, `FindAverage`, to do the following. The example refers to the category exam, but you will want your code to work for each category, and hence the category *name* will need to be a parameter to `FindAverage`.)

   Prompt the user for the number of items in the category:

   > Please enter the number grades in category exam: **4**

   Instead of prompting the user for an overall average exam grade, use a loop to read one grade at a time. The grades will be added together (on the fly) to give the grade for that category. For example, after you have asked for the number of exams, you'd prompt the user to enter each exam grade and have the program compute the sum. As soon as a category sum is calculated, also print out the average as shown in the sample below:

> Please enter the grade for exam 1: **100**
> Please enter the grade for exam 2: **90**
> Please enter the grade for exam 3: **80**
> Please enter the grade for exam 4: **92**
> Calculated average exam grade = 90.5

Of course you must return the grade to the caller for use in the overall weighted average grade.

A category may have only a single grade, in which case the user will just enter the number of grades as 1.

3. Once you have read in the data for each of the items within a category, you'll basically be able to *reuse* the code that you developed in the previous assignment to compute the weighted average and print the final letter grade.

4. Print the final numerical average, *this time rounded to one decimal place*. If the final average was actually 93.125, you would print 93.1. If the final average was actually 93, you would print 93.0. If the final average was actually 93.175, you would print 93.2.

18.6.2 Style Requirements

1. For this assignment, you are expected to start using functions for all aspects of the assignment. For example, it can become tedious in a hurry to write code to prompt for each of exams, labs, homework, etc. when a single function (with parameter named *category*) could be used to avoid repeating yourself. In particular you should write your function to take advantage of our `UI` class, from *User Input: UI*.

2. Also beginning with this assignment, it is expected that your work will be presented neatly. That is, we expect the following:

   - proper indentation that makes your program more readable by other humans. Use all spaces, not tabs to indent. You never know what default tabs your grader will have set up.

   - proper naming of classes and functions. In C#, the convention is to begin a name with a capital letter. You can have multiple words in a name, but these should be capitalized using a method known as

CamelCase *[CamelCase]*. We also recommend this same naming convention for variables but with a lowercase first letter. For variables, we are also ok with the use of underscores. For example, in homework 1 we used names like *exam_grade*. If you use CamelCase, you can name this variable *examGrade*.

- If you have any questions about the neatness or appearance of your code, please talk to the instructor or teaching assistant.

- This guide from CIS 193 at *[UPennCSharp]* provides a nice set of conventions to follow. We include this here so you know that other faculty at other universities also consider neatness/appearance to be important.

18.6.3 Grading Rubric

Warning: As a general rule, we expect programs to be complete, compile correctly, run, and be thoroughly tested. We are able to grade an incomplete program but will only give at most 10/25 for effort. Instead of submitting something incomplete, you are encouraged to complete your program and submit it per the late policy. Start early and get help!

25 point assignment broken down as follows:

- Loop until weights add to 100: 5

- Average any number of grades in a category: 5

- One function that is reused and works for the average in each category: 5

- Print final numerical grade rounded to one decimal place: 2

- Previous program features still work: 3

- Style: 5

18.6.4 Logs and Partners

You may work with a partner, following good pair-programming practice, sharing responsibility for all parts.

Only one of a pair needs to submit the actual programming assignment. However *both* students, *independently*, should write and include a log in their Homework submission. Students working alone should also submit a log, with fewer parts.

Each individual's log should indicate each of the following clearly:

- Your name and who your partner is (if you have one)

- Your approximate total number of hours working on the homework

- Some comment about how it went - what was hard ...

- An assessment of your contribution (if you have a partner)

- An assessment of your partner's contribution (if you have a partner).

Just omit the parts about a partner if you do not have one.

Note: Name the log file with the exact file name: "log.txt" and make it a plain text file. You can create it in a program editor or in a fancy document editor. If you use a fancy document editor, be sure to a "Save As..." dialog, and select the file format "plain text", usually indicated by the ".txt" suffix. It does not work to save a file in the default word processor format, and then just change its name (but not its format) in the file system.

18.7 Homework: Grade File

Copy project files in grade_file_homework_stub to your own project. Then you should have copies of the source file grade_files.cs for you to *complete* for this homework. The folder also contains sample data files including the examples discussed below.

In this assignment, we're going to begin taking steps to help you achieve greater *independence* when it comes to programming. This means (among other things) that you will be given what is commonly known as a specification. In software development–and in the business world in general, it is customary to capture a set of *business requirements* in what is commonly known as a *requirements specification* document. While what you read here will be much more concise, we want you to become familiar with requirements-driven thinking, without which many real-world software projects fail.

After presenting the set of requirements, we will give you some hints for how to *implement* the requirements. These hints may or may not prove completely helpful to you, and you are also invited to come up with your own solutions. As we inch closer to the semester project, you're going to want to use your imagination to create a good solution to a problem.

18.7.1 Brief Problem Statement

The previous two homework assignments represent a great simplification of the real-world process of grading. The notion that grade information must be entered manually is rather tedious, not to mention error prone. In the real world, grade information would be kept in a file (a spreadsheet is common), from which various calculations and summary reports could be generated.

In this assignment, the problem we are trying to solve is to take all of the *raw* grade data from one or more student files and prepare a *summary report* file with a line for each student.

Although we could do all of what we're describing here with a *spreadsheet*, the point is to show how we can use C# to read in a simplified form of comma-separated data, process it, and do some general-purpose calculations on the data.

18.7.2 Using C#

We'll be making use of a number of C# features (some old, some new) in this homework:

- decisions, loops, strings, and functions
- files
- arrays

18.7.3 Requirements

1. Unlike in previous assignments, this program must accept data from a collection of input files (that is, it will not be reading most of the data from the class `Console`).

2. The program needs a course abbreviation from the user. If there is a command line argument, use it as the course abbreviation. Make sure your code can read a command-line argument using the special form of `Main(string[] args)` already in the stub grade_files.cs. If the user does not provide at command line argument, prompt the user for it once the program starts. An example would be comp170. All data files will include the course abbreviation as part of their name. We will use comp170 in the examples below, but it could be something else. The folder also contains sample data files for a course abbreviation comp150.

Note that these data files are not in the Xamarin Studio execution directory, but in the project directory, so the *FIO Helper Class* is useful to provide flexibility in reading the data files.

3. There are two master files for any course. One is "categories_" + the course abbreviation + ".txt". For example, `categories_comp170.txt` is a sample data file provided and used below.

It will contain three lines. The first line is a comma separated list of category names like

```
Exam, Lab, Homework, Project, Class Participation
```

There may be extra spaces after the commas. Categories will be chosen so that *each one starts with a different letter*.

The second line contains the integer weights for each category, like

```
40, 15, 15, 20, 10
```

They do *not* need to add to 100. If the sum is called totWeights, get the final grade by summing for each category:

```
(category weight)(category grade)/totWeights
```

The third line will contain the number of grades in each category, like

```
2, 5, 3, 1, 2
```

The second master file will be "students_" + the course abbreviation + ".txt". For example `students_comp170.txt`. It will contain a list of student information records. Each record (one per input line) will have the following structure:

Student ID, Last Name, First Name

For example, the sample data file `students_comp170.txt` starts:

```
P12345678, Doe, John
P00000001, Hernandez, Maria
```

4. There will be a secondary file for each student, named after the student id and the course abbreviation and ".data". For example, John's scores would be kept in a file named `P12345678comp170.data`. Maria's scores would be in `P00000001comp170.data`. Each record (one per file line will have the following structure:

Category letter, Item, Points Earned

where:

- category letter is the first letter of the category. With the categories given in the example above, they would be E, L, H, P, and C.

- item is a number within that category (1, 2, 3, ...) - only used in part of the extra credit.

- points earned is a real number

- the lines are in no special order.

Sample data file `P12345678comp170.data` starts:

```
H, 1, 90
C, 1, 100
L, 3, 100
L, 2, 80
H, 2, 80
E, 1, 90
```

5. The program will process the data from each student file and calculate the average within each category, and then the weighted overall average. Also display the letter grade for each student, using code derived from the previous assignment.

6. Your program writes the final report file. It is named with the course abbreviation + "_summary.txt". Example: "comp170_summary.txt". This file must have a line for each student showing the student's last name, first

name, weighted average rounded to one decimal place, and letter grade. File
`comp170_summary.txt` would start with lines:

```
Doe, John 78.9 C+
Hernandez, Maria 88.2 B+
```

Write this file to the same directory where you found the input data. Again
the *FIO Helper Class* is useful.

7. The rest of the test data for course abbreviations comp170 and all the data
   for comp150 is in the homework directory. There are also sample solution
   files for the summaries (including some extra credit additions at the ends of
   lines). Their names end in `_solution.txt` to distinguish them from the
   summary files *you* should generate in tests.

   While your program should certainly work for course abbreviations
   comp170 and comp150, it should also work in general for any data files
   your refer to in the defined formats and place in the same folder.

8. Turn in materials as in the last homework, including a single copy of the
   homework source files and a log.txt file for each student, in the same form
   as for the last homework.

18.7.4 Hints

1. Read *Files, Paths, and Directories*. You're still going to need ReadLine()
   and WriteLine() in this assignment, the only difference is that we'll get the
   input from a file instead of the Console. The parameter syntax will be the
   same.

2. For each file line you'll want to use the *String Method Split*, and then the
   `Trim` method from *More String Methods* on each part to remove surround-
   ing spaces. Then use indexing to get the field of interest. (More below.)

3. You'll need an *outer loop* to read the records from the master name file.
   You'll need an *inner loop* (or a loop inside of a function) to read the records
   for each student.

4. When processing the records from a student file, you should process each
   one separately and not assumed they are grouped in any particular order.

This means, specifically, that your program simply reads a record, decides what category it is in, and updates the *running total* for that category. Once the entire file has been read, you can compute the average for each category based on the *number of items* that *should* be in that category, which may be more than the number of records in the file for items turned in.

5. There is no need to *keep* a score after you've read it and immediately used it. *Do* use an array, however, for the running total for each category.

6. In order to deal with a varying number of categories and different possible first letter codes, you will need to split the category name line into an array, say

```
string[] categories;
```

To know where to update data for each category, you can use this function after you read in a code, to determine the proper index. It is already in the stub of the solution file grade_files.cs:

```
/// Take the first letter code for a catagory, and
/// return the index of that category in categories.
static int codeIndex(string code, string[] categories)
{
   for (int i = 0; i < categories.Length; i++) {
      if (categories[i].Trim().StartsWith(code)) {
         return i;
      }
   }
   return -1; //required by compiler: shouldn't reach
}
```

You may assume the data is good and the -1 is never returned, but the compiler requires this last line.

7. You cannot have one fixed formula to calculate the final weighted grade, because you do not know the number of categories when writing the code. You will have to accumulate parts in a loop.

8. Test thoroughly! Be sure to test with and without command line parameter and with multiple data sets.

18.7.5 Grading Rubric (25 points)

1. Get the abbreviation from the command line if it is there. **[2]**

2. Otherwise get the abbreviation from prompting the user. **[1]**

3. Read the categories file and parse lines. **[2]**

4. Deal with each student. **[3]**

5. Calculate the cumulative grades in each category, reading a student's file once, using arrays. **[5]**

6. Calculate the overall grade and letter grade. **[3]**

7. Generate summary entries. **[3]**

8. Use functions where there would otherwise be two several-line blocks of code differing only in the name of the data evaluated and the name of the result generated. **[2]**

9. Use good style: formatting, naming conventions, meaningful names other than for simple array indices, lack of redundant code. **[4]**

Optional Extra Credit Opportunities! You may choose to do any combination that does not include both of the last two options about missing work.

1. Format the summary file in nice columns. Include the grades for each category, rounded to one decimal place. Include a heading line. For example the summary for the repository example Comp150 could start:

```
Name: Last, First          Avg Gr    E     H     P
Hopper, Grace              100.0 A  100.0 100.0 100.0
```

You may assume the last-first name field fits in 25 columns. Copy the first three column headings from above. The column headings for the categories can just be their one letter code. Names and letter grades should be left-justified (padded on the right, by using a *negative* field width). See *Left Justification*. **[2]**

2. Change the scheme for calculating letter grades to use a function that calculates the proper grade, where the only `if` statement is one simple one inside a *loop*. The `if` statement will have a `return` statement in its body, and no `else`. The loop will need to use corresponding arrays of data for grade cutoffs and grade names. **[3]**

3. For any student who has missed passing in all the required items, generate extra data on missing work in the summary, at the right end of the line for the student. Add this to whichever version of the earlier parts you use. Include an addendum starting with "Missing: " only if there are not enough grades in one or more categories. For each category where one or more grades is missing, including a count of the number of grades missing followed by the category letter. An example using the example categories is:

```
Doe, John 68.5 D+ Missing: 2 L 1 H
Smith, Chris 83.2 B Missing: 1 L
Star, Anna 91.2 A-
```

meaning Doe has 2 labs missing and 1 homework missing. Smith is missing one lab. Star has done all assigned work, since nothing is added. The solution files display this extra credit addition on the ends of lines. **[3]**

4. This is a much harder alternate version for handling missing work: Unlike the previous format, do not count and print the number of missing entries in each category in a form like "2 L ". Replace such an entry with a list of *each* item missing, in order, as in "L:1, 4 ", meaning labs 1 and 4 were missing. Assume that the expected item numbers for a category run from 1 through the number of grades in the category. You may assume no item number for the same category appears twice. For example, with the sample data files given in the repository for comp170, the summary line for John Doe would be:

```
Doe, John 78.9 C+ Missing: L: 1, 4 H: 3
```

The most straightforward way to do this requires something like an array of arrays, array of lists or array of sets. You may need to read ahead if you want to use one of these approaches. **[5]**

18.8 Homework: Book List

Objectives:

- Complete a simple data storing class (Book), with fields for title, author, and year of publication.

- Complete a class with a Collection (BookList) that uses the public methods of another class you wrote (Book), and select various data from the list.

- Complete a testing program (TestBookList), that creates a BookList, adds Books to the BookList, and tests BookList methods clearly and completely for a user looking at the output of the program and *not* the source code..

Copy stub files from the project books_homework_stub to your own project. Stubs for the assignment files are book.cs, book_list.cs, and test_book_list.cs,

Some of the method stubs included are only to be fleshed out if you are doing the corresponding extra credit option. They include a comment, just inside the method, `// code for extra credit`. There are also extra files used by the extra credit portion. They are discussed in the Extra Credit section at the end of the assignment.

Complete the first line in each file to show your names. At the top of the Book class include any comments about help in *all* of the classes.

Create methods one at a time, and test them. Complete `book.cs` first, preferably testing along the way. (You can write an initial version of the testing program, so it does not depend on BookList.) Then add methods to `book_list.cs`, and concurrently add and run tests in `test_book_list.cs`. Testing the Book class first means that when you get to BookList you can have more confidence that any problems you have are from the latest part you wrote, not parts written earlier in the class Book.

Remember to have each individual submit a **log**, `log.txt`, in the same format as the last assignment.

18.8.1 Book class

See the stub file provided. It should have instance fields for the author, title, and year (published).

Complete the constructor:

```
public Book(string title, string author, int year)
```

that initializes the fields. (Be careful, as we have discussed in class, when using the same names for these parameters as the instance variables!)

It should have three standard (one line) getter methods:

```
public string GetTitle()

public string GetAuthor()

public int GetYear()
```

and

```
public override string ToString()
```

`ToString` should return a *single* string spread across three lines, with no newline at the end. For example if the `Book` fields were "C# Yellow Book", "Rob Miles", and 2011, the string should appear, when printed, as

```
Title: C# Yellow Book
Author: Rob Miles
Year: 2011
```

The `override` in the heading is important so the compiler knows that this is the official method for the system to used implicitly to convert the object to a string.

Remember the use of @ with multi-line string literals.

18.8.2 BookList class

It has just one instance variable, already declared:

```
private List<Book> list;
```

It has a constructor (already written - creating an empty List):

```
public BookList()
```

It should have public methods:

```
    // Add book to the list.
    // The regular assignment version always returns true.
    public bool Addbook(Book book)
```

The regular version should just leave the final `return true;` The extra credit version is more elaborate.

Further methods:

```
    /// List the full descriptions of each book,
    /// with each book separated by a blank line.
    public void PrintList() //
```

```
    // List the titles (only!), one per line, of each book
    // in the list that is by the specified author.
    public void PrintTitlesByAuthor(string author)
```

```
    // List the full descriptions of each book printed
    // in the range of years specified,
    // with each book separated by a blank line.
    public void PrintBooksInYears(int firstYear, int lastYear)
```

For instance if the list included books published in 1807, 1983, 2004, 1948, 1990, and 2001, the statement

```
PrintBooksInYears(1940, 1990);
```

would list the books from 1983, 1948, and 1990.

18.8.3 TestBookList class

It should have a `Main` program that creates a BookList, adds some books to it (more than in the skeleton!), and convincingly displays tests of each of BookList's methods that exercise all paths through your code. Check for one-off errors in PrintBookYears. With all the methods that print something, the results are easy to see. *Do print a label*, as in the skeleton, before printing output from each method test, so that the user of the program can see the correctness of the test *without any knowledge of the source code*!

18.8.4 Grading Rubric

Book class

- [1 point] public Book(string title, string author, int year)
- [1] public string GetTitle()
- [1] public string GetAuthor()

- [1] public int GetYear()

- [2] public override string ToString()

BookList class

- [2] public bool AddBook(Book book)

- [2] public void PrintList()

- [2] public void PrintTitlesByAuthor(string author)

- [2] public void PrintBooksInYears(int firstYear, int lastYear)

TestBookList

- [2] Supply data to screen indicating what test is being done with what data and what results, so it is clear that each test works without looking at the source code.

- [5] Convincingly display tests of each of BookList's methods that exercise all paths through your code.

Overall:

- [4] Make your code easy to read - follow indenting standards, use reasonable identifier names.... Do not duplicate code when you could call a method already written.

18.8.5 Extra Credit

You may do any of the numbered options, except that the last one requires you to do the previous one first.

To get full credit for any particular option, tests for it must be *fully integrated* into TestBookList!

1. [2 points] Complete

```
/// Return a single string containing the same data as
/// printed by PrintList, including a final newline.
public override string ToString()
```

Also *change* the PrintList method body to the one line:

```
Console.Write(this);
```

(The `Write` and `WriteLine` methods print objects by using their `ToString` methods.)

Be sure to make this addition to TestBookList: Test the `ToString` method by converting the resulting BookList description string to upper case before printing it (which should produce a different result than the regular mixed case of the `PrintList` method test).

2. [4 points]

In the Book class, a new constructor:

```
/// Construct a Book, taking data from reader.
/// Read through three lines that contain the
/// title, author, and year of publication, respectively.
/// There may be an extra blank line at the beginning.
/// If so ignore it.
/// Nothing beyond the line with the year is read.
public Book(StreamReader reader)
```

In class BookList, a new constructor:

```
// Construct a new BookList using Book data read from
// reader.  The data coming from reader will contain groups
// of three line descriptions useful for the Book constructor
// that reads from a stream.  Each three-line book description
// *may or may not* be preceded by an empty line.
public BookList(StreamReader reader)
```

For testing we included special files in the right format: books_homework_stub/books.txt and books_homework_stub/morebooks.txt.

You will also want to include a reference to fio/fio.cs, so the text files are easy to find.

3. [4 points]

In class Book:

```
// Return true if all the corresponding fields in this Book
// and in aBook are equal.  Return false otherwise.
```

```
        public bool IsEqual(Book aBook)
        {  // code for extra credit

            return true; //so skeleton compiles
        }
    }
}
```

It is essential to have the `IsEqual` method working in Book before any of the new code in BookList, which all depends on the definition of `IsEqual` for a Book.

NOTE: We chose the name `IsEqual` to distinguish it from the more general `Equals` override that you could write. The `Equals` override allows for a parameter of any object type. With skills from Comp 271 you you be able to write the `Equals` override.

In class BookList:

```
        // Test if aBook is contained in this BookList.
        // Return true if book IsEqual to a Book in the list,
        // false otherwise.
        public bool Contains(Book book)
```

Caution: Do NOT try to use the `List` method `Contains`: Because we did *not* override the `Equals` method to specialize it for Books, the `List` method `Contains` will *fail*. You need to do a litle bit more work and write your own version with a loop.

Change the `AddBook` method from the regular assignment, so it satisfies this documentation:

```
        /// Adds aBook to the list if aBook is
        /// not already in the list.
        /// Return true if aBook is added,
        /// and false if it was already in the list.
```

In TestBookList you need to react to the return value, too.

4. [2 points] This one requires the previous elaboration of `AddBook`. In Book-List:

```
        // Add each Book in books to this BookList.
        // if it is not already contained in this BookList.
        // Return true if the current list was changed.
        // Return false if each Book in books is a
        // duplicate of a Book in the current list.
        public bool AddAll(BookList books)
```

You might want to code it first without worrying about the correct return value; then do the complete version. There are multiple approach to determining the return value, some much easier than others!

To fully test in TestBookList, you need to react to the return value, too.

18.9 Group Project

18.9.1 Objectives

- Being creative, imagining and describing a program, and working it through to completion
- Working in a team:
    - Communicating to each other
    - Dividing responsibilities
    - Helping each other
    - Finding consensus
    - Dealing with conflict
    - Providing process feedback
    - Integrating parts created by several people
- Developing new classes to fit your needs
- Using the .Net API library
- Designing a program for refinement
- Testing

- Evolving a program

- Creating documentation for user and implementers

- Programming in the large – not a small predefined problem

- Make something that is fun

See the project `csproject_stub`, already discussed in class.

What to turn in:

- Periodic reports

- intermediate implementation

- a final presentation

- a final implementation including user and programmer documentation and process documentation

- individual evaluations

18.9.2 Overview

You will be assigned to groups of 3-4 people to work on a project that will extend until the end of the semester, with only a little other work introduced in class, and of course your last exam. This leaves a month of course time (in a regular semester) for the project (classes, labs, dedicated homework time), ending with presentations in final exam period. Your group will be designing and implementing a project of your choosing. You could choose to make a text based game, starting from our game skeleton or not, or something completely different, started from scratch. The project should have some clear focus or aim, For instance a game should be able to be won.

Start by brainstorming and listing ideas – do not criticize ideas at this point. That is what is meant by brainstorming - having your internal critic going inhibits creativity. After you have a large list from brainstorming, it is time to think more practically and settle on one basic situation, and think of a considerable list of features you would like it to have. Order the features, considering importance, apparent ease of development, and what depends on what else. Get something simple working, and then add a few features at a time, testing the pieces added and the whole project so far. Test, debug, and make sure the program works

completely before using your past experience to decide what to add next. This may different than what you imagined before the work on your first stage! Like the provided project, early stages do not need to be full featured, but make sure that you are building up to a version with an aim, and which includes interesting features. You should end with a program that has enough instructions provided for the user, so someone who knows *nothing* of your implementation process or intentions can use the program successfully. Your implementation should also be documented, imagining that a new team of programmers is about to take over after your departure, looking to create yet another version.

18.9.3 Your Team

Your instructor will tell you about team makeup.

There are a number of roles that must be filled by team members. Some will be shared between all members, like coder, but for each role there should be a lead person who makes sure all the contributions come together. Each person will have more than one role. All members are expected to pull their own weight, though not all in exactly the same roles. Everyone should make sustained contributions, every week, documented in the weekly reports. Understand that this project will be the major course commitment for the rest of the semester. These roles vary from rather small to central. Not all are important immediately.

18.9.4 Roles

1. Leader: Makes sure the team is coordinated, encourage consensus on the overall plan, oversee that the agreements are carried through, be available as contact person for the team and the TA and your instructor.

2. Lead programmer: Keep track of different people's parts to make sure they fit together.

3. Coder/unit tester: *Everyone* must have a significant but not necessarily equal part in this job. Each person should have primary responsibility for one or more cohesive substantive units, and code and test and be particularly familiar with those parts. Do your best to make individual parts be cohesive and loosely coupled with other people's work, to save a lot of pain in the testing and debugging phase. When coding you are still encouraged to do

pair programming, though what pair from the team is working together at different times may be fluid. The lead programmer might be involved in pairs more often than others, but be sure the other coders get to drive often.

4. Librarian/version coordinator: The default should be for you to have a box.com folder shared with your whole team and me and your TA, with all as editors. Your folder should have the name of your team. You should always have a folder that contains the latest working version of the project. You should also keep old versions, for instance copied into numbered version folders. Box does not handle successive versions automatically. You can choose to use the more capable professional combination of Bitbucket and *Mercurial and Teamwork*. The latter will have a learning curve, and in that case this person should be the best informed on Mercurial, and help the other team members.

5. Report coordinator: Gather the contributions for reports from team members and make sure the whole reports get to posted on schedule. Your instructor needs a clear idea of the contributions of each member each week. If a team member is not clear on this to the report writer, *the report writer needs to be insistent*.

6. Instruction coordinator: Make sure there are clear written documents and help within the program for the user, who you assume is not a C# programmer and knows nothing about your program at the start.

7. Documentation coordinator: Make sure the documentation is clear for method users/consumers. This includes the documentation for programmers before the headings of methods and classes. This is for any time someone wants to use (not rewrite) a class or method you wrote. Also have implementer documentation, for someone who will want to modify or debug your code and needs to understand the details of your internal implementations. The extent of this can be greatly minimized by good naming.

8. Quality manager: Take charge of integrated tests of the whole program (following coder's unit tests). Make sure deficiencies are addressed.

Conflict resolution: You will certainly have disagreements and possibly conflicts. Try to resolve them within the team. When that is not working, anyone can go to the instructor with a problem. Do not delay coming to me if things are clearly not working with the team.

18.9.5 The process

Initial:

1. Agree on roles. These roles can change if necessary, but you are encouraged to stick with them for simplicity and consistency.

2. Agree on a team name and a short no-space abbreviation if necessary, and let me know it.

3. Brainstorm about the project. Distill the ideas into a direction and overall goals.

On individual versions (Two formal versions will be required):

1. Break out specific goals for the version. How are you heading for your overall goals? Are you biting off a significant and manageable amount? You are expected to check in with me on this part and 2 and 3 before moving very far. This will be new for most of you.

2. Plan and organize the necessary parts and the interfaces between the parts.

3. Write the interface documentation for consumers of the code for the parts you plan to write. Agree on them. You need to do this eventually anyway. Agreement up front can save you an enormous amount of time! Do not let the gung-ho hackers take off before you agree on documented interfaces. We have seen it happen: If you do not put your foot down, you are stuck with a bad plan that will complicate things. Otherwise lots of code needs to be rethought and rewritten.

4. If more than one person is working on the same class, plan the names, meanings, and restrictions on the private instance variables – all coders should be assuming the same things about the instance variables! Also agree on documentation for any private helping methods you share.

5. Code to match the agreed consumer interface and class implementation designs.

6. Check each other's code.

7. Do unit tests on your own work, and fix them and test again...

8. Do overall tests of everything together, and fix and test again...

9. Look back at what you did, how it went, what you could do better, and what to change in your process for the next version.

You are strongly encouraged to follow modern programming practice which involves splitting each of these formal versions into much smaller steps that can be completed and tested following a similar process. Order pieces so you only need to code a little bit more before testing and combining with pieces that already work. This is enormously helpful in isolating bugs! This is really important. If you thought you spent a long while fighting bugs in your small homework assignment, that is nothing compared to the time you can spend with a large project, particularly if you make a lot of haphazard changes all at once.

18.9.6 Splitting Up The Coding

Make good use of the separation of public interface and detailed implementation. If your project has loosely coupled class, the main part of the public interface should be limited and easy to comprehend.

Ideally have one individual (or pair) assigned a whole class. One useful feature for allowing compiling is to first generate a stub file like we have given you for homework, that includes the public interface documentation, headings, and dummy return values and compiles but does nothing. *Post this under a box folder for the current version number.* You will then provide your team members with something that tells them what they can use and allows them to compile their own part. Then later substitute more functional classes.

Your instructor and you will want to review your code. We do not want to have to reread almost the same thing over and over: Use the editor copy command with extreme caution. If you are considering making an exact copy, clearly use a common method instead. If you copy and then make substitutions in the same places, you are likely better off with a method with the common parts and with parameters inserted where there are differences. You can make a quick test with a couple of copied portions, but then *convert to using a method with parameters for the substitutions*. Besides being a waste of effort to define seven methods each defining a tool, with just a few strings differing from one method to the next, we will require you to rewrite it, with one method with parameters, and just seven different calls to the method with different parameters. Save yourself trouble and do it that way the first time, or at least after you code a second method and see how much it is like the first one you coded....

If you are making many substitutions of static textual data, put the data into a resource file in a variation of the Fake Advise Lab.

You only want to commit working code into the shared current version folder. Comment out incomplete additions that you want to show to everyone, or comment out the call to a method that compiles but does not yet function logically. An alternative is to have a separate folder for in-process code to share for comment, so you will not try to compile it with the current working version.

18.9.7 Weekly reports

Reports are due from the report writer each Tuesday.

1. Inside your team's box folder have a subfolder called WeeklyReports. A sample stub form to fill out on the computer is in Weekly-Report.rtf. Make the name of each weekly report document be the date it was due, like Mar26.rtf. It is easy to copy the table from this week to last week and edit it to show how much your plans matched reality. You should post a version for your team to look at first. Please distinguish drafts from the final version for me to look at. You might have a separate folder Drafts, and move the report into the WeeklyReport folder when it is final. Box easily allows moving files, but not renaming them.

2. Only one report should be generated each week, with the person in the role of report writer making sure a complete version is produced and placed in the WeeklyReport folder.

3. Under plans for the next week, include concrete tasks planned to be completed, and who will do them, with an informative explanation. The content and depth of the person's work should be clear. If you can state that clearly and be brief, great. The tasks do not only include coding: they can be any of the parts listed above, and for any particular part of the project, where that makes sense. *If individuals cannot state clearly what they are working on, then the team leader and lead coder have a significant issue in their leadership that needs to be addressed.*

4. In the review of the last week (after the first week) include the last week's plans and what actually happened, task by task, concretely, with enough detail to give an idea on the magnitude of the work. This can include the portion completed and/or changes in the plans and their reasons. "Still working

on X" is not useful: Who was doing what? What methods, doing what, were completed? Which are in process? Which are being debugged? What part remains to be done, and who is it assigned to? The report writer is responsible to get a clear statement from each team member.

18.9.8 Intermediate deliverables

These materials should be placed in a subfolder **Intermediate** of your team project folder. See the due date in the schedule.

- Copy the linked stub of projectPlans.rtf document. Then complete it:

    - List the project roles again, and who ended up filling them. For coding, say who was the person primarily responsible for each part.

    - If you used old classes, like those from the skeleton project or a lab or somewhere else, say which ones are included *unchanged* or give a summary of changes.

    - If your documentation of methods is not generally done, say what classes got clear documentation (or individual methods if only some were done).

    - Where are you planning to go from here, and who you envision being primarily responsible for different parts?

- Include parts 2-4 listed below under Final Deliverables, but for an intermediate version that runs, and does *not* need to have the goal working yet. Have documentation of your methods, including summary description and description of parameters and return values. If for some reason you do not have all the documentation that you were encouraged to write *first*, at least be sure to have and point out significant examples of your clear documentation of purpose, parameters, and return values. This allows instructor feedback for completing the rest.

- The idea is to have everyone get an idea of what is expected, so we have no misunderstandings about the final version. We will give you feedback from this version to incorporate in the final version. We do not want to have to say anyone did anything "wrong" on the final version. We want to be able to concentrate on your creative accomplishments.

- Look through the list of deliverables again and make sure your collection is complete.

18.9.9 Final Deliverables

Group Submission:

One submission of the group work is due one hour before the final presentations.

1. All files listed in parts 2-5. Also include a zip file, named with your team abbreviation, containing a Windows executable with (a separate copy of) any other image and data files needed. Test to make sure you can unzip and run the executable. The final submissions will be accessible to the whole class – so we can all play them!

2. Source code. You can name the classes appropriately for the content of your game.

3. User instructions. These should be partly built into the program. The most extensive documentation may be in a document file separate from the program, if you like. (Plain text, MS Word, Rich text (rtf), or PDF, please.) The starting message built into the beginning of the game should mention the file name of such external documentation, if you have it.

4. Programmer documentation. Document the public interface for all methods in comments directly before the method heading. Add implementation comments embedded in the code where they add clarity (not just verbosity). You may have a separate overview document. Include "Overview" in the file name

5. Overall project and process review in a document named like the linked stub, projectReview.rtf.

   - The first section should be Changes. So the instructor does not duplicate effort, please give an overview of the changes from the intermediate version. What classes are the same? What features were added? What classes are new? Which classes or methods were given major rewrites? What classes had only a few changes? (In this case try to list what to look for.)

   - List again the roles, and who filled them. For coding, say who was the person primarily responsible for each part.

- What did you learn? What were the biggest challenges? What would you do differently the next time? What are you most proud of?

- How could we administer this project better? What particularly worked about the structure we set up?

6. A 10-15 minute presentation of your work to the class in final exam period. What would you want to hear about other projects? (Say it about yours.) What was the overall idea? What was the overall organization? What did you learn that was beyond the regular class topics that others might find useful to know? What were your biggest challenges? Do not show off all your code just because it is there. Show specific bits that gave you trouble or otherwise are instructive, if you like.

Look through the list of deliverables again, before sending files, and check with the whole team to make sure your collection is complete.

Your Assessment of Individuals in the Group:

This is due electronically 10 minutes after the final class presentation period, from each team member, *independently*, turned in a manner specified by your instructor, like other homework assignments.

Change the name of the linked stub file Indiv-Mem-Assessment.rtf to your teamAbbreviation-yourName.rtf. You may want to tweak it after the group presentation, but have it essentially done beforehand.

Writing this is NOT a part of your collective group deliberations. It is individual in two senses: both in being about individual team members and in being the view of *one* individual, you. For this document only, everyone should be writing separately, privately, and independently from individual experience. If you lack data on some point, say so, rather than using what others are saying.

18.10 Lab: Version Control

Modern software development requires an early introduction source code management, also known as version control. While source code management is frequently touted as being beneficial to a *project team*, it is also of great value for *individuals* and provides a clear mechanism for tracking, preserving, and sharing your work.

In addition, it simplifies the process of demonstrating and submitting your work to your instructor (and graduate assistants). Long gone are the days of carrying stuff around on USB drives and sending e-mail attachments.

There are numerous options for version control. In the free/open source community, a number of solutions have emerged, including some old (CVS, Subversion) and some new (Mercurial, Git, and Bazaar). We're particularly fond of the Mercurial system. A key reason for our choice is that there is an excellent cloud-based solution to host your projects known as Bitbucket (http://bitbucket.org).

Mercurial is set up for our labs. You can install it for your personal machine from http://mercurial.selenic.com/downloads/ .

There are other similar solutions to Bitbucket but none at present provides a completely *free* solution for hosting *private* repositories, which allow you to keep your work *secret* from others.

The basic idea is to keep a main current copy of a project at a place like bitbucket. Anywhere that you work, you can download a copy of the central version. You can add and change files. There are several layers insulating changes to local files from changes to the central repository:

- You must explicitly *add* any new file names you want the repository to track.

- Even on a tracked file, you must *commit* changes to the local repository.

- For the committed changes to get to the central repository, you must *push* them.

- You have control over what files get ignored.

Later, when you want the latest changes to the central repository to get to your site, you need to *pull* data from the central repository, and then explicitly *update* your local repository, to incorporate the new data from the central repository.

18.10.1 Goals

In this lab, we're going to learn:

1. How to create a source code repository.

2. How to add files and folders to a repository and track them.

3. How to make sure certain files and folders are not kept in the repository.

4. How to push your changes to a remote repository (at Bitbucket.org).

5. How to do your work at home and in the lab.

6. How to get our book, examples, and projects (now that that you know how to use Mercurial).

Before We Proceed

You should know that this lab is designed to be repeated as many times as needed. You can create as many repositories as you wish, subject to the limitations at Bitbucket.org, and may want to create different repositories for different needs (one for yourself, one for you and your partner in pair programming, etc.) This lab assumes you are starting with a repository for your own work. We'll include a step for how to add a *collaborator* to the project.

Future labs will talk about additional things you need to know when it comes to collaboration, so please view this lab as a *beginning* aimed at helping you to start using version control right away for your own projects.

18.10.2 Steps

Create Bitbucket.org account

We're going to begin by creating a remote repository for our work. The advantage of doing so is that we get a *hosted* repository that we can use to push/pull our work. (Unlike a dangerous stunt, you *want* to be able to try this at home, too!)

Signing up for a repository at http://bitbucket.org is easy. From the landing page, just click on the option for the *Free Plan*. This allows you to create any number of public/private repositories with support for up to 5 users. This is all you'll need for your work in this course.

Once you've created your account (and confirmed it, if required) you are good to go for the rest of this lab!

Create repository at Bitbucket

Now we'll create a first repository at Bitbucket.org.

Go to `Repositories -> Create Repository` (the option is at the bottom of the list of menu options). You'll see this screen:

You'll need to fill in or select the following options:

- Name: A short name for your project. You are encouraged to keep this simple. If you are using this for all of your work in COMP 170 (which is fine) you might name the repository after your initials. So if your name is Linus Torvalds, you could give a short name like *LinusTorvaldsCOMP170* or *LTCOMP170*.

- Repository Type: Select Mercurial. Yes, we realize that Xamarin Studio supports Git natively, but for the reasons mentioned earlier, we have chosen Mercurial. We will allow you to use Git on your own if you can figure it out and use it properly. But this lab assumes Mercurial.

- Language: You can select anything you like here. We do C# for the most part in this class, so we recommend that you select it.

- Description: You can give any description you like. If you are working with

a partner please list both you and your partner's name in the description.

- Web Site: Optional
- Private checkbox should be checked.

So just go ahead and create your first repository. You can always create more of them later.

Here is an example of a filled out form:

Set up your Mercurial commit username

If you in a place where you have a permanent home directory, like on your machine or in the Linux Lab, create a file named .hgrc in *your home directory*. Your home directory is where you are dumped when you open a DOS or Linux/OS X terminal. This is *not* inside your repository. This file must contain the following lines, with the part after the equal sign personalized for you:

```
[ui]
username = John Doe <johndoe@johndoe.com>
```

It is a convention to give a name and email address, though it does not need to match the email address you gave when signing up for bitbucket.

Creating this file saves you the trouble of having to pass the `-u username` option to `hg` each time you do a *commit* operation.

You can put this file in your home directory in Windows labs, but it disappears. You might want to keep an extra copy in your repository, and copy it to the Windows home folder when in the lab.

As a gentle reminder, your home directory on Windows can be a bit difficult to find. The easiest way is to use your editor to locate your home folder. When in the DOS prompt, you will also see the path to your directory as part of the prompt. For example, on Windows 7, you will see `C:\Users\johndoe`.

> **Warning:** To ensure that you did this step correctly, please open a *new* terminal or DOS window at this time and use the `ls` or `dir` command to verify that the `.hgrc` file is indeed present in your home directory. If it is in any other folder, Mercurial (the **hg** command) will not be able to find it–and you will receive an error.

Clone a repository from Bitbucket

Open a terminal or DOS command shell.

On Windows, the Mono shell is not appropriate. You can get a regular DOS command shell by clicking the start menu and typing `cmd` and into the text box at the bottom of the start menu, and pressing return.

In the terminal/DOS-shell navigate with to the the directory where you want to place the repository as a sub-directory. This could be your home directory on your machine or a flash drive in a lab.

Windows only: To navigate in a DOS-Shell to a flash drive, you need to enter the short command:

```
E:
```

or possible another drive letter followed by a colon. DOS drive letters are annoy-
ing because they be different another time with different resources loaded. Once
you see the proper drive displayed, `cd` to the desired directory.

If all has gone well at bitbucket, you should be able to look at your bitbucket site
and see your new repository on the list of repositories .

For example, the co-author's new repository, `gkt170`, shows up on the list of
repositories (the dropdown) as `gkthiruvathukal/gkt170`.

So you can now go ahead by selecting this newly created repository from the list
of repositories. If all goes well, you should see the following screen:
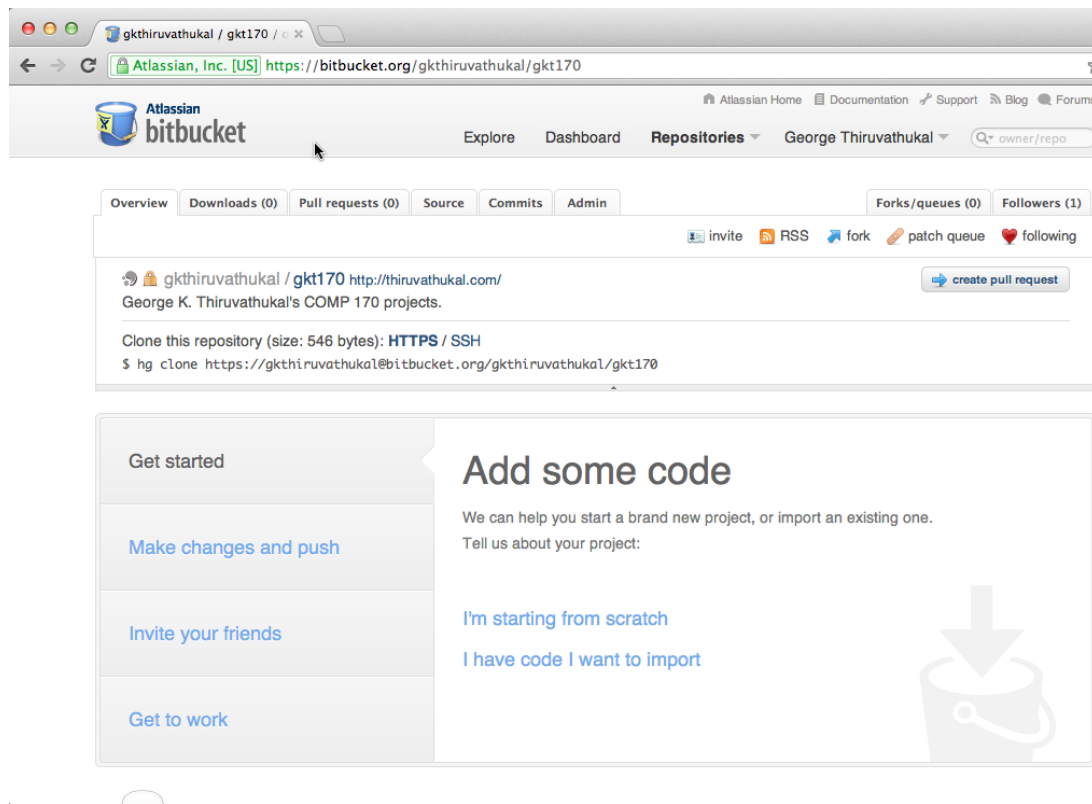
Somewhere on this screen, you should see this text:

```
Clone this repository (size: 546 bytes): HTTPS / SSH
hg clone https://yourusername@bitbucket.org/yourusername/yourrepository
```

Copy the command you see in the browser starting `hg clone`, and paste it in as a command in your terminal/DOS-shell window.

```
hg clone https://gkthiruvathukal@bitbucket.org/gkthiruvathukal/gkt170
```

You will see some output:

```
http authorization required
realm: Bitbucket.org HTTP
user: gkthiruvathukal
password:
destination directory: gkt170
no changes found
updating to branch default
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

You have created a copy of the (empty) bitbucket repository in a subdirectory named the same as yourrepository (gkt170 in the example). The is the "checkout directory", the top level of your copy.

Again, because the repository at Bitbucket is presently empty, the above output actually makes sense. There are no files to be updated. We'll learn more about what this output means later. It is possible to get *unresolved* files when you make changes that introduce conflicts. We're going to do whatever we can to avoid these for the small projects in our course work. However, when working in teams, it will become especially important that you and your teammate(s) are careful to communicate changes you are making, especially when changing the same files in a project.

> **Warning:** A version control system doesn't replace the need for human communication and being organized.

Add an .hgignore and Hello World file to your project

Change directory into the top-level directory of your local repository. That should mean `cd` to the directory whose name matches the bitbucket repository name.

The following is an example of a "dot hgignore" file. Mercurial will neither list or otherwise pay attention the files in this list:

```
# This indicates that we are using shell-like matching logic
#    instead of regular expressions.
syntax: glob
# For Mac users
Thumbs.db
.DS_Store
# This is where Xamarin Studio puts compiled stuff.
bin/
In case you compiled your own stuff, we ignore *.exe and *.dll
*.exe
*.dll
# This is a temporary debugging file generated by Xamarin Studio
*.pidb
# And one other thing we don't need.
*.userprefs
```

Here is a brief explanation of what we've included here and why:

- `syntax:  glob` indicates that uses the "glob" syntax, which comes from MS-DOS (the command prompt still found on Windows). Glob syntax allows you to do special things like match all files having a certain extension (e.g. `*.exe` matches `Hello.exe` and any other filename with extension .exe.)

- `Thumbs.db` and `.DS_Store`. Unfortunately, the Mac is still notorious for generating temporary files that serve no purpose, except on OS X. In general, we try to keep these files out of our repository and encourage you to do the same, especially if you are a Mac user.

- `*.exe` and `*.dll`. Anything that can be (re)produced by the Mono or Xamarin Studio tools should be excluded. In particular, do not keep these files in your repository. Today, they are quite small, but in future development work, they can be large. Worse, they are not text files (unlike your .cs files), so they cannot be stored optimally in a version control system.

- There are some other files produced by Xamarin Studio that we've put on the exclusion list, including `*.pidb` and `*.userprefs`. The reasoning for not keeping these is similar to that in the previous case.

Now do the following steps:

1. Using your text editor, create a file `.hgignore`. You can simply copy and paste the above contents into this file. Be careful of an editor like notepad, which adds ".txt" to the end of file names by default.

   Windows: To change from the default extension, use Save As, and change the file type from .txt by electing the drop-down menu beside file type, and select "All files".

1. Create or copy your existing `Hello World` example, hello.cs to the the `labs` folder.

2. Let's test whether .hgignore is having any effect. Go to the `labs` folder and compile the `Hello, World.` example.

3. Verify that the .cs and .exe files are in the labs directory (`ls` on Linux or OS X; `dir` on MS-DOS):

```
gkt@gkt-mini:~/gkt170/labs$ mcs hello.cs
gkt@gkt-mini:~/gkt170/labs$ ls -l
total 8
-rw-r--r-- 1 gkt gkt  224 2012-02-20 20:02 hello.cs
-rwxrwxr-x 1 gkt gkt 3072 2012-02-20 20:05 hello.exe
```

4. Check the status:

```
gkt@gkt-mini:~/gkt170/labs$ hg status
? .hgignore
? labs/hello.cs
```

What this tells us is that `.hgignore` and `labs/hello.cs` are not presently being tracked by our version control system, Mercurial. The file `labs/hello.exe` is not shown, because it's on the ignore list.

Note that we actually need to put the `.hgignore` file under version control if we want to use it wherever we happen to be working with our stuff (i.e. when we're not in the computer lab but, say, at home).

5. Add the file to version control:

```
gkt@gkt-mini:~/gkt170$ hg add .hgignore

gkt@gkt-mini:~/gkt170$ hg add labs/hello.cs
```

6. Commit the changes, and then see the log entry with the commands below. If you set the .hgrc file, the command somewhere inside your local repository could be:

```
hg commit -m "adding an .hgignore file and Hello, World to the project"
```

If you did not create .hgrc, you need also include identification with -u yourName after `commit`, as in

> hg commit -u gkt -m "adding an .hgignore file and Hello, World to the project"

It is Ok if your message wraps to a new line. You can check the log entry created by your commit:

```
gkt@gkt-mini:~/gkt170$ hg log
changeset:   0:9fe6ee1bf907
tag:         tip
user:        George K. Thiruvathukal <gkt@cs.luc.edu>
date:        Mon Feb 20 20:14:42 2012 -0600
summary:     adding an .hgignore file and Hello, World to the project
```

7. Push the changes to Bitbucket with the following command. (You'll be prompted for user/password not shown here):

```
hg push
```

You should get a response like:

```
pushing to https://gkthiruvathukal@bitbucket.org/gkthiruvathukal/gkt170
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 2 changes to 2 files
remote: bb/acl: gkthiruvathukal is allowed. accepted payload.
```

Create an initial structure for your project

We suggest that you follow a scheme similar to what we use when working with version control. We suggest that your source code goes in one or more folders, like work, or homework and labs.

So let's do it:

1. Make sure you are in the checkout directory, or `cd` to it.

2. Create directories:

```
mkdir hw
mkdir labs
```

We will be creating items in each one of these folders during the lab.

> **Warning:** Please note that most version control systems do not allow you to add *empty* folders to the repository. You must create at least one file and **hg add** it to the repository (and **hg add** and **hg push**) for the folder to actually be created. The above was just intended to make you aware of a desired "organization". You are free to organize your project any way you like as long as we are able to find your homework assignments.

Create and Test Content

1. copy in or create a simple program in a directory you created, like labs/hello.cs

2. Run it.

3. Now go back to the command prompt, and enterEnter:

> hg status

and produce a response like:

```
? labs/hello.cs
```

Mercurial shows you the tracked files that are modified (none here) or files not not being tracked (after a '?'), except for those files explicitly ignored. As you can see, the source (.cs) file is shown, but no "binary" objects (like an .exe file), since you gave instructions to ignore all such files.

4. Add the new solution/projects to Mercurial. At this point, if the above list looks "reasonable" to you, you can go ahead and just add *everything*. The

hg command makes this easy for you. Instead of adding the specific files, you can just type the following (nothing after the add):

> hg add

and produce a response like:

```
adding do_the_math.cs
```

If you inadvertently added something that you truly don't want in the repository, you can use the **hg rm** command to remove it. We have nothing at the moment that we want to remove, but want to make you aware that correcting mistakes is possible.

5. As before, commit and push. Here is a sequence from Dr. Thiruvathukal's Mac:

```
gkt@gkt-mini:~/gkt170$ hg commit -m "adding hello program"
gkt@gkt-mini:~/gkt170$ hg push
pushing to https://gkthiruvathukal@bitbucket.org/gkthiruvathukal/gkt170\
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 change to 1 file
remote: bb/acl: gkthiruvathukal is allowed. accepted payload.
```

Verify that your stuff really made it to bitbucket.org

At this point, it is entirely possible that you need some convincing to believe that everything we've been doing thus far is really making it to your repository at Bitbucket. Luckily, this is where having a web interface really can help us.

Do the following:

1. Log into bitbucket.org if you are not already logged in.

2. Go to Repositories and look for your repository. In the authors case, it is under `gkthiruvathukal / gkt170`.

3. It pays to take a quick look at the dashboard. You'll see the *recent commits* on the main screen. You should see at least two commits from our lab session thus far, both of which likely happened just "minutes ago".

4. You can click on any revision to see what changes were made. It is ok to do so at this time, but we're going to take a look at the powerful capability of "looking at the source". So go to the *Source* tab.

5. If all was done properly, you will see .hgignore and labs. These were all the result of our earlier sequence of commit+push operations. You can click on any folder to drill into the hierarchy of folders/files that have been pushed to Bitbucket (from your local repository). In labs you find your source code (for hello.cs). Then you can look at it–through the web! When you do so, you'll see something like this.

Working between lab and home (or home and lab)

It may not be immediately obvious, but what we have just shown you is how to work between the classroom/lab environment and home. In the typical scenario, when you go to your desk (or laptop), you will go through the following lifecycle:

- **hg pull**: To gather any changes that you made at another location. You are always pulling changes from the repository stored at Bitbucket, which is acting as our intermediary. Being "in the cloud" it is a great place to keep stuff without having to worry (for the most part) about the repository getting lost.

- **hg update**: To update your local copy of the repository with all of the

changes that you just pulled down from Bitbucket.

- Create or modify your folders/files as desired.

- If any files that you want included were just created, use **hg add**. It does not hurt to use this command, even if nothing was added.

- **hg commit -m message.**: Save any changes you've made, to your local repository only.

- **hg push**: Push the changes you've stashed in your local repository to the Bitbucket repository.

You might wonder why the pull/update and commit/push operations are separate. For a team of one (or two, if you have a pair), it is not likely that you'd make a mistake when coordinating changes to a central repository. In a larger team, however, some coordination is required. We're not going into all of those details in this lab, of course, but will likely revisit this topic as we get closer to the team project, which we think will make you thankful for having a version control system.

Getting our examples

We're going to conclude by taking this opportunity to introduce you to how *we* (Drs. Harrington and Thiruvathukal) are actually using the stuff we are teaching to work as a team on developing the book and examples.

1. Pick a different location (outside of your repository folder and its subfolders) to check out our stuff from Bitbucket:

```
hg clone https://bitbucket.org/loyolachicagocs_books/introcs-csharp-examples
```

2. Don't worry about breaking anything. Because Bitbucket knows what users are allowed to push changes to our repository, anything you change in your copy won't affect us.

3. Look under the source tab on the project page.

4. For example, if you performed a clone to introcs-csharp-examples, you should be able to change directory to introcs-csharp-examples:

```
gkt@gkt-mini:~$ cd introcs-csharp-examples
gkt@gkt-mini:~/introcs-csharp-examples$ ls
addition1
```

```
...
write_test
```

(Most output has been eliminated for conciseness.)

5. You can explore subfolders to see our programs.

18.11 Mercurial and Teamwork

As this course has a team project, this lecture is about how to work as a team and make effective use of the version control system, Mercurial, that we have been using throughout the semester. While the focus is on the use of Mercurial, the principles we are introducing here can be adapted to other situations (and alternate version control systems).

18.11.1 Planning and Communication

Two of the most important aspects when it comes to teamwork are planning and communication. In the real world, planning is often referred to as *project management*. And communication often takes the form of regular team *meetings*.

In later courses (e.g. software engineering) there is greater emphasis placed on thinking more broadly about software process. We're not going to cover SE in depth here but want you to be aware that software process is an important topic. Planning and communication are always supporting ingredients of a *good* software process.

At the level of actual programming, when two folks are working on the same project, it is important to think about how you can organize your work so each person on the team can get something done. As we've been learning throughout the semester, the C# language gives us a way to organize our code using projects (with Xamarin Studio). Within a project we can organize it as a collection of files, each of which maintains part of the solution to a problem. These parts are typically organized using classes within a namespace and methods.

So the key to working together–and apart–is to spend some time, initially, planning out the essential organization of a project and the files within it. Much like writing a term paper, you can create classes and methods that are needed–without

writing the actual *body* of the methods–and then commit your code to the repository. Then each member of the team can work on parts of the code and test them independently. Then you can sit together again as a pair to integrate the work you've done independently.

It's easier said than done, but this is intended as a suggestion for how to collaborate.

In any event, the above suggests that you actually need to *communicate* if you want to get anything done. You should start by discussing what needs to be done and work in *real time* to do what has been described above. Then you start coding. As you are coding, you are going to realize that as well as you planned the work to be done, you "forgot" or "misunderstood" some aspect. When this happens, you and your partner(s) need to communicate.

In the modern era of software development, we are richly blessed with synchronous communication methods such as instant messaging, texting, group chat, and other forms of synchronous collaboration. When you and your partner(s) are working on the project, we highly encourage you to keep a chat window open (Google Talk, AIM, Yahoo, IRC, whatever) and use it to communicate any issues as they arise.

18.11.2 Typical Scenario

In general, when working with Mercurial, you will find yourself using the following commands in roughly this order:

- `hg incoming`: look for incoming changes that were either made by yourself or your partner(s). We say "by yourself" because it is clearly possible that you are working on another computer somewhere else (laptop, home computer) and pushed some changes. So it is a good habit to see whether "anything has changed" since the last time you looked, even if you looked recently.

- `hg pull`: If there are incoming changes showing up as incoming, then you should in fact pull them in. This will stage the changes locally but they will not be incorporated until you type `hg update`.

- `hg update`: Absorbs all changes that were pulled from your remote repository. This operation has the potential to *overwrite* changes you are currently making, so you should make sure that the incoming changes that

you observed above are sensible. For example, if you are editing a file named `examples/MyProject/my_project.cs` and the incoming log suggests that the same file has been modified, you're likely to end up with a conflict when performing the update. (We'll cover conflict resolution shortly.)

- `hg add`: Whenever you add files to your folder, if you want them to be in the repository, you always need to *add* them. It is very easy to forget to do so. The `hg status` command can be used to figure out files that *might* need to be added.

- `hg status`: This command will become one of your best friends. You This typically tells you what has happened since you started your work in this directory (and since the last commit/push cycle). You especially want to keep on the lookout for the following:

    - `?`: This means that a file is *untracked* in the repository. If you see a file with the `?` status that is important, you'll want to add it using `hg add` as explained above.

    - `M`: This means that a file has been *modified*.

    - `A`: This means that a file has been *added*.

- `hg commit`: In general, you should commit all changes to your repository, especially before you leave the lab for the day. It is important to note that committing is a *local* operation and does not affect the remote repository (at bitbucket.org) until a corresponding push has been done.

- `hg push`: Almost immediately after a commit, it makes sense to do a push, especially if you are in the lab and will be continuing your work on other computers. In addition, as you've observed with previous homework, it is the only way to ensure that you can view the code on BitBucket (our hosting site for Mercurial projects).

18.11.3 Conflict Avoidance

Inevitably, when you are working on project, you are likely to end up in a situation where you and your partner(s) make changes that conflict with one another. In many cases, the version control software can automatically merge the changes. Here are just a few examples of where it is possible to do so:

- You make changes to different files.

- You make additions to the repository.

- You make changes to a common file that do not overlap. An example of this might be where you have two functions in your program and both you and your partner are careful not to modify them.

In general, we encourage you to coordinate your efforts, especially when you are doing something like the third situation.

Where you get into trouble is when there are changes to a common file that conflict with one another. When this happens, you have two choices in practice:

- use `hg merge` and `hg resolve` to merge your changes.

- make a copy of the conflicting files (e.g. Copy `hello.cs` to `hello.cs-backup` and use `hg update --clean` (changing your copy to match the current version in the remote repository) to just accept the latest versions of all files from the repository.

In our experience, the first option is tricky. You are given the option to perform the merge anyway or use a merge tool to select the changes of interest (and decide between them).

The second option basically results in two copies of the file. You can open up your editor to compare the files side-by-side or use a tool like `diff` on Unix or a Mac, which gives a side-by-side comparison:

```
diff -y hello.cs my_hello.cs
```

(You will need to expand the width of your console window to see clearly!)

This tool is not built into Windows. In the Windows lab, you can use the much less visually helpful

```
fc hello.cs my_hello.cs
```

to show differing segments from each file. You can also download difference display tools for Windows that are more visually helpful. One of many choices is at http://winmerge.org.

18.11.4 E-mail Notifications

One of the best ways to avoid conflicts when working on a team is to enable e-mail notification on your repository.

Bitbucket, the hosting service we are using and recommending for our students, provides full support for e-mail notification. Whenever you or your partner(s) push changes to the hosted repository, an e-mail will be generated.

These are the steps to set it up. (Owing to the changing nature of web interfaces, we are providing generic instructions that should be adaptable if the Bitbucket service decides to change its web user interface.)

1. Make sure your repository is selected. This is always the especially when you visit your repository by URL.

2. Select the administrative (Admin) tab.

3. Select Services (left-hand-side navigation).

4. Add the Email or Email Diff service. These services are basically equivalent, but one will generate links so you can view the differences that were just pushed. We recommend Email Diff.

5. Add the email notification address. You can only have one address. A good way to overcome this limitation is to set up a group service, say, at Google Groups.

18.11.5 Communication is Key to Success

At the risk of repeating ourselves, we close by reminding you of the central importance of good communication. The authors of this book communicate when it comes to their changes–even before we make them. Yet we occasionally trip over each other, and there is usually a fair amount of manual reconciliation required to deal with conflicts when we end up touching the same file by mistake.

When you absolutely and positively need to change a common file, it is important to ask yourself the important question: Shouldn't we be sitting together to make these changes? It's a rhetorical question, but working closely together, either in the same room or through a chat session/phone call, can result in significantly fewer headaches, especially during the early stages of a project.

So please take this time to stop what you are doing and communicate. You'll know your communication is good if you never need to do anything that has been described on this page. Then again, we're human. So you it is likely to happen at least once. (We know from experience but are doing everything possible to avoid conflicts in our work!)

18.12 Acknowledgments

- The Sphinx team at http://sphinx.pocoo.org for creating such an awesome documentation tool.

- The Bitbucket team at http://bitbucket.org for giving us a great place to collaborate. We are now moving to GitHub at http://github.com but still think very highly of the Bitbucket team, especially for students and faculty who need to do private projects. (We still use both.)

- Jeremy Chalmer at http://jchalmer.com/ for his past work to get a *nice* Twitter Bootstrap theme working with Sphinx. We're now using the Sphinx Bootstrap Theme project at http://loose-bits.com/2013/04/10/sphinx-bootstrap-theme-bootswatch.html but are indebted to Jeremy for inspiring us to take Bootstrap seriously.

- Speaking of Sphinx Bootstrap Theme, I wish to thank the two developers (Ryan Roemer and Russell Sim) for adding support for some missing features, especially to support shorter navigation links, links to key pages (e.g. genindex). Their tireless dedication and friendly attitude are what makes free/open source software so awesome!

genindex

[WirthADP] Niklaus Wirth, Algorithms + Data Structures = Programs, Prentice Hall, 1976.

[WikipediaShellSort] http://en.wikipedia.org/wiki/Shellsort

[uClibc] http://en.wikipedia.org/wiki/UClibc

[TCSortingJava] http://tools-of-computing.com/tc/CS/Sorts/SortAlgorithms.htm

[CamelCase] http://en.wikipedia.org/wiki/CamelCase

[UPennCSharp] http://www.cis.upenn.edu/~cis193/csstyle.html

Symbols

not use return value, 109
parameter, 94, 98
return, 100
return array, 285
summary of syntax, 117

G

generics, 351
    HashSet, 362
getter method, 379
global constant, 108
grade calculation 2 homework, 473
Grade File NUnit Exercise', 434
grade files homework, 477
greatest common divisor
    iterative, 214
    recursion, 215
greatest common divisor algorithm,
    209
grouping ( ), 37

H

HashSet, 362
    example, 363
heads or tails exercise, 252
help on command line, 465
hg, 500
Histogram exercise, 301
history, 309
history on command line, 466
homework
    booklist, 484
    grade calculation 2, 473
    grade files, 477
    grade_calc I, 467

I

IComparable Interface, 435

identifier, 45
    multi-word naming convention, 46
if, 134
    need braces, 154
    pitfall, 151
    statements nested, 158
if-else, 138
    pitfall, 153
if-else-if, 147
igame exercise, 444
immutable, 127
implication operator, 157
implicit casts in user-defined classes,
    420
indentation options in Xamarin Studio,
    52
index
    array, 282
    parallel arrays, 291
    variable not in loop heading, 297
IndexOf string method, 204
infinite loop, 195
information processing, 6
initializer, 41
InputWhole exercise, 196
insertion sort, 314
instance method, 378
instance of a class, 373
instance variable, 376
    redeclaring error, 385
instruction representation, 9
int, 37
    Parse, 60
    value range, 71
interactive while loop, 190
    repeat interactive input, 193
interface, 434
    hides actual underlying type, 437
    IComparable, 435

Z