Writing your own Exceptions

How to extend Exception

When would you write your own exception class?

When to write your own custom exception is a matter for discussion in your project design team. There could be some reasons for doing so which we'll discuss briefly. This lecture is more about the syntax, than about such design decisions.

You might want to keep exceptions as a part of your package, so that users of your class, doesn't have to bother about exceptions from totally unrelated packages.

For instance, you might want to hide some exceptions because they make clients of your code dependent on other packages.

Example of hiding an exception

If you have a class for fetching a list of objects, this might be done using a number of technologies, which may change over the evolution of your application.

You might start with fetching them from the filesystem, then move on to fetching them from a database and end up fetching them over the internet using some service.

Clients of your code should probably not have to know or worry about how your class fetches the objects. Having to deal with exceptions from other APIs may create a dependency - SQLExceptions, IOExceptions etc are examples.

A possible solution

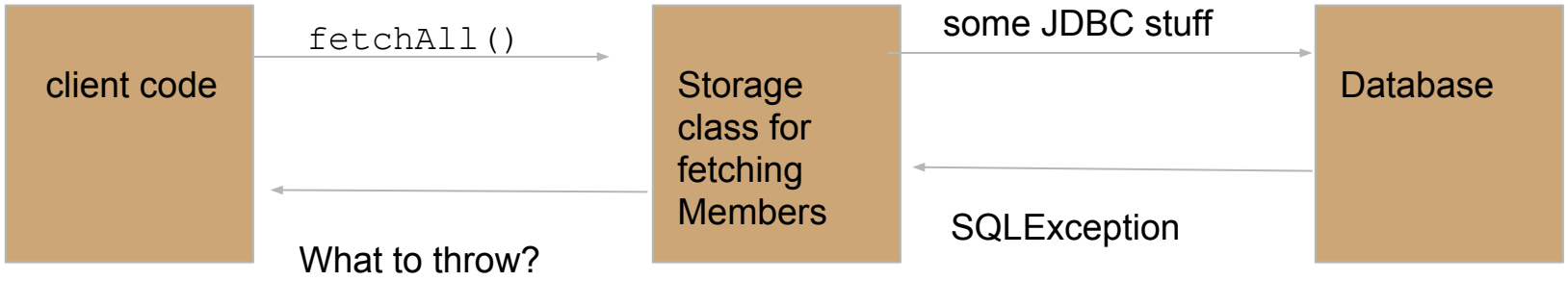
If we have a class for fetching objects, say Member objects from some storage to some application for Member management, clients of this class will have to import the package where the class exists.

But if the class uses a database for fetching, it might throw some SQLException from the fetch() method. This would mean that clients of this class would have to import also java.sql.SQLException for instance.

This is a tight coupling between clients and an external API, which isn't what we want.

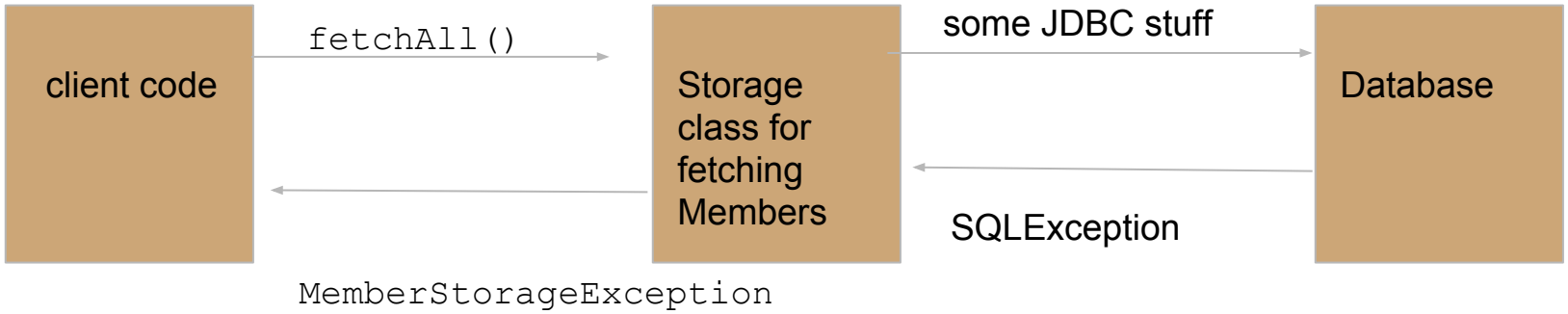
We could then create our own Exception and wrap any SQLExceptions.

System layout



We don’t want client code to import SQLException, because that would tie them to a database solution which might change…

System layout with custom exception



We use our own MemberStorageException class which wraps the SQLException. Client code now only has to depend on classes in our package, and we can replace the database with anything else, but keep our exception!

Extending RuntimeException or Exception?

Which one to extend is not an easy question to answer. But you should know that extending Exception creates a checked exception, so the same discussion as in previous lectures can be applied - do we want the compiler to enforce exception handling on our clients? Extend Exception!

A different way of looking at this could be to ask a different question:

Can we expect the client code to handle the exception and fix the problem? Extend Exception and use checked exceptions.

If they cannot fix the problem, we might as well use RuntimeException.

Extending Exception

To extend Exception, simply write a class with a descriptive name and declare that the class `extends Exception`.

Next, provide at least the following two constructors:

```
public YourException(String message){
  super(message); // allow for the getMessage() to work
}
public YourException(String message, Throwable cause){
  super(message, cause); // allow for wrapping an exception
}
```

Wrapping an exception

Wrapping an exception was what we did when we threw an exception of some type, but created it with a message and another exception like so:

```
catch(IOException e){
  throw new SomeOtherException("I/O problem", e);
}
```

The client code can, if it needs it, investigate the cause, using the method `getCause()` which will return the wrapped exception if it exists, or null if there is no wrapped exception.

Code example (focusing on syntax!)

```
public static void main(String[] args){
  try{
    doStuff();
  }catch(MyCheckedException e){ // catch a custom Exception type
    System.err.println("A MyCheckedException occurred: " +
                       e.getMessage());
    if(e.getCause()!=null){ // check for wrapped exception
      System.err.println("The cause was: " + e.getCause());
    }
  }
}
static void doStuff()throws MyCheckedException{
  // throw a custom exceptio which wraps a NullPointerException
  throw new MyCheckedException("Something went wrong",
                               new NullPointerException("Bad monkey"));
}
```

MyCheckedException

```
public class MyCheckedException extends Exception{

  // Provide two constructors which call the super class
  public MyCheckedException(String message){
    super(message);
  }
  public MyCheckedException(String message,
                            Throwable cause){
    super(message, cause);
  }
}
```

Recap

Creating our own exceptions exposes less dependencies to client code

Extend Exception if you want to have a checked exception

Extend RuntimeException if the clients can’t fix the problem anyway

Just use `extends` and provide at least two constructors

```
(String message)

(String message, Throwable cause)
```

Call the superclass’ constructors respectively