CHEAT SHEET

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

Working with delegates

- A delegate is a type that represents a kind of method. It defines the method's parameters and return type.
- Often the name of a delegate type ends with `Delegate` or `Callback`.
- You can use + and – to combine delegate variables. For example, if a program executes the statement `del3 = del1 + del2`, then `del3` will execute the methods referred to by `del1` and `del2`.
- If a delegate variable refers to an instance method, it executes with the object on whose instance it was assigned.
- Covariance lets a method return a value from a subclass of the result expected by a delegate.
- Contravariance lets a method take parameters that are from a superclass of the type expected by a delegate.
- The .NET Framework defines two built-in delegate types that you can use in many cases: `Action` and `Func`. The following code shows the declarations for `Action` and `Func` delegates that take two parameters:

```
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2)
public delegate TResult Func<in T1, in T2, out TResult>
(T1 arg1, T2 arg2)
```

- An anonymous method is a method with no name. The following code saves a reference to an anonymous method in variable `function`:

```
Func<float, float> function = delegate(float x) { return x * x; };
```

- A lambda expression uses a concise syntax to create an anonymous method. The following code shows examples of lambda expressions:

```
Action note1 = () => MessageBox.Show("Hi");
Action<string> note2 = message => MessageBox.Show(message);
Action<string> note3 = (message) => MessageBox.Show(message);
Action<string> note4 = (string message) => MessageBox.Show(message);
Func<float, float> square = (float x) => x * x;
```

- An expression lambda evaluates a single expression whose value is returned by the anonymous method.
- A statement lambda executes a series of statements. It must use a `return` statement to return a value.
- An async lambda is a lambda expression that includes the `async` keyword.

Working with events

- Events have publishers and subscribers. A given event may have many subscribers or no subscribers.
- Use a delegate type to define an event, as in the following code:

  ```
  public delegate void OverdrawnEventHandler();
  public event OverdrawnEventHandler Overdrawn;
  ```

- You can use the predefined Action delegate type to define events, as in the following code:

  ```
  public event Action Overdrawn;
  ```

- Microsoft best practice: Make the event's first parameter a sender object and the second an object that gives more information about the event. Derive the type of the object from the `EventArgs` class, and end its name with `Args` as in `OverdrawnEventArgs`.
- You can use the predefined `EventHandler` delegate type to define an event that takes an `object` named `sender` as a first parameter and an event data object as a second parameter, as shown in the following code:

  ```
  public event EventHandler<OverdrawnEventArgs> Overdrawn;
  ```

- Raise an event, as in the following code:

  ```
  if (EventName != null) EventName(arguments...);
  ```

- Classes cannot inherit events. To make it possible for derived classes to raise base class events, give the base class an `OnEventName` method that raises the event.
- A program can use `+=` and `-=` to subscribe and unsubscribe from events.
- If a program subscribes to an event more than once, the event handler is called multiple times.
- If a program unsubscribes from an event more times than it was subscribed, nothing bad happens.

Exception handling

- Error checking is the process of proactively anticipating errors and looking for them. Exception handling is the process of protecting a program from unexpected errors. Error checking is usually more efficient than exception handling.
- A `try-catch-finally` block must have at least one `catch` section or a `finally` section.
- The `finally` section *always* executes no matter how the program leaves a `try-catch-finally` block.
- The most-specific (most-derived) `catch` sections must come first in a `try-catch-finally` block.

- You can include only an exception type and omit the exception variable in a `catch` section if you don't need to do anything with the exception.
- The `Exception` class is an ancestor of all exception classes, so `catch (Exception)` catches all exceptions.
- A `catch` section with no exception type catches all exceptions.
- A `using` statement is equivalent to a `try-catch-finally` block with a `finally` section that disposes of the object.
- If an exception is not handled, control moves up the call stack until either a `try-catch-finally` block handles it or the program crashes.
- An exception object's `Message` property gives information about the exception. Its `ToString` method includes the `Message` plus additional information including a stack trace.
- The `SqlException` class represents all SQL Server exceptions. Properties such as `Class` and `Message` give additional information about the error.
- By default, integer operations do not throw `OverflowException`s. Use a checked block or the Advanced Build Settings dialog to make overflows throw this exception.
- Floating point operations do not cause overflow. Instead they set the result to `PositiveInfinity`, `NegativeInfinity`, or `NaN`.
- Use the floating point methods `IsInfinity`, `IsInfinity`, `IsInfinity`, and `IsNaN` to determine whether a result is one of these special values.
- To rethrow the current exception, use the `throw` statement without passing it an exception object. To rethrow the exception `ex` while resetting its stack trace, use `throw ex`.
- To return noncritical status information, a method should return a status value. To prevent a program from ignoring a critical issue, a method should throw an exception.
- If a method cannot add useful information to an exception, it should not catch and rethrow it. Instead it should let it propagate up to the calling code.
- To create a custom exception, derive it from the `System.Exception` class and end its name with `Exception`. Mark it serializable and give it constructors that match those defined by the `Exception` class.
- You can use `Debug.Assert` to throw an exception in a debug build to find suspicious data. The statement is ignored in release builds, so the program must continue even if `Debug.Assert` would stop the program in a debug build.
- If you catch an exception and throw a new one to add extra information, include a reference to the original exception in the new one's `InnerException` property.
- Clean up as much as possible before throwing or rethrowing an exception so that the calling code faces as few side effects as possible.

REVIEW OF KEY TERMS

checked By default, a program doesn't throw `OverflowException`s when an arithmetic operation causes an overflow. A program can use a `checked` block to make arithmetic expressions throw those exceptions.

contravariance A feature of C# that enables a method to take parameters that are from a *superclass* of the type expected by a delegate. For example, suppose the `Employee` class is derived from the `Person` class and the `EmployeeParameterDelegate` type represents methods that take an `Employee` object as a parameter. Then you could set an `EmployeeParameterDelegate` variable equal to a method that takes a `Person` as a parameter because `Person` is a superclass of `Employee`. When you invoke the delegate variable's method, you will pass it an `Employee` (because the delegate requires that the method take an `Employee` parameter) and an `Employee` is a kind of `Person`, so the method can handle it.

contravariant A variable is contravariant if it enables contravariance.

covariance A feature of C# that enables a method to return a value from a subclass of the result expected by a delegate. For example, suppose the `Employee` class is derived from the `Person` class and the `CreatePersonDelegate` type indicates a method that returns a `Person` object. Then you could set a `CreatePersonDelegate` variable equal to a method that returns an `Employee` because an Employee is a kind of `Person`.

delegate A data type that defines a method with given parameters and return value.

error checking The process to anticipate errors, check to see if they occur, and work around them, for example, validating an integer entered by the user in a `TextBox` instead of simply trying to parse it and failing if the value is not an integer. See also *exception handling*.

exception handling The process to protect the application when an unexpected error occurs, for example, protecting the code in case a file downloads fails when it is halfway done. See also *error checking*.

expression lambda A lambda expression that has a single expression on the right side.

lambda expression A concise syntax for defining anonymous methods.

publisher An object that raises an event.

statement lambda Similar to an expression lambda except it encloses its code in braces and it can execute more than one statement. If it should return a value, it must use a `return` statement.

subscriber An object that receives an event.

try-catch-finally block The program structure used to catch exceptions. The `try` section contains the code that might throw an exception, `catch` sections catch different exception types, and the `finally` section contains code to be executed when the `try` and `catch` sections finish executing.

unhandled exception Occurs when an exception is thrown and the program is not protected by a `try-catch-finally` block, either because the code isn't inside a `try` section or because there is no `catch` section that matches the exception.

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.