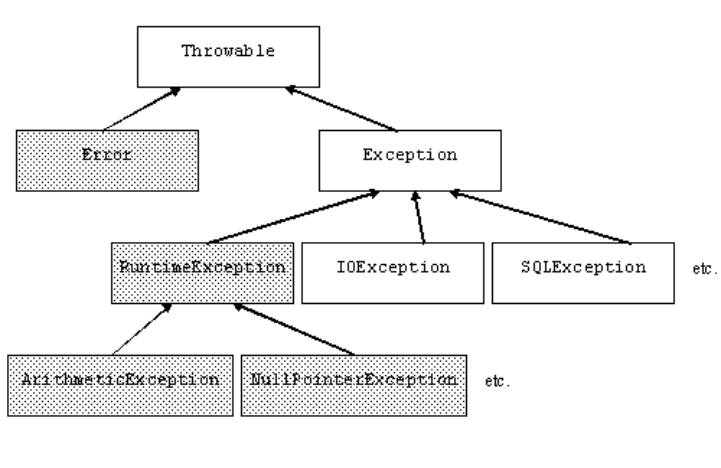
Programming in Java

Java Exceptions

KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk



Errors and Exceptions

An error is a bug in your program, for example

- dividing by zero
- going outside the bounds of an array
- trying to use a `null` reference

An exception is a problem whose cause is outside your program, for example

- trying to open a file that isn't there
- running out of memory

[Note: Java does not adopt these terms in quite the same way — as we will see later]

Birkbeck UNIVERSITY OF LONDON

What to do about errors and exceptions

An error is a bug in your program

- It should be fixed

An exception is a problem that your program may encounter

- The source of the problem is outside your program
- An exception is not the *normal* case, but...
- ...your program must be prepared to deal with it

This is not a formal distinction – it isn't always clear whether something should be an error or an exception

Dealing with exceptions

Most exceptions arise when you are handling files

- A needed file may be missing
- You may not have permission to write a file
- A file may be the wrong type

Exceptions may also arise when you use someone else's classes (or they use yours)

- You might use a class incorrectly
- Incorrect use should result in an exception

The problem with exceptions

Here's what you might like to do:

1. open a file
2. read a line from the file

But here's what you might have to do:

1. open a file
2. if the file doesn't exist, inform the user
3. if you don't have permission to use the file, inform the user
4. if the file isn't a text file, inform the user
5. read a line from the file
6. if you couldn't read a line, inform the user
7. etc., etc.

All this error checking really gets in the way of understanding the code

Three approaches to error checking

1. Ignore all but the most important errors
   - The code is cleaner, but the program will misbehave when it encounters an unusual error
2. Do something appropriate for every error
   - The code is cluttered, but the program works better
   - You might still forget some error conditions
3. Do the normal processing in one place, handle the errors in another (this is the Java way)
   - The code is at least reasonably uncluttered
   - Java tries to ensure that you handle every error

Example: Throwing an exception

When you throw an exception, you are throwing an object of an exception class

```
if (amount > balance){
    throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount; // This line is not executed when the
                            // exception is thrown
```

When you throw an exception, the normal control flow is terminated

The `try` statement

Java provides a new control structure, the `try` statement (also called the `try-catch` statement) to separate "normal" code from error handling:

```
try {
    do the normal code, ignoring possible exceptions
}
catch (some exception) {
    handle the exception
}
catch (some other exception) {
    handle the exception
}
```

Example: Catching Exceptions

When an exception is detected, execution *jumps* immediately to the first matching `catch` block

```
try {
    String filename = ...;
    Scanner in = new Scanner(new File(filename)); // IOException??
    String input = in.next();                     // NoSuchElementException??
    int value = Integer.parseInt(input); // NumberFormatException??
    ...
}
catch (IOException exception){
    System.out.println(exception);
}
catch (NumberFormatException exception){
    System.out.println("Input was not a number");
}
```

What happens if a `NoSuchElementException` occurs?

Exception handling is *not* optional

- As in other languages, *errors* usually just cause your program to crash
- Other languages leave it up to you whether you want to handle *exceptions*
  - There are a lot of sloppy programs in the world
  - It's normal for human beings to be lazy
- Java tries to force you to handle exceptions
  - This is sometimes a pain in the neck, but...
  - the result is almost always a better program

Birkbeck UNIVERSITY OF LONDON

`Error` and `Exception` are `Objects`

- In Java, an error doesn't *necessarily* cause your program to crash
- When an *error* occurs, Java throws an `Error` object for you to use
  - You can catch this object to try to recover
  - You can *ignore* the error (the program will crash)
- When an *exception* occurs, Java throws an `Exception` object for you to use
  - You cannot ignore an `Exception`; you must catch it
  - You get a *syntax error* if you forget to take care of any possible `Exception`

Checked Exceptions

Throw/catch applies to three types of exceptions:

Error: Internal errors

Unchecked: `RuntimeException` (what we called errors)

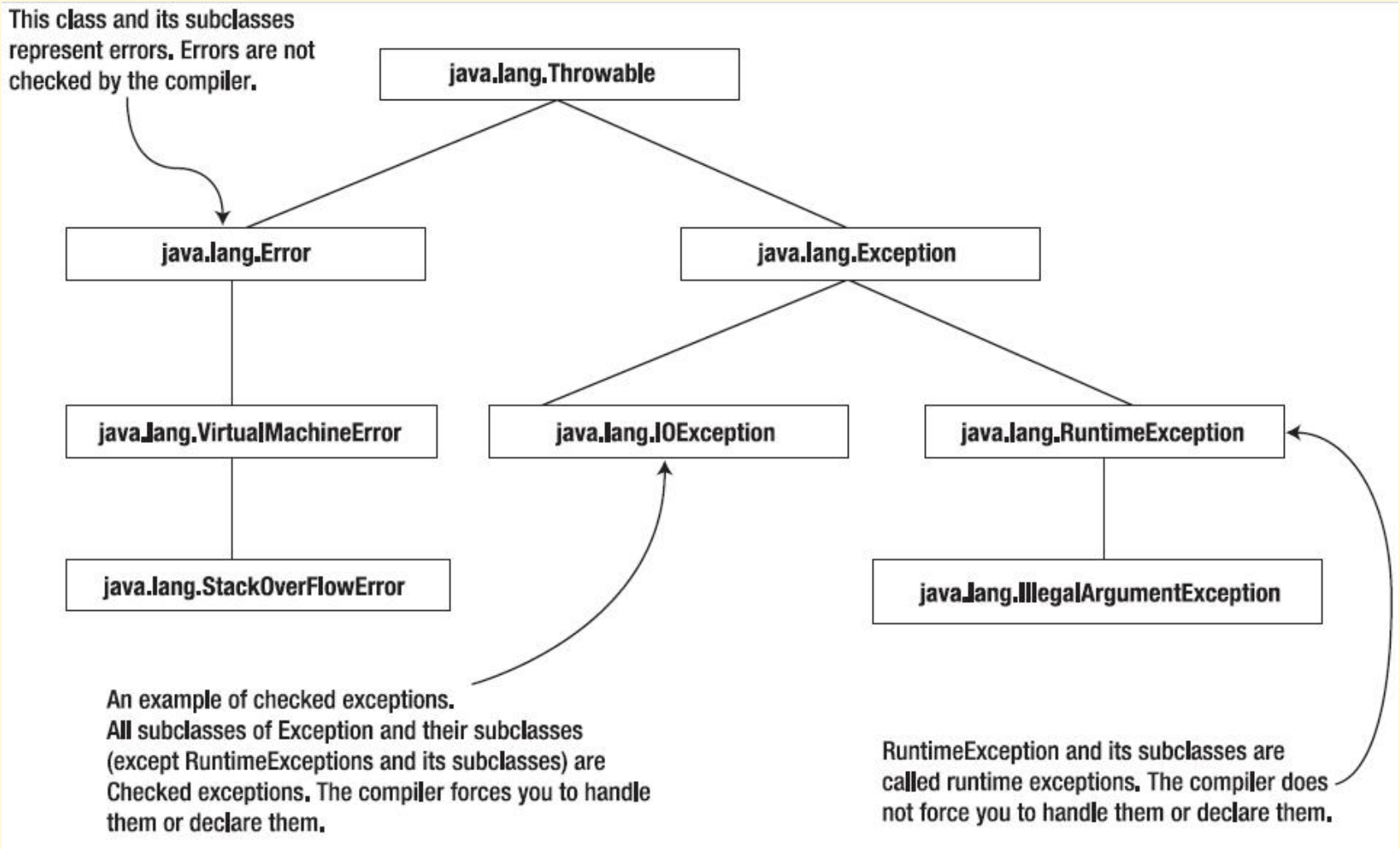
- Caused by the programmer
- Compiler does not check how you handle them

Checked: All other exceptions

- Not the programmer's fault
- Compiler checks to make sure you handle these

Checked exceptions are due to circumstances that the programmer cannot prevent

The exception hierarchy I



The exception hierarchy II

`Throwable`: the superclass of "throwable" objects

- `Error`: Usually should not be caught (instead, the bug that caused it should be fixed)
- `Exception`: A problem that must be caught
  - `RuntimeException`: A special subclass of Exception that does *not* need to be caught

Hence, it is the `Exceptions` that are most important to us (since we have to do something about them)

A few kinds of `Exceptions`

`IOException:` a problem doing input/output

- `FileNotFoundException:` no such file
- `EOFException:` tried to read past the End Of File

`NullPointerException:` tried to use a object that was actually null (this is a `RuntimeException`)

`NumberFormatException:` tried to convert a non-numeric string to a number (this is a `RuntimeException`)

`OutOfMemoryError:` the program has used all available memory (this is an `Error`)

There are over 200 predefined exception types

What to do about `Exceptions` I

You have two choices:

1. You can *catch* the exception and deal with it
   For Java's exceptions, this is usually the better choice
2. You can *pass the buck* and let some other part of the program deal with it
   (This is often better for exceptions that you create and throw)

What to do about `Exceptions` II

- Exceptions should be handled by the part of the program that is best equipped to do the right thing about them
- You can catch exceptions with a `try` statement
- When you catch an exception, you can try to repair the problem, or you can just print out information about what happened

What to do about `Exceptions` III

- You can *pass the buck* by stating that the method in which the exception occurs *throws* the exception

  Example:

```
void openFile(String fileName) throws IOException { ... }
```

- Note:
  - all checked exceptions that a method could possibly throw need to be specified in the `throws` clause
  - you may also list unchecked exceptions

What to do about `Exceptions` IV

- Whether you should catch or pass the buck depends on *whose responsibility it is* to do something about the exception
  - If the method *knows* what to do, it should do it
  - If it should really be up to the user (the method caller) to decide what to do, then *pass the buck*

How to use the `try` statement

- Put `try ...` around any code that might throw an exception (This is a *syntax* requirement you cannot ignore)
- For each `Exception` object that might be thrown, you must provide a `catch` phrase:

```
catch (exception_type  name) {...}
```

- You can have as many `catch` phrases as you need
- *name* is a formal parameter that holds the exception object
- You can send messages to this object and access its fields

The `finally` clause I

After all the catch phrases, you can have an *optional* `finally` clause

```
try { ... }
catch (AnExceptionType e) { ... }
catch (AnotherExceptionType e) { ... }
finally { ... }
```

The `finally` clause II

Whatever happens in `try` and `catch`, *even if a* `return` *statement* occurs, the `finally` code will be executed.

- If no exception occurs, the `finally` will be executed after the `try` code
- If an exception does occur and it is caught, the `finally` will be executed after the appropriate `catch` code
- If an exception does occur and it is not caught, the `finally` is executed immediately after the exception is thrown

How the `try` statement works

- The code in the `try {...}` part is executed
- If there are no problems, the `catch` phrases are skipped
- If an exception occurs, the program jumps *immediately* to the first `catch` clause that can handle that exception
- Whether or not an exception occurred, and whether or not it is caught, the `finally` code is executed

Birkbeck UNIVERSITY OF LONDON

Ordering the catch phrases

A `try` can be followed by many `catch`es

- The first one that *can* catch the exception is the one that *will* catch the exception

Bad:

```
catch(Exception e) { ... }
catch(IOException e) { ... }
```

This is bad because `IOException` is a subclass of `Exception`, so any `IOException` will be handled by the *first* `catch`

- The second `catch` phrase can never be used

Using the exception

When you say `catch(IOException e)`, `e` is a *formal parameter* of type `IOException`

- A catch phrase is almost like a miniature method
- `e` is an instance (object) of class `IOException`
- `Exception` objects have methods you can use

Here's an especially useful method that is defined for every exception type:

```
e.printStackTrace();
```

This prints out what the exception was, and how you got to the statement that caused it

printStackTrace()

`printStackTrace()` does not print on `System.out`, but on another stream, `System.err`

- Eclipse writes this to the same *Console* window, but writes it in red
- From the command line: both `System.out` and `System.err` are sent to the terminal window

`printStackTrace(stream)` prints on the given stream

- `printStackTrace(System.out)` prints on `System.out`, and this output is printed along with the "normal" output

Throwing an Exception

If your method uses code that might throw an exception, and you don't want to handle the exception in this method, you can say that the method *throws* the exception

Example:

```
String myGetLine( ) throws IOException { ... }
```

If you do this, then the method that calls this method must handle the exception or also have a similar `throws` clause

Constructing an `Exception`

Exceptions are objects; you can create your own `Exceptions` with `new`

- Exception types have two constructors: one with no parameters, and one with a `String` parameter

You can subclass `Exception` to create your own exception type

- But first, you should look through the predefined exceptions to see if there is already one that's appropriate

Throwing an `Exception`

Once you create an `Exception`, you can throw it

```
throw new UserException("Bad data");
```

You don't *have* to throw an `Exception`; here's another thing you can do with one:

```
new UserException("Bad data").printStackTrace();
```

Why create an `Exception`?

- If you are writing methods for someone else to use, you want to do something reasonable if they use your methods incorrectly
- Just doing the wrong thing isn't very friendly
- Remember, error messages are a good thing — much better than not having a clue what went wrong
- Exceptions are even better than error messages, because they allow the user of your class to decide what to do

Birkbeck UNIVERSITY OF LONDON

Programming Tip I

Throw Early

- When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix

Catch Late

- Conversely, a method should only catch an exception if it can really remedy the situation
- Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler

Programming Tip II

Do Not Squelch Exceptions

- When you call a method that throws a checked exception and you havent specified a handler, the compiler complains
- It is tempting to write a *do-nothing* catch block to *squelch* the compiler and come back to the code later — Bad Idea!
  - Exceptions were designed to transmit problem reports to a competent handler
  - Installing an incompetent handler simply hides an error condition that could be serious

The `assert` statement I

- The purpose of the `assert` statement is to document something you believe to be true
- There are two forms of the `assert` statement:

```
assert booleanExpression;

assert booleanExpression : expression;
```

- By default, Java has assertions *disabled* — that is, it ignores them

The `assert` statement II

To change this default

1. `Open Window Preferences Java Installed JREs`
2. Select the JRE you are using (should be 1.6.*something* or 1.7.*something*)
3. Click `Edit...`
4. For Default VM Arguments, enter `ea` (enable assertions)
5. Click `OK` (twice) to finish

Assertions or Exceptions? I

Exceptions

- Are used to catch error conditions "from outside", such as trying to read a file that doesn't exist
- Can also be used to check parameters, or the state of an object, to warn users of your class that they have done something wrong

Assertions or Exceptions? II

The `assert` statement

- Is used as *live* documentation, to specify something that you believe will always be true
- If you can think of circumstances where it won't be true, you should *not* be using `assert`
- But because assert is easier than exceptions, it can sometimes be used for error checking in your own `private` methods

Assertions or Exceptions? III

Philosophy:

You have to get your own class correct; you can't expect other classes to be correct

- Assertions are *internal*
- Exceptions are *external*

Additions in Java 7

- Multi-catch exceptions
- Automatic resource management
- The `Closable` interface

Multi-catch exceptions

```
try {
  boolean test = true;
  if (test) {
    throw new Exception1();
  } else {
    throw new Exception2(); }
  } catch (Exception1 | Exception2 e) {
    System.out.println("Exception type = " + e.getClass());
  }
}
```

Resource Management: pre-JDK7

Manually closing resources can be tricky and tedious

```
public void copy(String src, String dest) throws IOException {
     InputStream in = new FileInputStream(src);
     try {
        OutputStream out = new FileOutputStream(dest);
        try {
           byte[] buf = new byte[8 * 1024];
           int n;
           while ((n = in.read(buf)) >= 0)
              out.write(buf, 0, n);
        } finally {
           out.close();
        }
     } finally {
        in.close();
     }
}
```

Automatic Resource Management

```
static void copy(String src, String dest) throws IOException {
    try (InputStream  in  = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest)) {
        byte[] buf = new byte[8192];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    } //in and out closes
}
```

The `Closable` interface

Implemented by all auto close resources

```
package java.lang.auto;

/**
 * A resource that must be closed
 * when it is no longer needed.
 */
public interface AutoCloseable {
   void close() throws Exception;
}
```

```
package java.io;
public interface Closeable extends AutoCloseable {
   void close() throws IOException;
}
```

Summary: Exceptions I

- To signal an exceptional condition, use the `throw` statement to throw an exception object
- When you `throw` an exception, processing continues in an exception handler
- Place statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause
- *Checked exceptions* are due to external circumstances that the programmer cannot prevent
  - The compiler checks that your program handles these exceptions

Summary: Exceptions II

- Add a `throws` clause to a method that can throw a checked exception
- Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown
- Throw an exception as soon as a problem is detected
- Catch it only when the problem can be handled
- When designing a program, ask yourself what kinds of exceptions can occur
- For each exception, you need to decide which part of your program can competently handle it

Birkbeck UNIVERSITY OF LONDON

Questions