Exception Hierarchy*:*

Exceptions in a child class method should adhere to an exception hierarchy that is compatible with the parent class method.

In Java, for example, child class methods can throw exceptions that are subclasses of the exceptions thrown by the parent class method. This is because the parent class method can catch exceptions of its declared types, and child class methods are expected to be compatible with the parent class in this regard.

For example, if the parent class method declares that it can throw an IOException, a child class method can throw FileNotFoundException (which is a subclass of IOException) without any issue.

For checked exceptions (exceptions that are not subclasses of RuntimeException):

Child class methods must follow the exception hierarchy. This means they can throw exceptions that the parent class method is allowed to throw (or their subclasses).

If the child class method throws a broader or unrelated checked exception, it will result in a compilation error. This is because the parent class method may not be able to handle such exceptions.

However, if the child class method in the example were to throw an unrelated exception like SQLException, it would result in a compilation error.

Java:

```
class Parent {
    void doSomething() throws IOException {
        // Parent class method may throw an IOException.
    }
}

class Child extends Parent {
    @Override
    void doSomething() throws FileNotFoundException {
        // Child class method can throw a FileNotFoundException (subclass of IOException).
    }
}
```

Unchecked Exceptions (RuntimeExceptions)

: Unchecked exceptions, on the other hand, are exceptions that are not subject to the same strict rules.

They are instances of the RuntimeException class or its subclasses. These exceptions typically represent programming errors, and the compiler does not require you to explicitly catch or declare them.

They are often related to issues that could be avoided through better programming practices, like null pointer exceptions, index out of bounds, or type casting issues.

Now, when it comes to child classes and exceptions:

If a child class overrides a method from a parent class, the child class method can throw unchecked exceptions without any need to declare them in the method signature.

Here's an example in Java to illustrate this:

Java:

```
class Parent {
    void doSomething() {
        // Parent class method doesn't declare any exceptions.
    }
}

class Child extends Parent {
    @Override
    void doSomething() {
        // Child class method can throw unchecked exceptions, no need to declare them.
        throw new RuntimeException("This is an unchecked exception.");
    }
}
```

In this example, the Child class's doSomething method throws an unchecked exception (RuntimeException) without declaring it. This is allowed by the Java compiler because unchecked exceptions are not enforced in the same way as checked exceptions. You are not required to specify them in the method's signature with a throws clause.

It's important to note that while unchecked exceptions don't need to be declared, you should still handle them appropriately in your code. Failing to do so can lead to unexpected program behavior or crashes. Unchecked exceptions often indicate issues in your code that should be addressed through proper programming practices and error handling, rather than relying on the compiler to enforce handling.

Regenerate