Exceptions

Exceptions are not an expensive mechanism if used correctly and sparingly. There are simple guidelines that you can follow to steer clear of the risks involved with throwing too many exceptions and incurring a significant performance cost:

- Use exceptions for exceptional conditions: if you expect an exception to happen frequently, consider programming defensively instead of throwing an exception. There are exceptions to this rule (pun intended), but in high-performance scenarios, a condition that occurs 10% of the time should not be handled by throwing an exception.
- Check for exceptional conditions before invoking a method that can throw an exception. Examples of this approach are the `Stream.CanRead` property and the `TryParse` family of static methods on value types (e.g. `int.TryParse`).
- Do not throw exceptions as a control flow mechanism: don't throw an exception to exit a loop, to stop reading from a file, or to return information from a method.

The biggest performance costs associated with throwing and handling exceptions can be categorized into several buckets:

- Constructing an exception requires a stack walk (to fill in the stack trace), which gets more expensive the deeper the stack is.
- Throwing and handling an exception requires interacting with unmanaged code—the Windows Structured Exception Handling (SEH) infrastructure—and running through a chain of SEH handlers on the stack.
- Exceptions divert control flow and data flow from hot regions, causing page faults and cache misses as cold data and code are accessed.

To understand whether exceptions are causing a performance problem, you can use the `.NET CLR Exceptions` performance counter category (for more on performance counters, see Chapter 2). Specifically, the `# of Exceps Thrown / sec` counter can help pinpoint potential performance problems if thousands of exceptions are thrown per second.