Conditional Failures

A *conditional failure*, or *runtime check*, is any code that throws an exception if and only if a boolean condition holds. Of course, such code is prevalent in well-designed software. This page provides an overview of the common kinds of such checks.

Kinds of conditional failures

One could easily handle every conditional failure the same way: `if (!condition) throw new RuntimeException();`. But if you spend a moment to consider the nature of the check you're performing and handle it in the most appropriate way, you'll make your code more understandable, and errors easier to diagnose.

Here are the major kinds of runtime checks.

A **precondition check** ensures that the caller of a public method has obeyed the requirements of the method's specification. For example, a `sqrt` function may accept only nonnegative arguments.

A conventional **assertion** is a check that should only fail if the *class itself* (that contains the check) is broken in some way. (In some cases this can extend to the package.) These can take various forms including postconditions, class invariants, and *internal* preconditions (on non-public methods).

A **verification check** is used when you lack high confidence that an API you consume will meet its (real or implied) specification. It's easiest to understand this type of check as "like an assertion in almost every way, but we'll never want to disable them."

A **test assertion** is found in test code only, and ensures that the code under test has obeyed the requirements of its own specification. Note that this kind of "assertion" has almost nothing in common with true assertions in production code.

An **impossible-condition check** is one that cannot possibly fail unless surrounding code is later modified, or our deepest assumptions about platform behavior are grossly violated. These should be unnecessary but are often forced because the compiler can't recognize that a statement is unreachable, or because we know something about the control flow that the compiler cannot deduce.

Finally, an **exceptional result** means that the method can't provide the expected result, through no fault of its own, nor necessarily the fault of any other code involved. This may be similar to a precondition check, except that in this case the caller isn't expected to have known better. It's similar to a verification check, but failure of the dependency is not unexpected. For example, trying to read a line from a file when end-of-file has already been reached is no one's fault;

simply an exceptional result. Use a checked or unchecked exception according to the advice in *Effective Java, Second Edition* Item 58, page 244.

Summary

Kind of check	The throwing method is saying…	Commonly indicated with…
Precondition	"You messed up (caller)."	`IllegalArgumentException`, `IllegalStateException`
Assertion	"I messed up."	`assert`, `AssertionError`
Verification	"Someone I depend on messed up."	`VerifyException`
Test assertion	"The code I'm testing messed up."	`assertThat`, `assertEquals`, `AssertionError`
Impossible condition	"What the? the world is messed up!"	`AssertionError`
Exceptional result	"No one messed up, exactly (at least in this VM)."	other checked or unchecked exceptions

It's not the condition itself, it's the context

Notice that the very same condition, such as a negative employee ID, can be an "Exceptional result" in one part of the system (a user interface or system-to-system interface), while it is a "Precondition" at all public API boundaries below that point. And if the same check were ever performed for non-public method parameters for some reason, it would properly be considered an "Assertion" since our preconditions should have prevented the rogue value from getting that far. The context is what matters, and using different kinds of conditional failures conveys that context.