Exceptions

Exceptions

- An exception is a kind of "extraordinary" exit from otherwise normal control flow.
- Exceptions are used to "catch" **occasional** situations for which:
  - code would get cluttered if we had to check for them repeatedly
  - the situation may come from inside a method to which we do not have access.
- Such situations "throw" the exception.

Why we need to know about this

- It helps makes your code robust (insensitive to various failures).
- Many library methods throw exceptions; you need to know how to code for these methods.
- Indicating errors by embedding print-statements is clumsy and amateurish.

Exceptions Detail

- Exceptions could include:
  - Divide by 0
  - Arithmetic overflow
  - Error input-output operation
  - Bad input format
  - and others, including programmer-defined ones
- Exceptions in Java are implemented as Exception **objects**.
- Exceptions can carry **values** indicating the **cause** of the exception.
- Exceptions should **not** be used as a normal value-returning mechanism.

Exception Lingo

- When an exception occurs it is said to be
  - "thrown"
- If an exception is thrown inside a method, it can either be:
  - "caught" (the buck stops here), or
  - "passed" (hot potato)

Exception passing

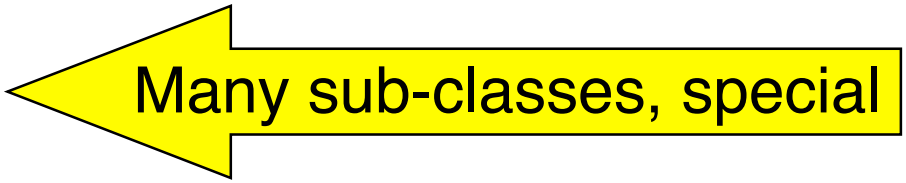
- An exception not otherwise caught will eventually get passed to the **top-level main**, at which point it will either be:
  - reported, then ignored, or
  - cause the program to terminate

Throwables

- In Java, there is a more general interface, of which Exception is a special case:
  - **Throwable** is the interface
    - **Exception** is an implementation, typically used as a base class.
    - **Error** is another implementation, usually indicating a more serious internal error.

Exception Examples

CloneNotSupportedException
DataFormatException
GeneralSecurityException
IllegalAccessException
InterruptedException
IOException
RuntimeException
UserException



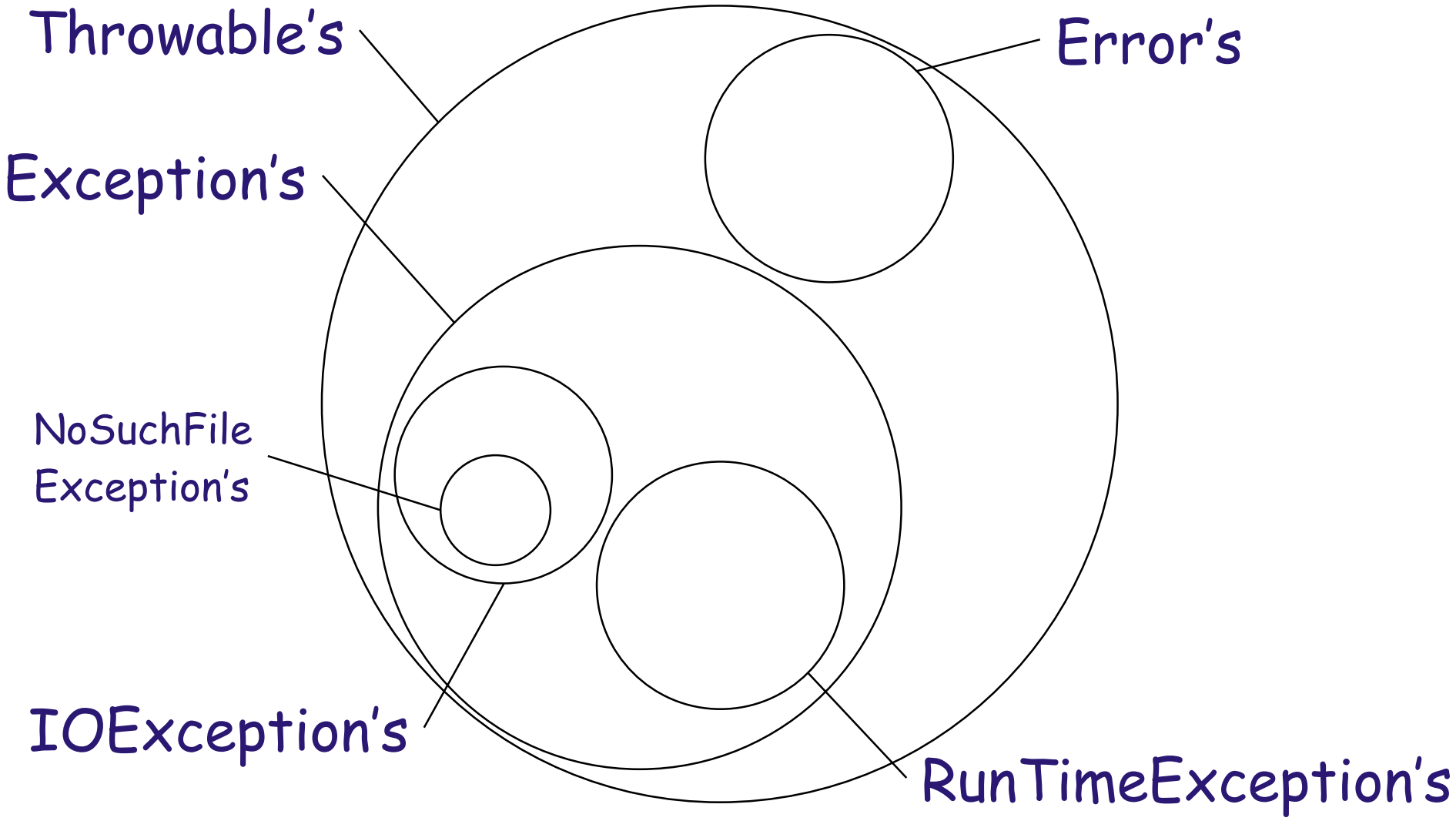
RuntimeException Sub-classes

ArithmeticException
ClassCastException
EmptyStackException
IllegalArgumentException
IndexOutOfBoundsException
NegativeArraySizeException
NoSuchElementException
NullPointerException
SystemException

RunTimeExceptions do not need to be declared in the method head.

The programmer can construct subclasses of RunTimeExceptions.

Throwable Hierarchy



Typical Exception Handling

- Keywords are:
  - **`try`**: execute some code (known as a **try-block**) in which an exception might be thrown
  - **`catch`**: handle the exception if it is thrown
  - **`finally`**: *optional* code executed after a try-bock **whether or not** an exception was thrown

Problem: Opening a File

- File named might not exist
- Attempting to open a non-existent file will throw an

  `FileNotFoundException`

- Need to catch, or will not compile

Correct Version

```
InputStream inStream = System.in;

if( arg.length > 0 )
  {
  String filename = arg[0];
  try
    {
    inStream = new FileInputStream(filename);
    }
  catch( FileNotFoundException e )
    {
    System.err.println("*** unable to open file: " + filename);
    System.exit(1);
    }
  }
```

"try" block

"catch" phrase

terminates program

Generally >1 Exception Type

```
try
  {
  . . .
  }
catch( ExceptionType1 e )
  {
  . . .
  }
catch( ExceptionType2 e )
  {
  . . .
  }
. . .

finally
  {
  . . .
  }
```

optional, always executed if present whether or not there is an exception

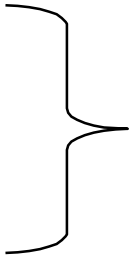
`finally` Code Example

```
class Contains
{
public static void main(String arg[])
  {
  if( arg.length != 2 )
    {
    System.err.println("usage: filename word"
    System.exit(1);
    }

  String filename = arg[0];
  String word = arg[1];

  FileInputStream stream = null;
  StreamTokenizer input;
  try
    {
    stream = new FileInputStream(filename);
    input = new StreamTokenizer(stream);
    boolean found = false;
    while( input.nextToken() != StreamTokenizer.TT_EOF )
      {
      if( word.equals(input.sval) )
        {
        found = true;
        break;
        }
      }
```

```
  catch( IOException e )
    {
    System.err.println("IO exception opening or reading file "
    }
  finally
    {
    if( stream != null )
      {
      try
        {
        stream.close();
        }
      catch( IOException e )
        {
        System.err.println("IO exception closing file " + file
        }
      }
    }
  }
}
```

Catch-all for Exceptions

```
try
  {
  . . .
  }
catch( ExceptionType1 e )
  {
  . . .
  }
catch( Exception e )
  {
  . . .
  }
```

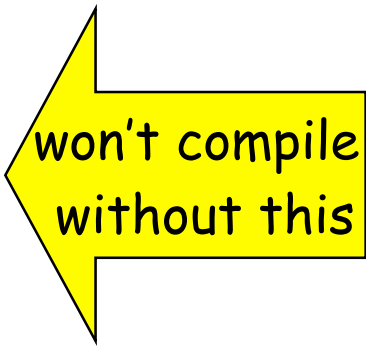


catches everything but `ExceptionType1`

Declaring

If a method throws an exception, this fact must be **declared**:

```
void myMethod() throws MyException
{
… throw new MyException(msg);
}
```



Declaring

If a method *passes* on an exception, this fact must be **declared:**

```
void myMethod() throws FileNotFoundException
{
inStream = new FileInputStream(filename);
}
```

Stopping the buck from being passed

If a method catches an exception, do **not** declare that it throws it, unless it does:

```
void myMethod() throws FileNotFoundException
{
try
  {
   inStream = new FileInputStream(filename);
  }
catch( FileNotFoundException e)
  {
  }
}
```

Exception on Declaration Rule

- The subclass of `RuntimeException` does not have to be declared.

- Example: `OpenListException` (used with OpenList) is so declared

```
class OpenListException extends RuntimeException
{
. . .
}
```