PIC 20A
Exceptions

David Hyde
UCLA Mathematics

Last edited: May 11, 2020

Introductory example

Imagine trying to read from a file.

```java
import java.io.*;

public class Test {
  public static void main(String[] args) {
    //Error!
    FileInputStream in
              = new FileInputStream("file.txt");
  }
}
```

This code doesn't compile because the exception for not having the file is unaccounted for.

Introductory example

Opening a file is risky behavior; it can result in an exception.

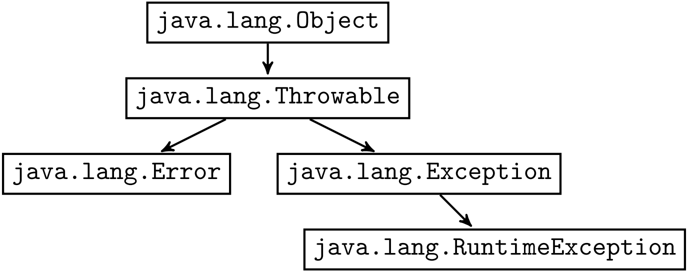
```java
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try {
      FileInputStream in
              = new FileInputStream("file.txt");
    } catch (FileNotFoundException e) {
      System.out.println("File not found.");
      return;
    }
    //Do something with opened file
    ...
  }
}
```

The 3 type of exceptions

Exceptions are objects that inherit `Throwable`.

- Unchecked exceptions inherit `RuntimeException`.
- Checked exceptions inherit `Exception` but not `RuntimeException`.
- Errors inherit `Error`.



Unchecked exceptions

Unchecked exceptions are not checked by the compiler, i.e., the code compiles without a `try-catch` block.

```java
public class Test {
  public static void main(String[] args) {
    int[] i_arr = {1,2,3};
    System.out.println(i_arr[3]);
    //compiles fine but doesn't run fine
  }
}
```

Unchecked exceptions

You can catch unchecked exceptions though.

```
public class Test {
  public static void main(String[] args) {
    int[] i_arr = {1,2,3};
    try {
      System.out.println(i_arr[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception happened");
      System.out.println(e);
      System.out.println(e.getMessage());
    }
  }
}
```

Unchecked exceptions

Unchecked exceptions are also called *runtime exceptions*.

You don't have to explicitly handle a runtime exception,
and you probably don't want to.

Since runtime exceptions are usually caused by bugs or flaws of your code,
you should fix the problem instead of catching the runtime exception.

Checked exceptions

You must explicitly handle checked exceptions, or your code won't compile. (Checked means checked by the compiler.)

Checked exceptions are part of the function specification as much as the input and outputs of the function are.

Functions specify, with the `throws` keyword, what kind of checked exceptions it `throws`.

Errors

`Error` represent fatal problems the program probably can't and shouldn't recover from such as `java.lang.OutOfMemoryError`.

`Errors` are treated like unchecked exceptions in that you don't have to explicitly handle them.

You really shouldn't `catch` `Errors`. If you do, you shouldn't continue the program.

Errors

This example causes a `java.lang.StackOverflowError`.

```java
public class Test {
  public static void fn() {
    fn();
  }
  public static void main(String[] args) {
    fn();
  }
}
```

(This is recursion without a base case.)

try block

To handle exceptions, the risky code must go in a `try` block.

```
try {
  risky code
}
catch ...
```

When an exception is `thrown` the remaining code of the `try` is skipped.

catch block

The `catch` block `catch`es specified exceptions that happened within `try`.

```
try {
  risky code
}
catch (ExceptionType name) {
  code
}
```

catch block

Exceptions are polymorphic. You can specify a superclass to `catch` exceptions of subclass types.

```java
try {
  ...
} catch (Exception e) {
  System.out.println("Any exception");
}
```

catch block

You can have multiple catch blocks.
An exception is checked against each catch block in the stated order.
(At most one catch block is run.)

```
try {
  ...
}
catch (RuntimeException e) {
  System.out.println("Unchecked exception");
}
catch (Exception e) {
  System.out.println("Checked exception");
}
catch (Throwable e) {
  System.out.println("Error");
}
```

(The order of the catch blocks do matter in this example.)

catch block

A single catch block can catch multiple types of exceptions using |.

```
try {
  ...
}
catch (IOException | SQLException e) {
  System.out.println(e);
}
```

throw

throw an exception with throw.

```java
public class Test {
  public static void fn(int i) {
    if (i==0) {
      throw new IllegalArgumentException();
      System.out.println("We never reach here");
    }
  }
  public static void main(String[] args) {
    fn(0);
  }
}
```

(java.lang.IllegalArgumentException is a runtime exception.)

throws

To `throw` a checked exception, specify it with `throws`.

```java
import java.io.*;

public class Test {
  public static void fn() throws IOException {
    ...
    throw new IOException();
  }
  public static void main(String[] args) {
    try {
      fn();
    } catch (IOException e) {
      ...
    }
  }
}
```

Ducking

Instead of `catch`ing an exception, you can *duck* or *specify* it.

```java
import java.io.*;

public class Test {
  public static void main(String[] args)
                  throws FileNotFoundException {
    //Now compiles
    FileInputStream in
              = new FileInputStream("file.txt");
  }
}
```

You “duck” the responsibility of `catch`ing the exception and pass it on to whoever calls your function.
(In this case, you’re passing on the responsibility to JVM, and JVM doesn’t `catch` anything.)

Ducking

Again, exceptions are polymorphic, so you can specify a superclass of what you throw.

```java
import java.io.*;

public class Test {
  public static void main(String[] args)
                  throws Exception {
    //Now compiles
    FileInputStream in
              = new FileInputStream("file.txt");
  }
}
```

Catch or specify

So you must "`catch` or specify" checked exceptions.

Some view `catch` or specify as a flaw of Java that forces cumbersome code. I disagree.

You can bypass the `catch` or specify mechanism by ducking and only throwing runtime exceptions. I don't recommend this.

Should you catch exceptions?

In a sense `Errors` are most severe, and checked exceptions are least severe.

As discussed, you shouldn't `catch` `Errors` and runtime exceptions

You must `catch` or specify checked exceptions, but you may not want to rely on the `catch`.

It's usually better to prevent exceptions from happening at all.
(E.g. check if a file exists before opening it.)

This way, exceptions are truly exceptional, and you use `catch` to exit the program gracefully instead of `catch`ing an exception and keep going.

finally block

The `finally` block *always* executes after the `try-catch` block.

```
try {
  ...
}
catch (SomeException e) {
  ...
}
catch (AnotherException e) {
  ...
}
finally {
  //resource cleanup
  //this code always runs no matter what
  ...
}
```

finally block

```
public static void fn() {
  try {
    throw new Exception();
  }
  catch (Exception e) {
    System.out.println("We’ll exit fn()");
    return;
  }
  finally {
    System.out.println("finally ran");
  }
  System.out.println("We never reach here");
}
```

(To prevent the `finally` block from running, you can unplug your computer’s power cord.)

Why do you need finally?

In Java, you usually don't free resources manually; the garbage collector handles memory management automatically.

However, some resources are very expensive, and you do want to free them manually (using provided methods). Put such resource cleanups in the `finally` block to ensure they always happen.

We'll see a concrete example of this when we talk about streams.