# Defensive Programming

# Contents

## Introduction

- Defensive programming doesn’t mean being defensive about your programming.
- The idea is based on defensive driving.
- In defensive programming, the main idea is that if a routine is passed bad data, it won’t be hurt.
- It is the recognition that programs will have problems and modifications, and that a smart programmer will develop code accordingly.

## Protecting Your Program from Invalid Inputs

- You might have heard the expression, “Garbage in, garbage out” many times in your life.
- For production software, garbage in, garbage out isn’t good enough.
- A good program never puts out garbage, regardless of what it takes in.
- A good program uses
  - “garbage in, nothing out”
  - “garbage in, error message out”
  - “no garbage allowed in”
- By today’s standards, “garbage in, garbage out” is the mark of a sloppy, non-secure program.

## Protecting Your Program from Invalid Inputs

There are three general ways to handle garbage in:

- ***Check the values of all data from external sources:***
  - When getting data from a file, a user, the network, or some other external interface, check to be sure that the data falls within the allowable range.
  - Make sure that numeric values are within tolerances and that strings are short enough to handle.
  - If a string is intended to represent a restricted range of values (such as a financial transaction ID or something similar), be sure that the string is valid for its intended purpose; otherwise reject it.

## Protecting Your Program from Invalid Inputs

- ***Check the values of all routine input parameters:***
  - Checking the values of routine input parameters is essentially the same as checking data that comes from an external source, except that the data comes from another routine instead of from an external interface.

# Protecting Your Program from Invalid Inputs

- ***Decide how to handle bad inputs:***
  - Once you’ve detected an invalid parameter, what do you do with it?
  - Depending on the situation, you might choose any of a dozen different approaches, which are described in detail later in this lecture.

## Assertions

- An assertion is code that’s used during development—usually a routine or macro—that allows a program to check itself as it runs.
- When an assertion is true, that means everything is operating as expected.
- When it’s false, that means it has detected an unexpected error in the code.

## **Assertions**

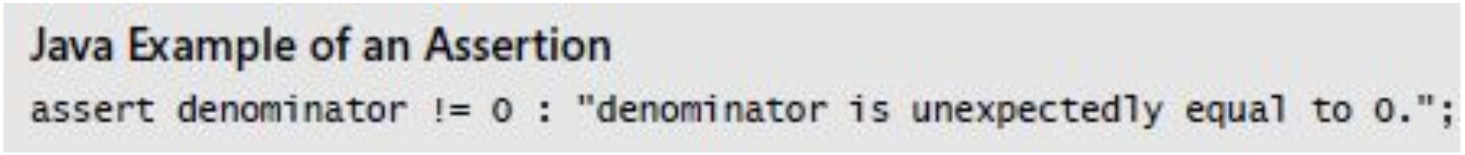
- For example, if the system assumes that a customer information file will never have more than 50,000 records, the program might contain an assertion that the number of records is less than or equal to 50,000.
- As long as the number of records is less than or equal to 50,000, the assertion will be silent.
- If it encounters more than 50,000 records, however, it will loudly “assert” that an error is in the program.

## Assertions

- An assertion usually takes two arguments:
  - A Boolean expression that describes the assumption that's supposed to be true
  - A message to display if it isn't.

# Assertions

- Here's what a Java assertion would look like if the variable *denominator* were expected to be nonzero:



Java Example of an Assertion

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

- This assertion asserts that *denominator* is not equal to *0*.
- The first argument, *denominator != 0*, is a Boolean expression that evaluates to *true* or *false*.
- The second argument is a message to print if the first argument is *false*—that is, if the assertion is false.

# Assertions

- Use assertions to document assumptions made in the code and to flush out unexpected conditions.
- Assertions can be used to check assumptions like these:
  - That an input parameter’s value falls within its expected range (or an output parameter’s value does)
  - That a file or stream is open (or closed) when a routine begins executing (or when it ends executing)
  - That a file or stream is at the beginning (or end) when a routine begins executing (or when it ends executing)
  - That a file or stream is open for read-only, write-only, or both read and write
  - That the value of an input-only variable is not changed by a routine
  - That a pointer is non-null

## Assertions

- Normally, you don’t want users to see assertion messages in production code; assertions are primarily for use during development and maintenance.
- Assertions are normally compiled into the code at development time and compiled out of the code for production.
- During development, assertions flush out contradictory assumptions, unexpected conditions, bad values passed to routines, and so on.
- During production, they can be compiled out of the code so that the assertions don’t degrade system performance.

## Assertions

- Here are some guidelines for using assertions:
  - Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur.
  - Avoid putting executable code into assertions.
  - Use assertions to document and verify pre-conditions and post-conditions.
  - For highly robust code, assert and then handle the error anyway.

## Error-Handling Techniques

- Assertions are used to handle errors that should never occur in the code.
- How do you handle errors that you do expect to occur?
- Depending on the specific circumstances, you might use any of the following approaches (Pg. 194, 195, 196)
  - Return a neutral value
  - Substitute the next piece of valid data

# Error-Handling Techniques

- Return the same answer as the previous time
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error-processing routine or object
- Display an error message
- Shut down

- A combination of these responses can also be used.

## Exceptions

- Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it.
- If code in one routine encounters an unexpected condition that it doesn’t know how to handle, it throws an exception, essentially throwing up its hands and yelling, “I don’t know what to do about this—I sure hope somebody else knows how to handle it!”.
- Code that has no sense of the context of an error can return control to other parts of the system that might have a better ability to interpret the error and do something useful about it.

## Exceptions

- Following are some suggestions for realizing the benefits of exceptions and avoiding the difficulties often associated with them.
  - Use exceptions to notify other parts of the program about errors that should not be ignored
  - Throw an exception only for conditions that are truly exceptional
  - Don’t use an exception to pass the buck

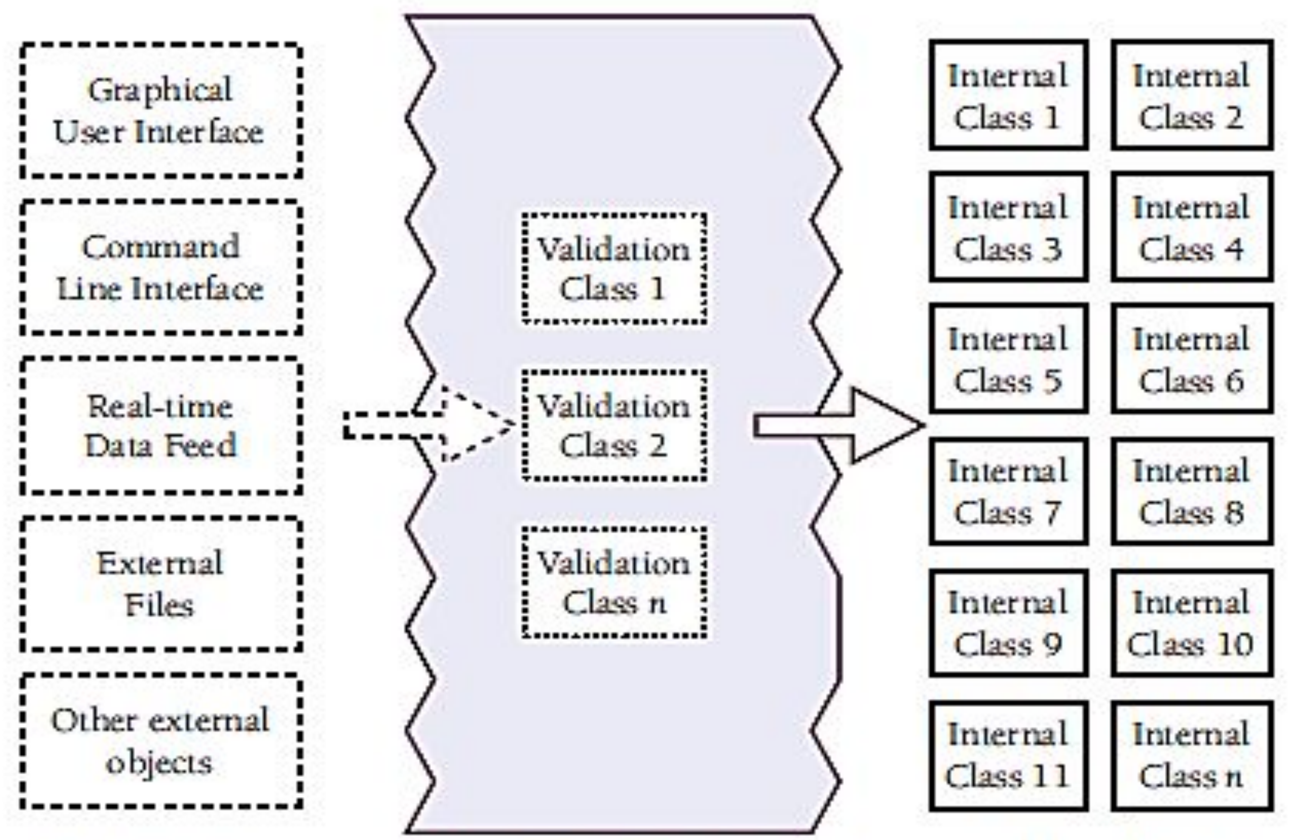
## Exceptions

- Include in the exception message all information that led to the exception
- Know the exceptions your library code throws
- Consider building a centralized exception reporter
- Standardize your project’s use of exceptions
- Consider alternatives to exceptions

## Barricade Your Program to Contain the Damage Caused by Errors

- Barricades are a damage-containment strategy.
- They are similar to firewalls in a building.
- A building’s firewalls prevent fire from spreading from one part of a building to another part.
- One way to barricade for defensive programming purposes is to designate certain interfaces as boundaries to “safe” areas.
- Check data crossing the boundaries of a safe area for validity, and respond sensibly if the data isn’t valid.

# Barricade Your Program to Contain the Damage Caused by Errors



Data here is assumed to be dirty and untrusted.

These classes are responsible for cleaning the data. They make up the barricade.

These classses can assume data is clean and trusted.

## Barricade Your Program to Contain the Damage Caused by Errors

- This same approach can be used at the class level.
- The class’s public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it.
- Once the data has been accepted by the class’s public methods, the class’s private methods can assume the data is safe.

## Relationship Between Barricades and Assertions

- The use of barricades makes the distinction between assertions and error handling clean-cut.
- Routines that are outside the barricade should use error handling because it isn’t safe to make any assumptions about the data.
- Routines inside the barricade should use assertions, because the data passed to them is supposed to be sanitized before it’s passed across the barricade.
- If one of the routines inside the barricade detects bad data, that’s an error in the program rather than an error in the data.

## <u>Determining How Much Defensive Programming to Leave in Production Code</u>

- One of the paradoxes of defensive programming is that during development, you’d like an error to be noticeable.
- But during production, you’d rather have the error be as unobtrusive as possible.

## Determining How Much Defensive Programming to Leave in Production Code

- Here are some guidelines for deciding which defensive programming tools to leave in your production code and which to leave out:
  - Leave in code that checks for important errors
  - Remove code that checks for trivial errors
  - Remove code that results in hard crashes
  - Leave in code that helps the program crash gracefully
  - Make sure that the error messages you leave in are friendly

## Key Points

- Production code should handle errors in a more sophisticated way than “garbage in, garbage out.”
- Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.
- Assertions can help detect errors early, especially in large systems, high-reliability systems, and fast-changing code bases.

## Key Points

- The decision about how to handle bad inputs is a key error-handling decision and a key high-level design decision.
- Exceptions provide a means of handling errors that operates in a different dimension from the normal flow of the code. They are a valuable addition to the programmer’s intellectual toolbox when used with care, and they should be weighed against other error-processing techniques.

## Key Points

- Constraints that apply to the production system do not necessarily apply to the development version. You can use that to your advantage, adding code to the development version that helps to flush out errors quickly.

# Readings

- **[Chapter 8]** Code Complete: A Practical Handbook of Software Construction by Steve McConnell, Microsoft Press; 2nd Edition (July 7, 2004). ISBN-10: 0735619670