Discuss the Java error handling mechanism? What is the difference between Runtime (**unchecked**) exceptions

and **checked** exceptions? What is the implication of catching all the exceptions with the type “*Exception*”?

**Errors:** When a dynamic linking failure or some other “hard” failure in the virtual machine occurs, the virtual

machine throws an Error. Typical Java programs should not catch Errors. In addition, it’s unlikely that typical Java

programs will ever throw Errors either.

**Exceptions:** Most programs throw and catch objects that derive from the Exception class. Exceptions indicate

that a problem occurred but that the problem is not a serious JVM problem. An Exception class has many

subclasses. These descendants indicate various types of exceptions that can occur. For example,

NegativeArraySizeException indicates that a program attempted to create an array with a negative size. One

exception subclass has special meaning in the Java language: RuntimeException. All the exceptions except

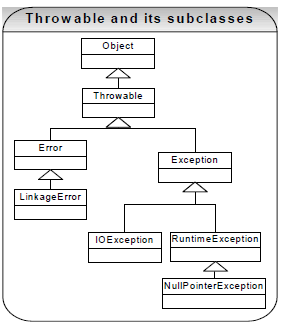
RuntimeException are compiler checked exceptions. If a method is capable of throwing a checked exception it

must declare it in its method header or handle it in a try/catch block. Failure to do so raises a compiler error. So

checked exceptions can, at compile time, greatly reduce the occurrence of unhandled exceptions surfacing at

runtime in a given application at the expense of requiring large throws declarations and encouraging use of poorlyconstructed

try/catch blocks. Checked exceptions are present in other languages like C++, C#, and Python.



***Runtime Exceptions (unchecked exception)***

A RuntimeException class represents exceptions that occur within the Java virtual machine (during runtime). An

example of a runtime exception is NullPointerException. The cost of checking for the runtime exception often

outweighs the benefit of catching it. Attempting to catch or specify all of them all the time would make your code

unreadable and unmaintainable. The compiler allows runtime exceptions to go uncaught and unspecified. If you like, you can catch these exceptions just like other exceptions. However, you do not have to declare it in your

“throws" clause or catch it in your catch clause. In addition, you can create your own *RuntimeException*

subclasses and this approach is probably preferred at times because checked exceptions can complicate method

signatures and can be difficult to follow.

**Exception handling best practices:**

**Why is it not advisable to catch type “*Exception”*? CO**

Exception handling in Java is **polymorphic** in nature. For example if you catch type *Exception in* your code then it

can catch or throw its descendent types like *IOException* as well*.* So if you catch the type *Exception* before the

type *IOException* then the type *Exception* block will catch the entire exceptions and type *IOException* block is

never reached. In order to catch the type *IOException* and handle it differently to type *Exception*, *IOException*

should be caught first (remember that you can’t have a bigger basket above a smaller basket).

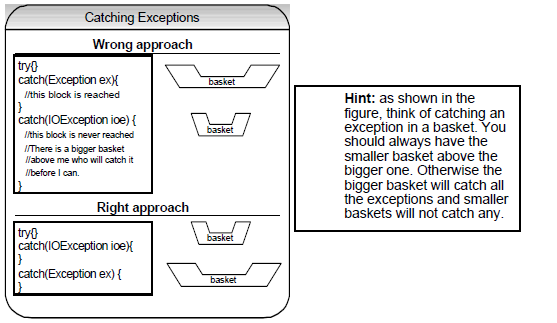
The diagram below is an example for illustration only. In practice it is not recommended to catch type

“***Exception”***. We should only catch specific subtypes of the *Exception* class. Having a bigger basket (i.e.

*Exception*) will hide or cause problems. Since the *RunTimeException* is a subtype of *Exception,* catching the type

*Exception* will catch all the run time exceptions (like NullpointerException, ArrayIndexOut-OfBounds-Exception) as

well.



**Why should you throw an exception early? CO**

The exception stack trace helps you pinpoint where an exception occurred by showing us the exact sequence of

method calls that lead to the exception. By throwing your exception early, the exception becomes more accurate

and more specific. Avoid suppressing or ignoring exceptions. Also avoid using exceptions just to get a flow control.

…

InputStream in = new FileInputStream(fileName); // assume this line throws an exception because filename == null.

…

**Use the following code because you get a more accurate stack trace:**

…

**if(filename == null) {**

**throw new IllegalArgumentException(“file name is null”);**

**}**

InputStream in = new FileInputStream(fileName);

…

**Why should you catch a checked exception late in a catch {} block?**

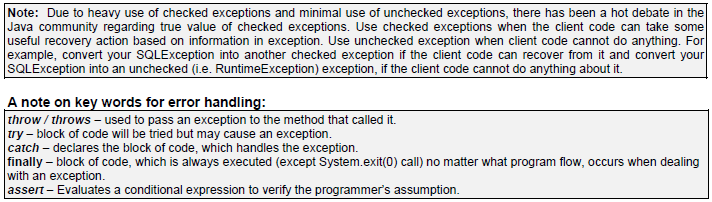
You should not try to catch the exception before your program can handle it in an appropriate manner. The natural

tendency when a compiler complains about a checked exception is to catch it so that the compiler stops reporting errors. The best practice is to catch the exception at the appropriate layer (e.g. an exception thrown at an

integration layer can be caught at a presentation layer in a catch {} block), where your program can either

meaningfully recover from the exception and continue to execute or log the exception only once in detail, so that

user can identify the cause of the exception.



What is a user defined exception?

User defined exceptions may be implemented by defining a new exception class by extending the *Exception* class.

public class MyException extends Exception {

/\* class definition of constructors goes here \*/

public MyException() {

super();

}

public MyException (String errorMessage) {

super (errorMessage);

}

}

Throw and/or throws statement is used to signal the occurrence of an exception. Throw an exception:

**throw** new MyException(“I threw my own exception.”)

To declare an exception: public myMethod() **throws** MyException {…}