
Exceptions in JAVA

Angel Gruev
Dreamix Ltd.



Outline

- What are exceptions
- Java exceptions syntax
- Hierarchy
- Principles

What Exceptions are For

- To handle Bad Things
 - I/O errors, other *runtime* errors
 - when a function fails to fulfill its specification
 - so you can restore program stability (or exit gracefully)
- To *force you* to handle Bad Things
 - because return codes can be tedious
 - and sometimes you're lazy

Example

//Constructor

```
public FileReader(String fileName)  
    throws FileNotFoundException
```

//Method

```
public void close() throws IOException
```



What Exceptions are NOT For

□ NOT For Alternate Returns:

- e.g., when end-of-file is reached:

```
try{
    while (s = f.read())
    ...
catch(IOException e){ ... }
```

What Exceptions are NOT For [2]

□ NOT For Alternate Returns:

// Horrible abuse of exceptions. Don't ever do this!

```
try {  
    int i = 0;  
    while(true)  
        range[i++].climb();  
} catch(ArrayIndexOutOfBoundsException e) {...}
```

□ Exceptions are only for the exceptional!



Keywords in Java

- **throws**

Describes the exceptions which can be raised by a method.

- **throw**

Raises an exception to the first available handler in the call stack, unwinding the stack along the way.

- **try**

Marks the start of a block associated with a set of exception handlers.



Keywords in Java (2)

□ **catch**

If the block enclosed by the try generates an exception of this type, control moves here; watch out for implicit subsumption.

□ **finally**

Always called when the try block concludes, and after any necessary catch handler is complete.

Example

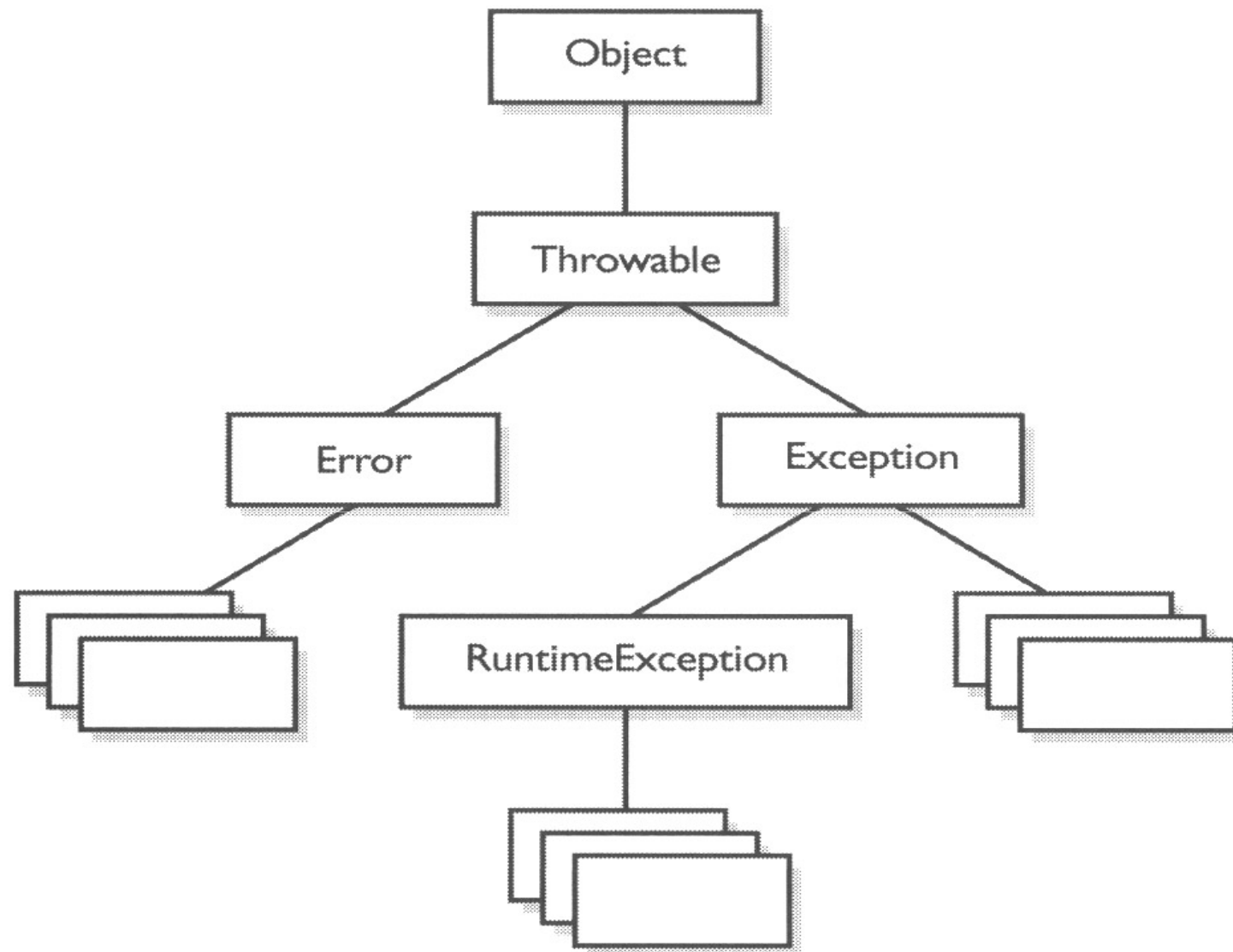
```
public void setProperty(String p_strValue) throws
    NullPointerException{
    if (p_strValue == null) {
        throw new NullPointerException("...");
    }
}

public void myMethod() {
    MyClass oClass = new MyClass();
    try {
        oClass.setProperty("foo");
        oClass.doSomeWork();
    } catch (NullPointerException npe) {
        System.err.println("Unable to set
            property: "+npe.toString());
    } finally {
        oClass.cleanup();
    }
}
```



Steps of try...catch...finally

- Every try block must have at least one catch or finally block attached.
- If an exception is raised during a try block:
 - The rest of the code in the try block is skipped over.
 - If there is a catch block of the correct, or derived type, it is entered.
 - If there is a finally block, it is entered.
 - If there is no such block, the JVM moves up one stack frame.



Class java.lang.Exception

- The one you usually derive from
- “Checked Exceptions”
 - specifications checked at *compile* time
 - you must either catch or advertise these
 - Used for recoverable errors
 - Not programmer errors



Class java.lang.Error

- For JVM Failures and other Weird Things
 - let program terminate
- **InternalError** is one of these
- Don't catch them
 - you don't know what to do!
- These are "unchecked exceptions"
 - not required to advertise



Class

java.lang.RuntimeException

- Program logic errors
 - e.g., bad cast, using a null handle, array index violation, etc.
 - Shouldn't happen!
- These are called "unchecked exceptions"
- Examples:
 - ArithmeticException (e.g., divide by 0)
 - ClassCastException
 - IndexOutOfBoundsException
 - NullPointerException
- indicate *precondition violations*



Principle

- “Use checked exceptions for recoverable conditions and run-time exceptions for programming errors” (Bloch, *Effective Java*)



Exceptions

- Avoid unnecessary use of checked exceptions
 - places a nontrivial burden on the programmer
 - Burden is higher if it is sole exception
- The burden is justified if:
 - the exceptional condition cannot be prevented by proper use of the API
 - the programmer using the API can take some useful action

Three Critical Decisions

- How do you decide to raise an exception rather than return?
- Should you reuse an existing exception or create a new type?
 1. Can you map this to an existing exception class?
 2. Is the checked/unchecked status of mapped exception acceptable?
- How do you deal with subsumption in a rich exception hierarchy?
 1. Avoid throwing a common base class (e.g. IOException).
 2. Never throw an instance of the Exception or Throwable classes.



APIs and Exceptions

- *A well-designed API must not force its clients to use exceptions for ordinary control flow.*

- Iterator API:

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext();) {  
    Foo foo = i.next();  
    ...  
}
```

- Bad example:

```
try {  
    Iterator<Foo> i = collection.iterator();  
    while(true) {  
        Foo foo = i.next();  
        ...  
    }  
} catch (NoSuchElementException e) { ... }
```



Exceptions info

- Exceptions are full-fledged objects
 - arbitrary methods can be defined
 - Provide additional information
 - Bad practice: parse the string of the exception for additional info

The `finally` Clause

- For code that must ALWAYS run
 - No matter what!
 - Even if a `return` or `break` occurs first
 - Exception: `System.exit()`
- Placed after handlers (if they exist)
 - try-block must either have a handler or a finally-block



```
class FinallyTest
{
    public static void f()
    throws Exception
    {
        try
        {
            // return;           // 0
            // System.exit(0);    // 1
            // throw new Exception(); // 2
            // throw new Exception(); // 3a
        }
        catch (Exception x)
        {
            // throw new Exception(); // 3b
        }
        finally
        {
            System.out.println("finally!");
        }

        System.out.println("last statement");
    }
}
```



Program Output

-
- 0:
 finally!
 last statement
 - 1:
 finally!
 - 2:
 (no output)
 - 3a:
 same as 0:
 - 3a + 3b:
 compiler error (last statement not
 reachable)
-



Standard exceptions

- Favor the use of standard exceptions
 - Exceptions are no exception to the general rule that code reuse is good
- Reusing preexisting exceptions has several benefits
 - makes your API easier to learn and use because it matches established conventions
 - Easier to read code
 - fewer exception classes mean a smaller memory footprint and less time spent loading classes

Standard Exceptions

- ☐ IllegalArgumentException
- ☐ IllegalStateException

- ☐ NullPointerException
- ☐ IndexOutOfBoundsException
- ☐ UnsupportedOperationException
- ☐ NumberFormatException

Abstraction and Exceptions

- Throw exceptions appropriate to the abstraction

Higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction.

- General way

```
// Exception Translation
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```



Exception chaining

- lower-level exception might be helpful to someone debugging the problem that caused the higher-level exception.

```
// Exception Chaining
try {
    ... // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

Document exceptions

- Always declare checked exceptions individually
 - document precisely the conditions under which each one is thrown
 - Javadoc @throws tag

- Document unchecked exception too
 - But don't put them in throws clause of method
 - Ideal, not always achievable

Don't ignore exceptions

- An empty catch block defeats the purpose of exceptions

```
// Empty catch block ignores exception -  
Highly suspect!  
try {  
    ...  
} catch (SomeException e) {}
```

- Exceptions of this case
 - i.e closing a FileInputStream

Questions?

