CHEAT SHEET

This cheat sheet is designed as a way for you to quickly study the key points of this chapter.

Input validation

- Use `TrackBar`, `ComboBox`, `ListBox`, `DateTimePicker`, `FolderBrowserDialog`, and other controls to avoid validation if possible.
- Make frequent validations (such as during keystrokes) provide nonintrusive feedback (such as changing the field's background color).
- Do not trap the user in a field until its value is entered correctly.
- Remember that some values (such as "–.") may be invalid but may be part of a valid value (such as "–.0").
- When the user tries to accept a form, validate all fields. Refuse to accept the form if there are invalid values. Warn the user if there are unusual values.

Validating data—built-in validation functions

- Use string length to check for missing values.
- Initialize a `ComboBox` or `ListBox` so that it always has a valid selection.
- Use `TryParse` to validate data types such as `int` or `decimal`.
- String methods that can help with validation include `Contains`, `EndsWith`, `IndexOf`, `IndexOfAny`, `IsNullOrEmpty`, `IsNullOrWhitespace`, `LastIndexOf`, `LastIndexOfAny`, `Remove`, `Replace`, `Split`, `StartsWith`, `Substring`, `ToLower`, `ToUpper`, `Trim`, `TrimEnd`, and `TrimStart`.

Validating data—regular expressions

- Table 11-3 summarizes the useful Regex methods `IsMatch`, `Matches`, `Replace`, and `Split`.
- Use string literals (beginning with the `@` character) to make it easier to use regular expressions that contain escape characters.
- For example, the following code checks whether the variable `phone` contains a value that matches a 7-digit U.S. phone number pattern:

```
if (Regex.IsMatch(phone, @"^\d{3}-\d{4}$")) ...
```

- Table 11-10 summarizes some of the most useful regular expression components.

TABLE 11-10: Useful Regular Expression Components

ITEM	PURPOSE
`\`	Begins a special symbol such as `\n` or escapes the following character
`^`	Matches the beginning of string or line
`$`	Matches the end of string or line

continues

TABLE 11-10 *(continued)*

ITEM	PURPOSE
`\A`	Matches the beginning of string (even if in multiline mode)
`\Z`	Matches the end of string (even if in multiline mode)
`*`	Matches the preceding 0 or more times
`+`	Matches the preceding 1 or more times
`?`	Matches the preceding 0 or 1 times
`.`	Matches any character
`[abc]`	Matches any one of the characters inside the brackets
`[^abc]`	Matches one character that is not inside the brackets
`[a-z]`	Matches one character in the range of characters
`[^a-z]`	Matches one character that is not in the range of characters
`x|y`	Matches `x` or `y`
(*pattern*)	Makes a numbered match group
(?<*name*>*expr*)	Makes a named match group
`\2`	Refers to previously defined group number 2
`\k<`*name*`>`	Refers to previously defined group named *name*
`{n}`	Matches exactly `n` occurrences
`{n,}`	Matches `n` or more occurrences
`{n,m}`	Matches between `n` and `m` occurrences
`\b`	Matches a word boundary
`\B`	Matches a nonword boundary
`\d`	Matches a digit
`\D`	Matches a nondigit
`\f`	Matches a form-feed
`\n`	Matches a newline
`\r`	Matches a carriage return
`\s`	Matches whitespace (space, tab, form-feed, and so on)

continues

TABLE 11-10 *(continued)*

ITEM	PURPOSE
`\S`	Matches nonwhitespace
`\t`	Matches a tab
`\v`	Matches a vertical tab
`\w`	Matches a word character (includes underscore)
`\W`	Matches a nonword character

- Use sanity checks to look for unusual values.

Managing data integrity

- After you validate user inputs, the code must still protect the data as it is processed.
- Use `Debug.Assert` statements to validate data as it moves through the program.

Debugging

- Use the `#define` and `#undef` directives to define and undefined preprocessor symbols.
- Use the `#if`, `#elif`, `#else`, and `#endif` directives to determine what code is included in the program depending on which preprocessor symbols are defined.
- Use `#warning` and `#error` to add warnings and errors to the Error List.
- Use `#line` to change a line number and optionally the name of the file as reported in errors.
- Use `#region` and `#endregion` to make collapsible code regions.
- Use `#pragma warning disable` *`number`* and `#pragma warning restore` *`number`* to disable and restore warnings.
- The `DEBUG` and `TRACE` compiler constants are predefined. Normally, `DEBUG` is defined in debug builds, and `TRACE` is defined in debug and release builds.
- Calls to `Debug` and `Trace` class methods are ignored if the symbols `DEBUG` and `TRACE` are not defined, respectively.
- Useful `Debug` and `Trace` methods include `Assert`, `Fail`, `Flush`, `Indent`, `Unindent`, `Write`, `WriteIf`, `WriteLine`, and `WriteLineIf`.
- You can add listeners to the `Debug` and `Trace` objects. Standard listeners write messages to the Output window, event logs, and text files.

Program Database files

- You need a PDB file to debug a compiled executable.

Instrumenting applications

- Tracing means instrumenting the program to trace its progress. You can use `Debug` and `Trace` for tracing.
- Logging means recording key events. Methods for logging include writing into a text file, using `Debug` and `Trace` with a listener that writes into a text file, and writing in an event log.
- Profiling means gathering information about a program to study characteristics such as speed and memory usage. Methods for profiling include using a profiler, instrumenting the code by hand, and using performance counters.

REVIEW OF KEY TERMS

assertion A piece of code that makes a particular claim about the data and that throws an exception if that claim is false. In C# you can use the `System.Diagnostics.Debug.Assert` method to make assertions.

character class A regular expression construction that represents a set of characters to match.

conditional compilation constant A predefined symbol created by Visual Studio that you can use with the `#if`, `#elif`, `#else`, and `#endif` directives to determine what code is included in the program. These include `DEBUG` and `TRACE`, which are normally defined in debug and release builds, respectively.

data validation Program code that verifies that a data value such as a string entered by the user makes sense. For example, the program might require that a value be nonblank, that a monetary value be a valid value such as $12.34 not "ten," or that an e-mail address contain the `@` symbol.

escape sequence A sequence of characters that have special meaning, for example, in a regular expression.

inline options Options set in a regular expression by using the syntax `(?imnsx)`.

instrumenting Adding features to a program to study the program itself.

logging The process of instrumenting a program, so it records key events.

pattern A regular expression used for matching parts of a string.

performance counter A system-wide counter used to track some type of activity on the computer.

profiler An automated tool that gathers performance data for a program by instrumenting its code or by sampling.

profiling The process of instrumenting a program to study its speed, memory, disk usage, or other performance characteristics.

regular expression An expression in a regular expression language that defines a pattern to match. Regular expressions let a program match patterns and make replacements in strings.

sanity check A test on data to see if the data makes sense. For example, if a user enters the cost of a ream of paper as $1e10.00, that might be a typographical error, and the user may have meant $100.00. Sometimes the user might actually have intended an unusual value, so the program must decide whether to reject the value or ask the user whether the value is correct.

tracing The process of instrumenting a program so that you can track what it is doing.

EXAM TIPS AND TRICKS

The Review of Key Terms and the Cheat Sheet for this chapter can be printed off to help you study. You can find these files in the ZIP file for this chapter at `www.wrox.com/remtitle.cgi?isbn=1118612094` on the Download Code tab.