First, Do No Harm:
A Curricular Approach to Exceptions

Duane Buck
Otterbein University
Westerville, Ohio
614-802-1775
dbuck@otterbein.edu

ABSTRACT
This paper advocates the adoption of deferred error coding within computer science curricula. It argues that it is both a sound development strategy and aligns well pedagogically. By deferring specific error handling, the student better appreciates its subtleties and its importance as an independent topic. This paper also includes other topics which may enhance curricula: taxonomies of exceptions and exception handlers and the relationships between them, subtle pitfalls of exception handling, and factors influencing the selection of error reporting patterns. Much of the discussion is language independent, but specific attention is given to the Java checked exception controversy.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Error Handling and Recovery

General Terms
Algorithms, Reliability, and Languages.

Keywords
Java, Checked Exception, Refactoring.

1. INTRODUCTION
When executing an application, there is a path through the application’s instructions as one service after another is requested of the underlying application programming interface (API) and fulfilled. Ultimately this results in the application providing one of its functions. This is referred to here as a “direct-path.” However, some API requests may not be satisfied, for various reasons, and require alternative processing which is not considered to be part of the direct-path.

The term exception is used broadly here to refer to a failed request. Exceptions fall into one of two broad categories, expected and unexpected. Expected exceptions represent circumstances that are unavoidable and should be planned for; unexpected exceptions typically indicate program bugs, which should not occur in a production application. Expected exceptions require specific alternative processing, which depends on their foreseeability and a cost/benefit analysis. An unexpected exception also requires alternative processing, but the response is limited: an appropriate shutdown that supports debugging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ITICSE’13, July 1-3, 2013, Canterbury, UK.
Copyright 2013 ACM X-XXXXXX-XX/XX/X…$10.00.

In older languages, such as C, exceptions were indicated by API request return codes, and a great deal of the source code dealt with checking the codes, which was error prone and hampered reliability. There was a major advance when newer languages, such as C++, introduced modern exception handling. [3] With this facility, it is possible to code the direct-path without explicitly coding an action after each call to the API. For unexpected errors, nothing needs to be, or should be, coded. This is possible because unexpected exceptions may be addressed by a context appropriate default handler (sometimes called the uncaught exception handler). 1 Therefore, the programmer has only to explicitly catch exceptions that are expected and code the specific alternative processing required.

This outcome is ideal except for the problem that it is up to API designers to document expected exceptions, and up to the programmer to read the documentation and implement the specific alternative processing required in the context of the application. Otherwise, unless the need to handle an exception is uncovered during debugging, a system could go into production missing a handler; if the exception occurred, it would be treated as a bug. The designers of Java therefore attempted to further improve reliability by supporting a second type of exception for problems arising “outside of the immediate control of the program” (equivalent to expected exceptions, as defined here). When using checked exceptions, the lack of an explicit exception handler is statically detected at compile time.

The designers of Java realized their invention could improve reliability only if programmers followed through on the information it provided. To ensure this, they compelled programmers to explicitly code alternative processing for each checked exception by classifying the failure to “catch or specify” it in every method where it could arise, rather than generating a warning which might be ignored. Paradoxically, as explained below, this creates an environment that tends to result in applications having more serious flaws than those checked exceptions were designed to address. 2 [1, 2, 10]

There has been a long standing controversy regarding checked exceptions. [2, 5, 10] It will be argued here that although checked exceptions are useful, the error classification of failure to “catch or specify” is problematic. This is because: (1) “catch or specify” is not always possible, (2) for some applications “catch or specify” is unnecessary or even undesirable, and (3) when “catch

1 The use of a default handler is described by Longshaw and Woods [6, p. 40] as the “Big Outer Try Block Pattern.”
2 That checked exceptions protect the system from crashing due to an uncaught exception is a myth that has emerged to justify them. It has a basis in reality; early software environments did crash for that reason. Footnote 17 debunks this myth.
or specify” is strictly enforced, it invites the dysfunctional practice of coding specific exception handling before initial debugging.

The first two issues are discussed briefly within this manuscript. The primary issue to be addressed here is the third: the dysfunctionality of developing specialized exception handlers before initial debugging (i.e., simultaneously with coding the direct-path). In this case, the error handling code: (1) potentially masks bugs during debugging, (2) adds to the volume of code initially being debugged, (3) often requires a design scope larger than the method being coded, (4) competes for attention with coding the (usually more interesting) direct-path, (5) requires maintenance because the code-base may not yet have stabilized, (6) is developed without the insights gained from first-hand experience with the exceptions, and (7) requires a specialized skill set which the implementer of the direct-path may not possess. This manuscript is the first to identify these issues. They point strongly to the practice being dysfunctional. This may not have been recognized because it was not employed; the natural tendency is to develop the direct-path before the edge-cases. [10]

Because of the issues encountered when developing specific error handlers prior to initial debugging, as well as for pedagogical concerns, this manuscript proposes a two phase development strategy: deferred error coding. This emerges organically in most languages, but requires an explicit coding technique when using Java. When a curriculum uses another language, deferred error coding remains important because of the pedagogy it enables.

The primary focus of this paper is on how deferred error coding may be beneficial in the computer science curriculum, but it also presents background material and important related topics which, in appropriate courses, may be a useful supplement to textbooks. Section 1 gives the motivations. Section 2 introduces taxonomies of exceptions and exception handlers, and relates the two. Section 3 discusses error reporting patterns in Java. It includes the description of a hybrid approach found in some newer Java library classes. Section 4 points to the difficulties with exception classification in Java. Section 5 presents the “deferred error coding” curriculum approach where the application’s direct-path is initially debugged before its error handling is refactored. Its implementation using Java is also addressed there, together with common exception handling pitfalls to be avoided. Section 6 provides a different point of view as it examines the design of error reporting within an API. Section 7 presents the conclusion where it is argued that the checked exception controversy may be resolved. It then presents a minimal curricular change that could by itself improve checked exception handling, based on “First, do no harm.” Section 8 provides the references cited.

2. EXCEPTION AND HANDLER TYPES

An exception is triggered by one of four conditions. It may be (1) an as yet undetected program bug, (2) a system error (e.g., out-of-memory), (3) an environment fault (e.g., a network outage), or (4) an issue related to the application domain (e.g., incorrect input). The order here is from least likely to most likely to be “expected” in the sense that the potential error is understood well enough that a specific alternative action could be developed. The first two situations are generally classified as unexpected, and the latter two expected. Note that some situations (especially application domain errors), although not reported by API exceptions, might still require “handling,” perhaps by throwing an application defined exception.

It is beyond the scope of the present paper to discuss exception handling comprehensively. The paper by Chen et al. [1] has an excellent discussion of refactoring exception handling, which this paper draws upon. The broad issues are discussed here. There are three major categories in the taxonomy of exception handlers:

(1) A message/terminate handler is provided by the uncaught exception handler, which assumes the exception is due to a bug. It may be customized for the execution environment as discussed earlier. Occasionally, a message/terminate handler is specifically coded for an unrecoverable situation not resulting from a program bug. In this case, the message would be tailored to communicate with the end-user.

(2) A message/rollback handler is used in the case where a single request could not be completed, but the system may be capable of completing other requests. The request is often an action requested from a user-interface. The handler informs the user, and then needs to transfer control back to the “event loop” so the user can make additional requests. The difficulty with this type of handler is that it must ensure that the partially completed execution of a request does not invalidate further execution of the application. Borrowing from database terminology, the transaction must be rolled back.

(3) A retry/fallback handler first tries to complete the function of the method invocation, which may involve attempting the same action again and/or executing an alternate implementation. Usually after some number of failed tries, it falls back to a message/rollback or message/terminate handler, depending on the context and the severity of the issue.

Table 1 shows for each exception type the types of handlers that may be employed. The shaded cells represent unusual handlers for that particular type of exception. Exceptions due to environmental faults are the only ones that do not have a usual handler type. For those, the choice of handler should be based on a cost and benefit analysis within the particular context. The retry/fallback provides the best user experience, but also has the highest development cost. If retry/fallback is not feasible or not cost justified, then message/rollback should be considered if providing a subset of “commands” is possible, as this provides the next best user experience. If that is not feasible or not cost justified, then the least desirable, but also least expensive, message/terminate handler is the only other choice.

The other types of exceptions each have a usual handler type. Program bugs are corrected rather than specifically handled, and are reported by a message/terminate handler, usually the uncaught exception handler. System errors typically indicate that the system is unstable, and therefore one should not attempt the complex endeavor of addressing the issue, so they are also usually handled with a message/terminate handler, which may again be the uncaught exception handler. This is a reasonable choice because system errors are sometimes triggered by bugs. Specific

---

3 The author delivered an invited presentation at JavaOne 2012, San Francisco, discussing this and other topics explored below.

4 Haase’s “Unhandled Exception” pattern [4, p. 105] is similar to the deferred error coding recommended here.

5 When debugging, a default handler usually supplies debugging information and terminates the activity in progress. Because this behavior is inappropriate for an end user, a custom default handler is usually installed when running in production.
handlers are usually coded for environment faults and application domain exceptions, and may implement recovery, if appropriate.

<table>
<thead>
<tr>
<th>Type of Handler</th>
<th>message/terminate</th>
<th>message/rollback</th>
<th>retry/fallback</th>
</tr>
</thead>
<tbody>
<tr>
<td>system error</td>
<td>Report debugging information, and inform IT and the user.</td>
<td>Continued execution is risky. Occasionally, system errors (e.g., out-of-memory), may be understood well enough to attempt one of these handlers.</td>
<td></td>
</tr>
<tr>
<td>environment fault</td>
<td>Report to IT. Inform user.</td>
<td>Report to IT. Inform user of the issue; continue w/o repair.</td>
<td>Attempt retries and/or alternate implementation; then terminate or rollback.</td>
</tr>
<tr>
<td>application domain error</td>
<td>Inform user.</td>
<td>Inform user and allow them to try again.</td>
<td>N/A; same input will get same result.</td>
</tr>
</tbody>
</table>

3. JAVA EXPECTED ERROR REPORTING

3.1 Return Codes vs. Checked Exceptions

After he extensively reviewed the literature of error handling and recovery, Tellefsen [9, p. 50] concluded that “return codes are useful for returning error information, simply because they are easier to use, and they would probably be used even if they were disallowed by project guidelines.” It is therefore not surprising that return codes remain in use today in the Java libraries to report expected errors, even though return codes are problematic.

A return code often takes the form of a single return value being multiplexed so it is either a result or a status indicator. The Java library `Map` class’ `get()` method is an example of this. It returns an object reference in the normal case, otherwise it returns `null`. An application programmer may find return codes beneficial because the `if/else` construct is familiar and easy to code. However, they need to be cognizant of the danger of not checking a return code and losing the source of an error. This is more likely to happen with a multiplexed return code, because it is tempting to code the function inside of another expression, assuming no error will occur. Fortunately, a null reference or negative index value frequently results in a quick exception.

An interesting juxtaposition occurs here. The Java library makes use of return codes for some expected errors even though failure to check return codes is a major issue affecting debugging and reliability. They do this seemingly because of the difficulty programmers have coding exception handlers. Meanwhile, Java forces coding explicit handling of checked exceptions before compilation and debugging can commence. For expediency, programmers tend to take shortcuts to a compilation: they ignore return codes and insert the minimal code required to ignore checked exceptions. The result is that execution continues for both error reporting mechanisms, complicating debugging, and potentially leading to unreliable applications.

3.2 Hybrid Error Reporting

For expected errors, when external events cannot asynchronously alter the validity of a request, an API may supply a method to check the validity of a request before it is made. This provides the best qualities of return codes and unchecked exceptions without their drawbacks. Some of the newer Java APIs (e.g., `Scanner`) use validity requests. An API documents an expected error, not with a checked exception, but with a method to precheck validity. The programmer uses the familiar `if/else` construct to code the alternative action, as with return codes. If the programmer fails to do the validity precheck, an exception signaling the expected situation will (hopefully) occur during testing. This gives a meaningful stack-trace pointing to the problem. For some APIs, the programmer may choose to catch the exception instead of using the query method, when that makes error coding easier.

4. JAVA EXCEPTION CLASSIFICATIONS

Although unanticipated issues arose with the forced early implementation of checked exception handlers, one might expect that the anticipated benefit, knowing that the correct exceptions have specific handlers, is enjoyed. However, the classifications of exceptions have proven idiosyncratic and guidelines have shifted.

In the Java library, some exceptions, which for all practical purposes are expected, are confusingly classified as unchecked, and vice-versa. For example, consider the familiar library function `int Integer.parseInt(String s)` which converts the input `s` to an `int`. The origin of the input is almost certainly from outside of the program (probably an end-user), so it would be expected to occasionally be incorrect. However, `parseInt()` throws an unchecked exception when given invalid input. An apparent misclassification of this kind, where an expected exception is classified as unchecked, defeats the purpose for which checked exceptions were designed.

The opposite problem is also troublesome because checked exceptions may unnecessarily complicate the use of an interface. The guideline published in the quasi-official Java Tutorial regarding the use of checked exceptions has shifted from virtually mandating their use when not reporting bugs, [2] to using them when a client can “reasonably be expected to recover,” which better focuses on their purpose. [5] This was apparently a workaround addressing the “unnecessary complication” argument.

What is worse than the preceding issues is that expected and unexpected situations are sometimes merged into a single checked exception class when reported. This is the case with `java.io.IOException` which is thrown by the `read()` methods of several IO classes. If an expected error occurs (like losing the connection to a network resource, which is outside the control of the programmer), a checked exception is properly thrown. However, `IOException` is also thrown if the IO object is closed before the `read()` method is invoked. The latter is inexplicable because clearly such a situation should be reported.

---

6 Alternatively, an API may provide a separate method to access the return code (e.g., `Scanner`; see Footnote 16 below).

7 If the `Integer` class API provided a function to check a `String` for valid integer syntax, it would be correct to consider passing an invalid string to `parseInt()` unexpected. This would follow the hybrid error reporting pattern given above.
by a subtype of RuntimeException indicating a program bug (the canonical choice would be IllegalState Exception in the java.lang package). The unfortunate result is that parsing the exception’s message is the only alternative available to determine the cause of the exception.8

Some Java library APIs (e.g., java.sql) and third-party APIs classify all of their declared exceptions as checked, even if they are due to programming bugs, although this goes against the published guideline. This misclassification may be due to esthetic concerns of the API’s designers, who want to have their exceptions under a single supertype for their entire API, which makes it impossible for some of the exceptions to be checked and some unchecked.9 A similar esthetic may also be behind the unusual choice of using IOException to report a bug.

5. CURRICULUM IMPLICATIONS
Error coding is an important but complex topic that deserves attention in the curriculum. Having the students first learn direct-path implementation without the complexity of error coding is important to avoid covering too many topics at one time. Returning to existing code and refactoring it to include specific error handling as a separate step allows the student to focus their attention on that topic. In fact, two phase development is generally advantageous, as discussed in Section 1. It is especially important for pedagogy in order to avoid cognitive overload.

When introducing deferred error coding, the motivations presented in Section 1 may be a useful topic. When teaching refactoring, the taxonomies and their relationships examined in Table 1 would be a resource, as would be the dysfunctional examples and their alternatives in Subsection 5.2. During later courses, the API factors to be discussed in Section 6 may also be a topic of interest. The remainder of this section examines the two phases of the deferred error coding pedagogy advocated here.

5.1 Direct-path with Termination upon Error
As previously discussed, when using languages other than Java, deferred error coding is organic, and curricula have implicitly embraced it (at least for exceptions). When using Java in a curriculum, a technique for implementing deferred error coding must be taught. Although the technique recommended here adds more verbiage than desirable, it imposes the least cognitive load among the available choices. The student is instructed to insert the following “boilerplate” template around any method invocation that throws a checked exception:

```java
try {
    aMethodThrowingACheckedException();
} catch (ACheckedException ex) {
    throw new RuntimeException(ex);
}
```

This handler will trigger the uncaught exception handler which should be a context appropriate message/terminate handler.10 A program with this boilerplate handler has a valid form of error handling, but it might provide a suboptimal user experience.

Although not discussed further here, deferring specific error coding until refactoring also applies to the other API error reporting patterns: conventional exceptions, requests with validity query methods, and return codes. For return codes, deferred error coding requires inserting code to throw a runtime exception in the event of an error,11 similar to the boilerplate code for checked exceptions discussed above. Like deferred coding for checked exceptions, deferred error coding for the other API error reporting patterns allows the direct-path to be coded expediently, yields good debugging information, and provides a foundation for the refactoring that follows.12

There are several advantages to using the deferred specific error coding approach. Then starting out, the student is taught an expedient approach that is not dysfunctional. Early on the student will see in which contexts things can go wrong and trigger exceptions. The student also comes to understand that a program without specific error handling, for at least domain level exceptions (user errors), is not “finished.”13 Later the student will learn how to refactor their application to create a robust solution.

5.2 Refactoring Error Handling
By refactoring error handling, the students are not forced to divide their attention between the direct-path, which is their central concern initially, and error handling. Advanced assignments will require students to refactor the error coding. This may involve both studying the API and testing to determine which exceptions are recoverable in the context of the application. For those, the student will code a specific alternative action. Each location where boilerplate code throws a RuntimeException needs to be studied to determine if it should to be refactored to a more specific handler. As discussed earlier, testing is also required because some methods throw misclassified unchecked exceptions that should be caught and vice-versa. Once the type of handler is selected, the student may need to include multiple lines of code in a try/catch block, possibly need to use a throws clause to send an exception to the invoking method, and might have the need for try blocks with a finally clauses to release resources14 when non-terminating handler is invoked.

It is beyond the scope of this manuscript to cover implementation of error handling in detail. Instead, some practices whose dysfunctionality may not be apparent will be enumerated. The examples are drawn from textbooks, the standard Java library, and an Eclipse code template, and probably arose under the influence of checked exceptions. Each example is immediately followed by an alternative that addresses the problem cited. Unfortunately the authors, who will remain anonymous, appear to be oblivious to the issues raised.

---

8 A returned error status becomes a RuntimeException:

```java
<result> = aRequestWithReturnCode();
if(<result-indicates-failure>)
    throw new RuntimeException(<message>);
```

9 When all errors result in a runtime exception (known as failing-fast), a code base has reached the first goal (G1) of the error handling refactoring methodology presented by Chen, et al. [1]

10 A mathematics colleague points out that for his purposes it is finished because an exception handler might mask an error and cause erroneous output, which is far more troublesome than rerunning. This is probably true of many one-off programs.

11 Every method given a throws clause is a candidate for this.
5.2.1 Ignored Checked Exception
The following ignores a checked exception that is unexpected:

```java
try {
    Thread.currentThread().sleep(10);
} catch (InterruptedException e) {}
```

In this case, ignoring the exception seems innocuous. Unless the application uses cooperating threads, and invokes the current thread's `interrupt()` method, theoretically the exception will not occur; if it does, it indicates a bug. However, this code ignores it and proceeds. Although ignoring an expected exception might be a reasonable fix-up, one should never ignore an exception that is unexpected; Müller and Simmons [7, Subsections 2.1 and 4.2] provide an extended discussion.

**Alternative:** To handle an unexpected checked exception, simply retain the boilerplate code discussed above. To document refactoring is completed, `RuntimeException` may be replaced by an application subtype, e.g., `UncheckedException`:

```java
try {
    Thread.currentThread().sleep(10);
} catch (Exception e) {
    throw new UncheckedException(e);
}
```

If the exception occurs, it will properly be reported as a bug.

5.2.2 Noting and Ignoring a Checked Exception
Examine the following code that is creating an InputStream:

```java
try {
    is=new FileInputStream(f);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Here the code reports the error, but does so in a way that is insensitive to the user currently executing the program (perhaps it is the end-user). Printing to the console is also problematic because applications are typically deployed without console windows. The bigger problem here is that the program keeps running and its results will be unpredictable. Unfortunately, this is similar to Eclipse’s default code template that assists coding the invocation of methods that throw checked exceptions.

**Alternative:** The standard boilerplate code is far superior to the above. However, because this error is expected (the existence of the file is not under the control of the programmer), a message/rollback handler is probably indicated when refactoring.

5.2.3 Fix-up of an Exception Triggered by a Bug
This code reading InputStream is has subtle issue:

```java
try {
    b=is.read();
} catch (IOException e) { b=0; }
```

The error handler performs a simple fix-up to an environment fault, using a default value and continuing. But, as discussed, the problem is that unexpected and expected situations have been merged into one exception class by the API designer. It might signal an IO error, which this handler addresses, but another possible cause of the exception is that the InputStream has been closed, a program bug. In that case, the application will continue execution and make it difficult to locate the error. This mixing is common in some library classes (e.g., `java.sql`).

**Alternative:** The boilerplate handler should be augmented to examine the exception, verify it was not caused by a program bug, and if so, execute specific handling. Otherwise, it will throw the boilerplate `RuntimeException` to invoke the default handler.

5.2.4 Supertype Exception in throws Clause
The elided method given below throws to a method on the stack a checked exception which is a supertype of other exceptions:

```java
void fun1() throws IOException
{
    b=is.read();
...
```

This code is dysfunctional because the throws clause throws all subtypes of `IOException`. As a result, a programmer opening a file within the same method will not be required by the compiler to handle the `FileNotFoundException`, for instance.

**Alternative:** A supertype checked exception should be caught locally. If it cannot be dealt with locally, it should be wrapped and thrown in an application defined exception:

```java
void fun1() throws IOExceptionWrapper
{
    try {
        b=is.read();
    } catch (IOException ex) {
        throw new IOExceptionWrapper(ex);
    ...}
```

The above throws clause names the class wrapping the supertype exception, so the invoking method will have to “Catch or Specify” `IOExceptionWrapper` in this case. Now, when a programmer opens a file, they will be required to catch or throw `FileNotFoundException`.

Fortunately, supertype exceptions are infrequently thrown in library APIs. The bigger lesson is that generally, only specific subtypes should be caught, or appear in a throws clause; if a supertype is specified, subtypes will not require specific handling.

6. API DESIGN IMPLICATIONS
The underlying problem to be solved in an API is how to give feedback to the application regarding an action the application requests or plans to request. As discussed earlier, some form of return code may always be with us even though using return codes is problematic. The use of return codes is justified for especially common situations, like a failure searching for a substring.

Although enforcing specific error coding at compile time through checked exceptions may appear beneficial, as has been discussed extensively, doing so tends to encourage dysfunctional error coding and should be used with caution. Additional concerns have also been raised. Robillard and Murphy [8, p. 2] discuss how coding using the checked exception mechanism tends to lead to “complex and spaghetti like exception structures” (e.g., the tunneling scenario to be discussed in Section 7). Another concern with checked exceptions is noted by Haase at the end of his summary: “The benefits of checked exceptions can be summarized by saying that their use provides documentation and ensures that exceptions are handled. There is however a downside to this, namely that checked exceptions reduce flexibility.” [4, p. 94] For example, when an application being modified needs to invoke a method that throws a checked exception, it is not easily accomplished. Either the call hierarchy must be modified to the point the exception is handled, or it must be tunneled.

The above concerns about checked exceptions are serious and recognized by the wider software community. The designers of the post Java language C# chose not to include checked
exceptions [10], and according to Chen et al. [1, p. 335] “unchecked exceptions are preferred in several well-known open source projects written in Java, including the Eclipse SWT project and the Spring Framework.”

A better alternative to both checked exceptions and return codes may be to provide hybrid error reporting as discussed in Subsection 3.2. Using a separate query method, an application can evaluate a request's validity. If the request is valid, the application can make the request and the proper outcome is guaranteed. If the programmer fails to make the check, and an invalid request is made, a runtime exception will be thrown, triggering the default handler.\textsuperscript{15} Using a validity query for each type of request, programmers employ the if/else construct, with which they have vast experience. This makes the query style easier to code and read for many application programmers. The designers of the \texttt{Scanner} class used query methods.\textsuperscript{16} Because that class is a recent addition to the Java library, its designers may have called upon experience to point to that solution.

7. CONCLUSION

Java checked exceptions, although in theory beneficial for reporting expected exceptions, have created a problem in the curriculum. They distract the student from the central function of their project, and force them to reason about constructs they may not yet understand. The recommendation made here is to have students follow the two phases of “deferred error coding.” The first phase implements the direct-path and keeps the code base behaving in a predictable manner. As the student gains more insight, he or she will then enter the second phase, refactoring the error handling.

The issues raised about checked exceptions are almost exclusively due to classifying failure to “catch or specify” an error. If the compiler instead generated a warning, an application could be debugged without explicit error handling, and during refactoring the warnings could be used to locate checked exceptions not handled. Issuing warnings also solves the other major issue: the need to throw checked exceptions across foreign software boundaries. Rather than wrapping it inside a runtime exception and tunneling it across the boundary, one would expect a warning. A warning would also be issued for the method handling the exception while invoking the foreign software. This would verify that the exception is caught. Ultimately, one might add annotations to suppress the warnings, similarly to when generics are not statically verifiable. Also, because checked exceptions will no longer impose an unnecessary burden, the guideline for an exception being checked can return to the its being “outside of the immediate control of the program.” This is simpler than the workaround guideline, which is that the exception be checked if the client may “reasonably be expected to recover,” because the latter requires an API designer to make assumptions. Technically, it is apparent that failure to “catch or specify,” could easily be designated a warning,\textsuperscript{17} which would resolve the major issues that have arisen in the controversy.

Even if the central recommendation of this manuscript – deferral of specific error coding within the curriculum – is not adopted, a significant benefit will follow from discussing with students the dangers involved when handling checked exceptions. The Hippocratic Oath includes “First, do no harm,” which is good advice in this context. The student should be instructed to first code using the standard boilerplate template presented in Section 5 whenever they encounter a checked exception that they either: (1) think will not occur, or (2) are unsure how to handle. This will take little class time and will significantly reduce the level of dysfunctional error handling during debugging, which may help provide the additional insight the student needs.

8. REFERENCES


\textsuperscript{15}The programmer might catch the exception instead of using a query method if it simplified the implementation.

\textsuperscript{16}It is interesting that \texttt{Scanner} has a little known “return code retrieval” method, \texttt{ioException()}, which returns the \texttt{IOException} last thrown by the \texttt{Scanner}'s underlying \texttt{Readable}, or \texttt{null} if no such exception exists. It apparently was added to relieve the client from having to deal with \texttt{IOException}. The programmer should check that the method returns \texttt{null} before executing a user action. Otherwise, because an \texttt{IOException} is treated as end-of-file, an unintended action may result. Many users may unknowingly use applications that have this bug. \texttt{Scanner}'s handling of the checked exception provides a good case study of how checked exceptions, paradoxically, may negatively impact reliability.

\textsuperscript{17}To confirm this, note that when generics are used, in some cases the compiler is unable to detect the failure to “catch or specify” a checked exception; it appears checked exceptions are like generics in that the “check” is made at compile time, and they require no additional runtime support. Also see Footnote 2 for a myth “explaining” why checked exceptions must be explicitly handled.