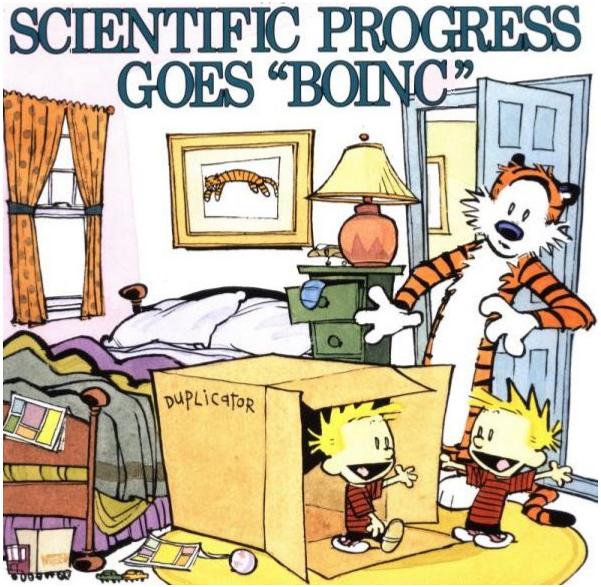

SCIENTIFIC PROGRESS GOES "BOINC"
DUPLICATOR

- What is the purpose of exceptions and exception handling?
- Vocabulary: throw/raise and catch/handle
- Java checked and unchecked exceptions
- Exception propagation
- Java `try` statement
- “Final wishes”
- Java `try`-resource statement

What are exceptions?

Bad things happen occasionally.

arithmetic: $\div 0$, $\sqrt{-9}$

environmental: no space, malformed input

application: can't invert a singular matrix, can't pop an element from an empty stack

Some of these anomolies can be detected by hardware, some by the operating system, some by the runtime system of the language, and other only by the application itself. Which is which may depend on the language implementation, hardware, etc.
In general, an exception should be used when there is an inability to fulfill a specification.

Catching and Throw Exceptions

Raising an exception halts normal execution abruptly, then alternative statements are sought to be executed, possibly the program terminates.
There are many exceptions already defined by the Java language and the Java API And the programs may create their own exceptions. Exceptions are said to be *raised* or *thrown* at the point of interruption and are said to be *handled* or *caught* at the point when normal execution resumes.

Examples

```
public static void main (String[] args) {
    out.println (9.8/0.0); // OK
    out.println (5/0);     // Exception
}
```

```
> java Arith
Infinity
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Arith.main(Arith.java:5)
>
```

Examples from JVM

Fragment of the file `NonMain.java`:

```
public static void main (String args) {
   out.println ("Hello?");
}
```

Common mistakes running a Java program yield very confusing exceptions raised by the runtime system even before programs gets started.

```
> java NotMain
java.lang.NoClassDefFoundError: NotMain
Exception in thread "main"


> java NonMain
java.lang.NoSuchMethodError: main
Exception in thread "main"
```

- Wrong class name: `NonMain` not `NotMain`
- Not an entry point: `String` array is required

More Examples

Some common predefined exceptions: `ArrayIndexOutOfBoundsException`, `NumberFormatException`, `NegativeArraySizeException`, `NullPointerException`.

`Examples.java` – predefined exceptions

Programmer defined exceptions are important. Examples of them come later.

Why Exceptions?

Why not use ordinary computations values like integers (return-code)? Answer: to separate normal flow of control from error handling making it easier to read.

The C programming language does not have exception handling. Well-written (defensive) code is hard to read:

```
if ((fd=open(name, O_RDONLY))==-1) {
   fprintf (stderr, "Error %d opening file", errno);
   exit();
}
```

See also the well-written C program from Stevens.

main.c

Reasons Not To Use Ad Hoc Approach

There are some problems with an ad hoc approach:

- Easy to leave out checking, hence the code will be error prone
- Poor modular decomposition
- Hard to test such programs
- Inconsistency — sometimes null, sometimes -1
- No additional info about the exception

Catching

What can you do when an exception happens? If you can, repair the problem. *Sometimes there is nothing to do!* Some options are: try again anyway, log the incident, terminate the program early (with an explanation), print a stack trace, ...
If you do nothing Java will terminate the program and print a stack trace—often this is the best you can do.
More about the `try` statement later. First, we must understand what an exception is in Java, so we can learn the mechanisms in Java which support exception handling.

Digression

Exceptions are not always the correct approach.
Better language design could eliminate `NullPointerException`.

Java SE 8 introduces a new class called `java.util.Optional<T>` that is inspired from the ideas of Haskell and Scala. It is a class that encapsulates an optional value. You can view Optional as a single-value container that either contains a value or doesn't (it is then said to be "empty").
See the Elvis operator in Kotlin and Groovy.

```
final String version =
    computer?.getSoundcard()?
      .getUSB()?.getVersion() ?: "UNKNOWN";

final String version =
  computer.map (Computer::getSoundcard)
          .map (Soundcard::getUSB)
          .map (USB::getVersion)
          .orElse("UNKNOWN");
```

Monad!

```java
Optional<String> nameOptional = Optional.of("Bob");
int len = nameOptional.map(String::length).orElse(0);
```

Java's approach is a small step in eliminating the "billion dollar" mistake that `null` is.

Brian Goetz himself on StackOverflow

> *But we did have a clear intention when adding this feature, and it was not to be a general purpose Maybe type, as much as many people would have liked us to do so. Our intention was to provide a limited mechanism for library method return types where there needed to be a clear way to represent "no result", and using null for such was overwhelmingly likely to cause errors.*
>
> *For example, you probably should never use it for something that returns an array of results, or a list of results; instead return an empty array or list. You should almost never use it as a field of something or a method parameter.*
>
> *I think routinely using it as a return value for getters would definitely be over-use.*
>
> *There's nothing wrong with Optional that it should be avoided, it's just not what many people wish it were, and accordingly we were fairly concerned about the risk of zealous over-use.*

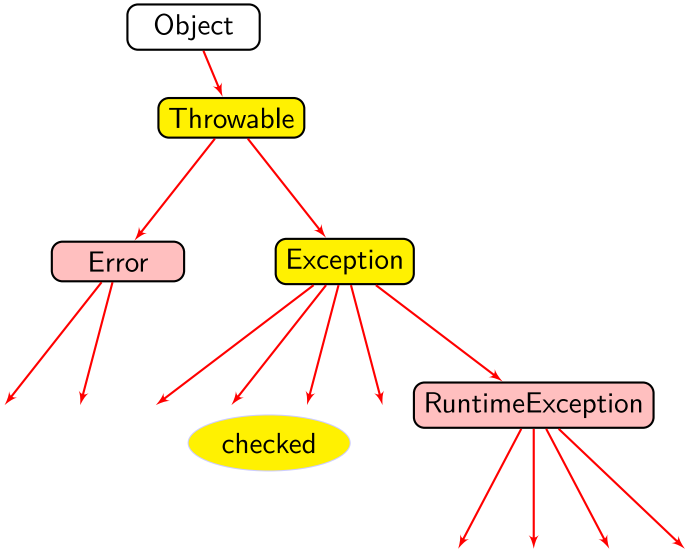
What is an Exception?

Exceptions in Java are *not* new and different entities (like in Ada or SML). Exceptions are Java classes which are subclasses of `java.lang.Throwable`.

An exception is not a separate kind of entity in Java, it is a class. But exceptions do have a special role in the language. They are not the programmer's data, but they serve as signals or indicators.

Since different instances of an exception class are usually indistinguishable and not very important, we have a tendency to blur the distinction between an exception and an instance of the exception. We speak of *the* `FileNotFoundException` exception, and not of *an* instance of the `FileNotFoundException` exception.

Class Hierarchy



Checked

Some exceptions are *checked exceptions* this is important to know. (As we will see later.)

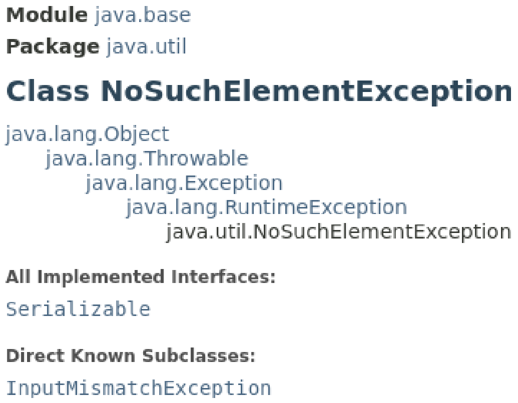
Definition

A *checked exception* is an exception [Java class] that derives from `Throwable`, but not from either `Error` or `RuntimeException`.

A purpose of *checked exception* is to call extra attention to some analomies.

Example

The exception `NoSuchElementException` is an unchecked exception because it is a subclass of `RuntimeException`, as seen can be seen in the Java 18 API documentation below:



Module java.base

Package java.util

Class NoSuchElementException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.NoSuchElementException
```

All Implemented Interfaces:

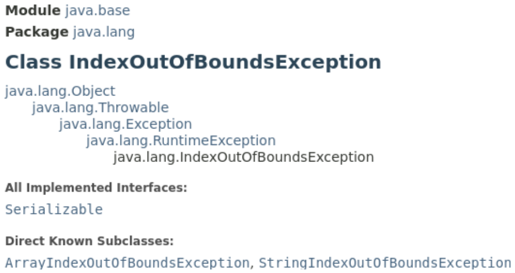
`Serializable`

Direct Known Subclasses:

`InputMismatchException`

Example

The exception `IndexOutOfBoundsException` is an unchecked exception because it is a subclass of `RuntimeException`, as seen can be seen in the Java 18 API documentation below:



Module java.base
Package java.lang

Class IndexOutOfBoundsException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
```

All Implemented Interfaces:

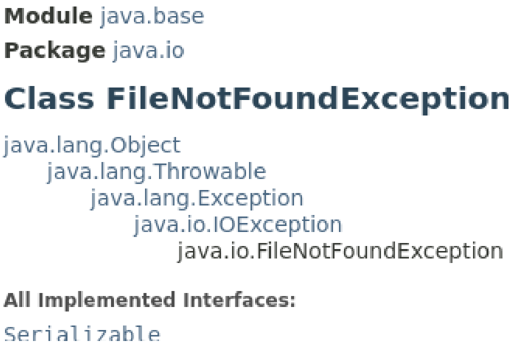
`Serializable`

Direct Known Subclasses:

`ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`

Example

The exception `FileNotFoundException` is a checked exception because it is a subclass of `Throwable` and it is not a subclass of `RuntimeException` nor `Error`, as seen can be seen in the Java 18 API documentation below:



Module java.base

Package java.io

Class FileNotFoundException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

All Implemented Interfaces:

Serializable

throw

The programmer raises an exception with the `throw` statement, for example:

```
throw new Exception ()
```

If a methods throws a checked exception (and does not catch it), then it must declare the fact in a `throws` clause.

```
static void method (String arg) throws AnException
```

This is *checked* by the compiler.

Do not confuse the `throw` statement with the `throws` clause of a method declaration. These are two different keywords.

The Java language requires that a checked exception be caught or declared, otherwise the program won't compile.

```java
import java.io.*;
import java.util.*;
public class Checked {
    // Will raise NoSuchElementException,
    // if args.length==0
    public static void main (String[] args) {
        System.out.println (Collections.min (
            Arrays.asList (args)));
    }
    // Will raise FileNotFoundException,
    // if 'file_name' does not exist
    public static void main (String file_name)
        throws FileNotFoundException {
        InputStream inp=new FileInputStream(file_name);
    }
}
```

Blocks With Handlers

In Java (as most languages) a set of handlers watches over a block of code. When an exception is raised somewhere (perhaps in a subroutine call) in the block, execution stops at that point and a handler is sought. If one of the handlers is invoked after a successful search, the code of the handler is executed (naturally), and then the entire block ends as if no exception were ever raised.

The try statement contains and guards a block of statements.

```
// Some statements
try {
   // the normal flow of
   // executable statements
} catch (final IOException ex) {
   // handler for ``ex''
} catch (final Exception ex) {
   // handler for ``ex''
}
// Additional statements
```

NB. Order of handlers is important; catching `Exception` is like `else` in a conditional statement. The handlers are checked in order.

Because exceptions are classes in Java, the programmer can take advantage of the organization of the class hierarchy in creating and catch exceptions.

```
try {
   // the normal flow of
   // executable statements
} catch (final MalformedURLException ex) {
   // ...
} catch (final EOFException ex) {
   // ...
} catch (final IOException ex) {
   // ...
} catch (final Exception ex) {
   // ...
}
```

`MalformedURLException` and `EOFException` are both subclasses of `IOException`. The order of the handlers is crucial.

Catching Multiple Exceptions

```
try {

    // the normal flow of
    // executable statements

} catch (final IOException | SQLException ex) {
     logger.log (ex);
     throw ex
}
```

It is possible to have a handler catch distinct exceptions. NB. The name of the exception instance is implicitly final. This is because the type of the instance is possibly different when different exceptions are thrown. Of course, it is rare to assign a new value to exception varaible.

Exception Propagation

There is no attempt at going back and finishing remaining actions in the block. Although this appears desirable, it is more difficult than it looks. Nonetheless, the programmer can still program any sort of resumption imaginable by careful use of block-structured exception handling,

Exception propagation. Modern languages all take the same approach to *exception propagation*, the search for the place to resume normal execution: follow the dynamic chain of method or block activations. This is obviously correct: the caller who asked for the service should hear of the failure.

Exception Propagation

- `Propagation.java` – simple propagation
- `PropagationFin.java` – with `finally` clause

Examples

- `Declare.java` – catch or declare checked exceptions
- `Pre.java` – handling some predefined exceptions
- `Value.java` – hierarchy of user defined excpetions some with members to hold additional information
- `Trace.java` – call `printStackTrace()`

Exception Chaining∗

Exceptions can have other exceptions as causes.

```
void aMethod() throws SomeException {
   try {
     someOtherMethod ();
   } catch (final SomeOtherException soe) {
     final SomeException se =
        new SomeException (soe.getMessage());
     se.initCause (soe);
     throw se;
   }
}
```

Java has one combined statement:

```java
// previous statements
try {
    // executable statements
} catch (final NullPointerException ex) {
    // handler for an exception
} finally {
    // 1. normal, 2. caught exc, 3. uncaught
    // exc, 4. break, continue, 5. return
    // final wishes
}
// following statements
```

`finally` Clause

There is some confusion with the `finally` clause.
The code in the `finally` clause ought not change the kind of control flow: normal, exceptional, return, or break. In other words, if the block is getting ready to break out of a loop it ought not to return instead. Or, if the block is getting ready to raise an exception it ought not to break out of a loop instead.

- `FinReturn.java` – Java warns

Finalization

> ***Finalization.*** *Probably the worst feature in Java. The original intention was to provide a method for automatically closing resources when they were no longer needed. However, the mechanism relies upon Java's garbage collection, which is non-deterministic. Thus, using finalization to reclaim resources is fundamentally unsafe. Therefore it is impossible to use finalization as a way of avoiding resource exhaustion, and the feature cannot be fixed. In other words, never use finalization.*

Evans, *Java: The Legend*, 2015, page 22.

Instead use the “try-resources” statement introduced in Java 7.

Try-Resource Construct

```
try (final BufferedReader br =
    new BufferedReader (new InputStreamReader (
    new FileInputStream("file.txt"),"LATIN-1"))) {
    for(;;) {
        final String line = br.readLine();
        if (line==null) break;
        System.out.println(line);
    }
} catch (final IOException ex) {
    ex.printStackTrace();
}
```

See a more substantial example in a server: KnockKnockServer.java.

1. Use exception handling for unusual, unexpected, unlikely situations.
2. Do not use exception handling for detecting the end-of-file in the input streams. That is not an unusual case and there are specific methods for detecting end-of-file.
3. Raise an exception when a method cannot fulfill its specification.
4. Do not catch an exception to cover-up bad programming.
5. Do not handle an exception unless you can *fix* the problem.
6. There is little point in catching an exception just to report it. The runtime system reports exceptions adequately.
7. Never fail to report or log that an exception has been raised. (Checkstyle will flag `catch (Exception e)` for `EmptyBlock`.)