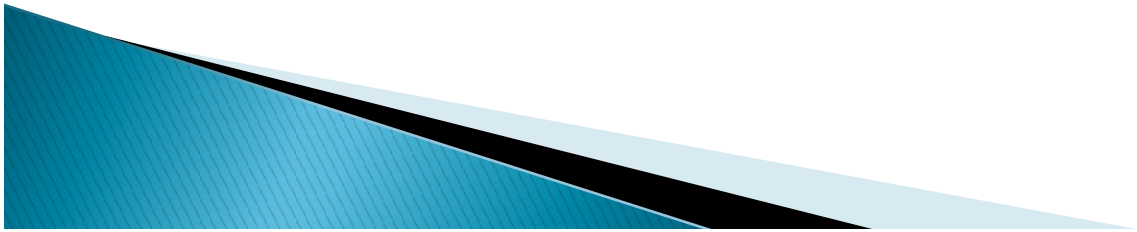
Exception and Error Handling

By
Sunil Kumar(Master of Sc.)
Bangalore, India

1. Agenda

- The role of exception
- The try/catch/finally statement
- Exceptions crossing method boundaries
- Throwing Exceptions
- Custom Exception Type



Error Handling with Exceptions

- A critical aspect of building a successful application is good error handling. If you build a program that doesn't have good handling, it doesn't matter how cool it is.
- It crashes all the time, no one's going to use it. So we need to make sure that handling errors appropriately is really kind of implicit in our application development. Now traditional mechanisms would use things like error code that had to be checked, or flags that had to be checked, and that really intrusive. We may put if statements all over the place to deal with potential errors really going to close up your code. It's really kind of awkward, and code looks unnecessarily big.
- Exceptions provide a non-intrusive way to signal errors. So if exception comes, we can write our code so that, we except things go well. But if an error occurs, an exception is thrown and we can handle it. try/catch/finally provides a structural way to handle exceptions.
- The try block contains the normal code to execute, it is the things we expect to go well. And if all go well, the try block will simply run to completion, and nothing special will happen.

Error Handling with Exceptions

- The catch block contains the error handling code.
  - Block executes only if matching exception is thrown.
- The finally block contains cleanup code if needed.
  - Runs in all cases following try or catch block. So if anything we wanted to do after the process is over, we put that into finally block.
- Previous code without try and catch block i.e. without handling exception.

```
int i  =12;
int j = 5; //What if we will use 2 instead of 5
int result = i/(j-2);
System.out.println(result);
```

Error Handling with Exceptions

- Now we know that this may give any error, so instead of giving some condition like (j-2) ! = 0 we can have an error handling mechanism as below.

```
int i  =12;
int j = 5; //What if we will use 2 instead of 5
try{
    int result = i/(j-2);
    System.out.println(result);
}
catch(Exception e){
    System.out.println("Error: "+e.getMessage());
}
```

Error Handling with Exceptions

- Here getMessage() method gives us back the appropriate message for that exception. It is from the inbuilt class Exception. We have another method printStackTrace() which will print the stack trace showing exactly what went wrong.
- If program runs correctly then it will end up normally. But if there is some error, from that line it will directly go to catch block after skipping the other lines to execute in the try block.

Error Handling with Exceptions

```
BufferedReader reader = null;
int total = 0;
reader = new BufferedReader(new FileReder("C:\\Numberrs.txt"));
String line = null;
while line((line = reader.readlLine()) != null)
        total += Integer.valueOf(line);
System.out.println("Total:" +total);
```

C:\Numbers.txt

```
5
12
6
4
```

- Here in this program, we have used a built-in Java Class called BufferedReader. BufferedReader provides an efficient way to read content.
- Inside BufferedReader we have used another built in class FileReader that takes care of the details of reading content from a file. BufferedReader is something that adds some buffering on top of that to make reading from a file more efficient.

Error Handling with Exceptions

- It also provide some helper methods. One more point to notice that we are having extra \ in the name of the filename. In Java, back-slash is used to identify special characters, so \\ says; we really want it and don't treat this as special character.
- Now, when this is done, we have an instance of our BufferedReader, with a reference to it assigned into the variable reader, and then from there we have declared a local variable called line which will hold the string. Then we have put a while statement which reads from the BufferedReader. We have directly assigned the line variable inside the while statement. One by one it will try to take the input and then add to total which we are printing it at the end.

Error Handling with Exceptions

- There are lots of chances where an error can occurs.
  - File may not exist.
  - There could be some bad data.
  - Some kind of system error may occur.
- To handle this, we may put different kind of if statement and handle it separately or we can have just the try catch block as below.

Error Handling with Exceptions

```
BufferedReader reader = null;
int total = 0;
try{
reader = new BufferedReader(new FileReder("C:\\Numberrs.txt"));
String line = null;
while line((line = reader.readlLine()) != null)
        total += Integer.valueOf(line);
System.out.println("Total:" +total);
}
catch(Exception e){
        System.out.println(e.getMessage());
}
finally{
try{
        if(reader != null)
        reader.close();
} catch(Exception e){
        System.out.println(e.getMessage());
}
}
```

Exception Class Hierarchy

- Errors are represented by exceptions. Each exception defined here is an object. So each exception type is described with a class. Now, as with all classes, the root of the class hierarchy for exceptions is the Object class. When an exception occurs, it is thrown, and we should use catch to handle it. In order for a class to be able to be thrown, it has to inherit from base class – Throwable.

Exception Class Hierarchy

- One of the class it inherits from Throwable is the class Error. There are number of classes that in turn inherit from Error.
- Now the error we tend to interact with much more inherit from a class, Exception. One key class that inherits from Exception is the class, RuntimeException. And there are many other classes that in turn inherit from RuntimeException. In general case, those classes that inherit from RuntimeException represent error in our program.

Exception Class Hierarchy

- The areas where we work most commonly are the classes that inherit more directly from Exception Class.
- A checked exception means that the compiler actually looks and anytime one of these exceptions is thrown, the compiler looks to see if we actually handled it. If we don’t handle a checked exception, the compiler will raise an error means it will not let us build our program. So we have to handle all checked exceptions.
- The exceptions under runtime exceptions are considered as unchecked exceptions means, we can handle them but the compiler doesn’t require that we must do.

Typed Exceptions

- Exceptions can be handled by type.
- Each exception type can have a separate catch block.
- First assignable catch is selected.
- Start catch blocks with most specific exception types and get more general as we go down.

Typed Exceptions Example

```
BufferedReader reader = null;
int total = 0;
try{
        ….
}
catch(NumberFormatException e){
        System.out.println(“Invalid value: ”+e.getMessage());
}
catch(FileNotFoundException e){
        System.out.println(“Not Found: ”+e.getMessage());
}
catch(IOException e){
        System.out.println(“Error interacting with the file:  ”+e.getMessage());
}
finally{
……..
}
```

Exceptions and Method

- Exceptions propagate up the call stack
  - Can cross method boundaries
- Exceptions are part of a method's contract
  - Method is responsible for any checked exception that might occur
- It can be handled by 2 ways
  - Catch the exception
  - Document that the exception might occur
    - Use the throw clause
- We will take an example to understand this concept.

Exceptions and Method

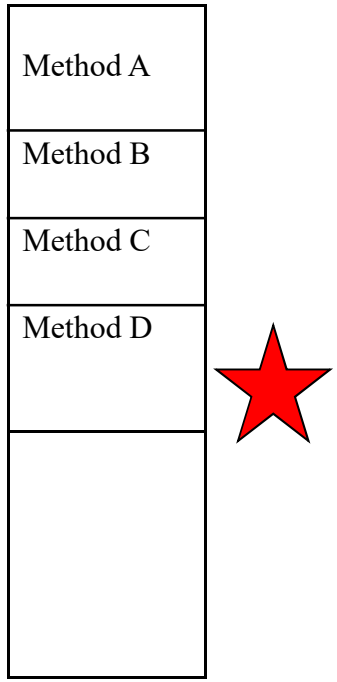
Method A

```
void methodA() {
.....
try{
    methodB
} catch() {
.....
   }
}
```

```
void
methodB(){
.....
methodC();
}
```

```
void methodC(){
.....
methodD();
}
```

```
void methodD(){
.....
//Does something that
//throws an excepton
}
```



Exceptions and Method

```
public class Flight{
        int passengers;
//other method elided for clarity
public void addPassengers(String filename) throws IOException{
        BufferedReader reader = null;
reader = new BufferedReader(new FileReader(filename));
String line = null;
while ((line = reader.readLine()) != null){
        String[] parts = line.split("  ");
        Passengers += Integer.valueOf(parts[0]);
}
}finally {
        If(reader != null)
                Reader.close();
         }
    }
}
```

C:\PassengerList.txt

```
2 Wilson
4 Nyasha
7    Sunil
4    Dean
```

Exceptions and Method

- In this program we know that exceptions can be thrown while totaling the number of passengers.
- In this case we don't want to catch them. We can catch them here, but it probably makes sense that those exceptions propagate up to the caller, because that filename came from the caller. So, if there is anything wrong with it, they might know what to do with it based on the exception.
- So, rather than catch the exception here, we are going to document that it gets thrown.

Exceptions and Method

- So, on our method declaration here, we have added the throws clause which makes sure that it will propagate the error to the caller.
- Here we have added IOException, because we know that we throw an IOException is because, if we were to look, the FileReader constructor is documented to throw a FileNotFoundException, and the BufferedReader readLine method has a throws that says that it throws an IOException.
- Since a FileNotFoundException is also an IOException, we simply say, throws IOException. Point to note here is that, if we didn't catch it and we didn't put the throws on, the compiler would actually complain that we weren't dealing with the exception. So, by saying that we throw it, that is our responsibility in dealing with it.

Exceptions and Method

- Now since exceptions are part of a methods contract, how does that work when one class overrides a method from another class?
- When we override a method, the throws clause of an overriding method must be compatible with the throws clause of the overridden method. There are 3 ways for it.
  - Can exclude exceptions
  - Can have the same exceptions
  - Can have a derived exception

Exceptions and Method

```
public class CargoFlight extends Flight{
    //other method elided for clarity
    @override
     public void addPassengers(String filename) throws IOException{
     public void addPassengers(String filename) throws FileNotFoundException{

          //…….
    }
}
```

- So in this case when we are overriding the addPassengers here in this CargoFlight then it has to be compatible with the throws clause of the Flight's implementation of addPassengers.

Throwing Exception

- Code can throw exceptions
  - Use the throw keyword to provide an instance of the exception we want to throw.
- Since exceptions are object so, must create exception instance before throwing.
  - Create them with new keyword.
  - Be sure to provide meaning detail.