The Java™ Tutorials

Trail: Essential Classes
Lesson: Exceptions

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Unchecked Exceptions — The Controversy

Because the Java programming language does not require methods to catch or to specify unchecked exceptions (`RuntimeException`, `Error`, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from `RuntimeException`. Both of these shortcuts allow programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps the intent of the `catch` or `specify` requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any `Exception` that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and `return` value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a `RuntimeException` is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly `null`. If an argument is `null`, the method might throw a `NullPointerException`, which is an *unchecked* exception.

Generally speaking, do not throw a `RuntimeException` or create a subclass of `RuntimeException` simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Previous page: Creating Exception Classes
Next page: Advantages of Exceptions