Unhandled exception has occurred in your application. If you click continue the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Details  Continue  Quit

Exception Handling

Ibrahim Albluwi
Warm Up Question!

From a Software Engineering point of view, what is wrong with the following implementation of the push method in an implementation of the Stack ADT?

```java
public void push(Item item) {
    if (item == null)
        System.out.println("Can't push a null element.");
    else if (n == a.length)
        System.out.println("Can't push to a full stack.");
    else
        a[n++] = item;
}
```

What if the stack is used in a GUI application?

How can the calling method know if a push is successful?
Candidate Solution: Error Codes

- Stack methods **detect** errors, but do not **handle** them.
- Stack methods return information about errors in the form of **error codes**.

**Advantages:**

- Stack implementation is not tied to any certain error reporting mechanism.
- Clients of a Stack are well-informed about errors.

```java
public int push(Item item) {
    if (item == null)
        return -1; // Error code for null items
    else if (n == a.length)
        return -2; // Error code for full stack
    a[n++] = item;
    return 0;    // Success code
}
```

Any problems?
Disadvantages:

- Code may be **difficult to read, debug and maintain**:
  - -1, 0, 1, etc. are not inherently meaningful.
  - The same error code can have different meanings in different methods.
  - Error codes need to be manually propagated from one method to another.

```java
int err1 = mysStack.push(myItem);
if (err1 == -1)
    err1 = mysStack.push("Non-NULL String");
int err2 = doSomething();

if (err1 == -2 && err2 == -1)
    return -1;
else if (err2 == -2)
    return -2;
else
    return -3;
```
Disadvantages:

- Code may be *difficult to read, debug* and *maintain*:
- API limitations need to be worked around!

```java
public Item peek() {
    if (isEmpty()) return ???;
    return a[n-1];
}

public int peek(Item[] result) {
    if (isEmpty()) return -1;
    result[0] = a[n-1];
    return 0;
}

public Item peek() {
    if (isEmpty()) {
        internalErrorFlag = -1;
        return new Item();
    }
    internalErrorFlag = 0;
    return a[n-1];
}
```
Candidate Solution: Error Codes

Disadvantages:

- Code may be *difficult to read, debug* and *maintain*:
- API limitations need to be worked around!
- Clients may not check for error codes.
  - Experience shows that error codes are often ignored!
  - Behavior is undetermined if errors are not accounted for.
Candidate Solution: Error Codes

Disadvantages:

- Code may be *difficult to read, debug* and *maintain*:
- API limitations need to be worked around!
- Clients may not check for error codes.
  - Experience shows that error codes are often ignored!
  - Behavior is undetermined if errors are not accounted for.
- What if unexpected errors occur that are not described by any error code?
Disadvantages:

- Code may be *difficult to read, debug* and *maintain*:
- API limitations need to be worked around!
- Clients may not check for error codes.
  - Experience shows that error codes are often ignored!
  - Behavior is undetermined if errors are not accounted for.
- What if unexpected errors occur that are not described by any error code?
- No information about the trace of how the error occurred.

**Solution:** Exception Handling!
Exception Handling

**Advantage 1:** Allows separating error detection from error handling.

**Advantage 2:** Allows separating error handling code from regular code.

**Advantage 3:** Errors can’t go unnoticed: *Specify or Handle* rule.

**Advantage 4:** Automatically propagate errors up the call stack.

**Advantage 6:** Keeps track of information on the error stack trace.

**Advantage 7:** Allows grouping and differentiating error types.
Throw something, catch something!
Catching Exceptions

```java
... some code ...
try {
    ... some code ...
    element = myStack.pop();
    ... some code ...
} catch (NoSuchElementException e) {
    // code that handles the exception
}

// this may throw a NoSuchElementException
Control is transferred to the catch block if a NoSuchElementException is thrown
```
Catching Exceptions

... some code ...

```java
try {
    ... some code ...
    element = myStack.pop();
    ... some code ...
}
```catch(NoSuchElementException e) {
    // code that handles the exception
}
```java
catch(FileNotFoundException e) {
    // code that handles the exception
}
```catch(IOException e) {
    // code that handles the exception
}
```java

Add a catch block for every exception that you would like to handle.
Catching Exceptions

```java
try {
    // some code ...
    element = myStack.pop();
    // some code ...
} catch (NoSuchElementException e) {
    // code that handles the exception
} catch (FileNotFoundException e) {
    // code that handles the exception
} catch (IOException e) {
    // code that handles the exception
}
```

Object holding information about the exception: cause, message, stack trace, etc.
Catching Exceptions

```java
... some code ...
try {
    ... some code ...
    element = myStack.pop();
    ... some code ...
}
``` catch(NoSuchElementException e) {
    // code that handles the exception
} catch(FileNotFoundException e) {
    // code that handles the exception
} catch(IOException e) {
    // code that handles the exception
} finally {
    // Example: close the file
}

Code to be executed regardless of whether an exception is thrown or not.
Stack Trace

- **method3**: No exception handling
- **method2**: No exception handling
- **method1**: with exception handling
- **main**
Exception is propagated across the stack frames.

- **method4**: Exception is thrown
- **method3**: No exception handling
- **method2**: No exception handling
- **method1**: with exception handling

main
Stack Trace

Exception is propagated across the stack frames.

- **method4**: Exception is thrown
- **method3**: No exception handling
- **method2**: No exception handling
- **method1**: with exception handling
- **main**

Information about the exception stack trace is stored in the exception object as it is being propagated across the stack frames.
Catching Exceptions

What to do with a caught exception?

- Address the issue! Make sure your code is in a good and stable state in spite of the error.

- Do not hide the issue! Make sure either the admin knows (by logging information) or the user knows (by showing an error message), etc.

- If it is not your responsibility to handle the exception, consider decorating the exception with some information and then re-throwing it again!

```java
try { ... } catch (SomeException e) {
    SomeOtherExceptionType newE = new SomeOtherExceptionType(e);
    // Add some information to newE
    throw newE;
}
```

Preserves information already stored in e.
Exception Handling

**Advantage 1:** Allows separating error detection from error handling.

**Advantage 2:** Allows separating error handling code from regular code.

**Advantage 3:** Errors can’t go unnoticed: *Specify or Handle* rule.

**Advantage 4:** Automatically propagate errors up the call stack.

**Advantage 6:** Keeps track of information on the error stack trace.

**Advantage 7:** Allows grouping and differentiating error types.
Exception Handling

**Advantage 1:** Allows separating error detection from error handling.

**Advantage 2:** Allows separating error handling code from regular code.

**Advantage 3:** Errors can’t go unnoticed: *Specify or Handle* rule.

**Advantage 4:** Automatically propagate errors up the call stack.

**Advantage 6:** Keeps track of information on the error stack trace.

**Advantage 7:** Allows grouping and differentiating error types.
Separate Error Handling Code from Regular Code

initialize errorCode = 0;
open the file;
if (theFileIsOpen) {
  determine the length of the file;
  if (gotTheFileLength) {
    allocate that much memory;
    if (gotEnoughMemory) {
      read the file into memory;
      if (readFailed) {
        errorCode = -1;
      }
    } else {
      errorCode = -2;
    }
  } else {
    errorCode = -3;
  }
  close the file;
  if (theFileDidntClose && errorCode == 0) {
    errorCode = -4;
  } else {
    errorCode = errorCode and -4;
  }
} else {
  errorCode = -5;
}
return errorCode;

Cleaner looking code

Instead of code that looks like this ———————->
Write code that looks like this ——>

```java
try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
} catch (fileOpenFailed) {
    doSomething;
} catch (sizeDeterminationFailed) {
    doSomething;
} catch (memoryAllocationFailed) {
    doSomething;
} catch (readFailed) {
    doSomething;
} catch (fileCloseFailed) {
    doSomething;
}
```
Exception Handling

**Advantage 1:** Allows separating error detection from error handling.

**Advantage 2:** Allows separating error handling code from regular code.

**Advantage 3:** Errors can’t go unnoticed: *Specify or Handle* rule.

**Advantage 4:** Automatically propagate errors up the call stack.

**Advantage 6:** Keeps track of information on the error stack trace.

**Advantage 7:** Allows grouping and differentiating error types.
Recovery is impossible or meaningless!

initCause()
gCause(t)
getMessage()
getStackTrace()
fillInStackTrace()
toString()

OutOfMemoryError

StackOverflowError

LinkageError

NumberFormatException

IllegalArgumentError

IllegalArgumentException

IndexOutOfBoundsException

FileNotFoundException

IOException

SocketException

NullPointerException
Recovery is possible and expected.
Java Exceptions Hierarchy

Unchecked Exceptions. Can ignore handling them
Java Exceptions Hierarchy

Unchecked Exceptions. Must specify or handle!
Unchecked Exceptions

Not checked by the compiler. Catching and handling these exceptions is optional. These exceptions are:

Subclasses of class Error: Like StackOverflowError, OutOfMemoryError, etc. It is not ordinarily expected for a program to be able to recover from these errors. Therefore, it doesn’t make sense to enforce catching/handling them.

Subclasses of class RuntimeException: Like ClassCastException, ArrayIndexOutOfBoundsException, ArithmeticException, etc. These are typically caused by coding flaws. Since, predicting coding flaws is difficult, enforcing catching/handling them is impractical.
Checked by the compiler. Dealing with these exceptions is not optional. These exceptions include everything inherited from class Exception.

Assume method `foo()` throws an `IOException`, which is a checked exception. If method `bar()` calls method `foo()`, then it must do one of the following, or the compiler will complain:

**Handle**

```
public void bar() {
    ... someCode ...
    try {
        ... someCode ...
        foo();
        ... someCode ...
    } catch (IOException e) {
        // code for handling exception
    }
    ... someCode ...
}
```

**Specify**

```
public void bar() throws IOException {
    ... someCode ...
    foo();
    ... someCode ...
}
```

Methods calling `bar()` must either handle or specify!
Create Your Own Exceptions

- If the pre-defined Java Exceptions are not enough, create your own exceptions.
- Choose which Exception classes to subclass (Checked vs Unchecked).
- Organize your exception classes into an inheritance hierarchy to facilitate handling groups of exceptions together.

```java
try {
    ...
} catch (ConnectionException e) {
    // Handles ConnectionException, as well as subclasses
    // like LostConnectionException, InvalidAddressException,
    // AuthenticationErrorException, etc.
} catch (UserInputException e) {
    // Handles UserInputException, as well as subclasses
    // like NoInputProvidedException, InvalidCharsException,
    // DumbUserException, etc.
}
```

Note: These are hypothetical exceptions!
What is the output of the following piece of code?

```java
try {
    throw new IllegalArgumentException();
} catch (Exception e) {
    System.out.println("Exception!");
} catch (RuntimeException e) {
    System.out.println("RuntimeException!");
} catch (IllegalArgumentException e) {
    System.out.println("IllegalArgumentException!");
}
```

A. Exception!
B. RuntimeException!
C. IllegalArgumentException!
D. All of the above.
E. None of the above.
What is the output of the following piece of code?

```java
try {
    throw new IllegalArgumentException();
} catch (Exception e) {
    System.out.println("Exception!");
} catch (RuntimeException e) {
    System.out.println("RuntimeException!");
} catch (IllegalArgumentException e) {
    System.out.println("IllegalArgumentException!");
}
```

A. Exception!
B. RuntimeException!
C. IllegalArgumentException!
D. All of the above.
E. None of the above.
try {
    throw new IllegalArgumentException();
} catch (Exception e) {
    System.out.println("Exception!");
} catch (RuntimeException e) {
    System.out.println("RuntimeException!");
} catch (IllegalArgumentException e) {
    System.out.println("IllegalArgumentException!");
}

2 errors

Exceptions must be caught from most specific to most general.
try {
    ... lotsOfExceptionThrowingCode ...
    ... lotsOfExceptionThrowingCode ...
} catch(Exception e) {
    ... SameExceptionHandlingCodeForAll ...
}
Bad Practice

```java
try {
    // lotsOfExceptionThrowingCode
    // lotsOfExceptionThrowingCode
} catch(Exception e) {
    // Do nothing! Keep it empty!
}
```
Enumeration Types (Live Demo)
JavaDoc (Live Demo)
Anonymous Classes (Live Demo)
Java Exceptions Hierarchy chart was retrieved Monday 22nd 2018 from: https://3.bp.blogspot.com/-j8y3jyEkRKq/WDCVASlGsoI/AAAAAAAADQ8/oTdt8ty-emUBcNuzVzXpZKpTU2nGWeVrACLcB/s1600/ExceptionClassHierarchy.png