Emergency Room: Exceptions

- Robustness of Prgrams
- Errors and Reactions
  - Exceptions
- Java Exception Types

Be aware!

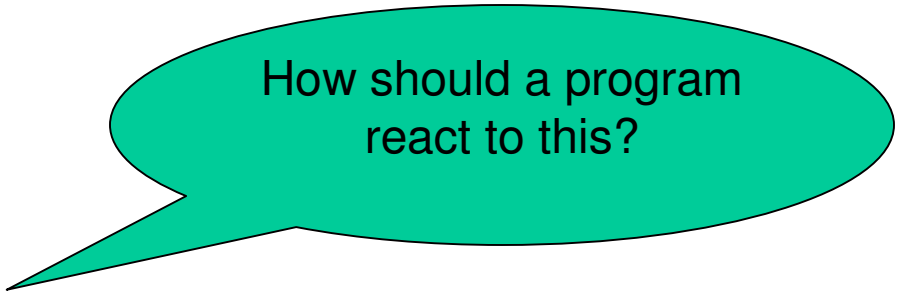
If anything can go wrong, it will go wrong. (Murphy's Law)

- *Users can be incredibly stupid.*
- *Devices and networks tend to fail.*
- *Programmers make mistakes or work carelessly.*

Good program design
- expects errors to happen and
- deals with them adequately.

Robustness

- *Apollo13:*
  "A failure of the mission is not acceptable."
- Most errors can be foreseen:
  - Printer:
    - lack of paper
    - paper jam
    - lack of toner
    - Spool overflow
    - network disruption
    - Software error
  - Arithmetics
    - number over- or underflow
    - Division by zero



Typical Runtime Errors:

If anything can go wrong, it will go wrong. (Murphys Law)

- logical errors (too few/too many loop passes, wrong conditions..)
- arithmetic errors (division by zero, number overflow)
- general value errors (type errors, null references, illegal values)

- illegal operations (freeing a port while used)
- I/O errors (file not found or corrupted, device not ready)
- network errors (device out of reach)
- device error (printer not ready)

Object Usage...

```
public interface Counter {…}
public class BasicCounter implements Counter {… }
public class SettableCounter extends BasicCounter {...}

public class Tester {
    Counter c1, c2 = new BasicCounter();
    public static void main(String[] args) {
       c1.count();                          // will fail!
       ((SettableCounter)c2).reset();       // will fail!
    }
 }
```

Integer Reading...

```
public int readInt() {
    int number;             // result
    String input = ""; // reading buffer

    int c = System.in.read(); // may fail
    while (c != '\n' & c != '\r')
    {     input += c;
          c = System.in.read(); // may fail
    }
    number = Integer.parseInt(input); // may fail
    return number;
}
```

Array Usage ☺

```
public class Storage {
      private Bill[] bills = new Bill[100];

      public void addBill(int index, Bill newbill)
      { bills[index] = newbill; }
}
```

```
Storage myBills = new Storage();
int index;
// ...
myBills.addBill(index, new Bill("ALDI",10.39));
                                        // may fail!!
```

Basic Questions for Components

? Can the component be unavailable?

? What if so?

? Can the component behave unexpectedly?

? What if so?

? Can the component encounter difficult circumstances?

? What if so?

Basic Questions for Operations

? Can the operation fail?

? What if so?

? Can the operation behave unexpectedly?

? What if so?

? Can you check on operation entry whether problems will occur?

? What if so?

Categorizing Errors

- catastrophic
  - irrepairable
  - e.g. lost connection to data base
    continuation useless
  - but prepare for restart (save data, close files,...)
- simple
  - avoidable through simple local checks
  - e.g. division by zero
  - Please check your checks ☺
- tricky
  - non-local cause
  - local mending impossible/useless
  - program continuation desirable
  - ***→ this is where we need a strategy!***

Possible Reactions

- ✓local repair / bridge
- ✓central repair / bridge
- ✓organized retreat

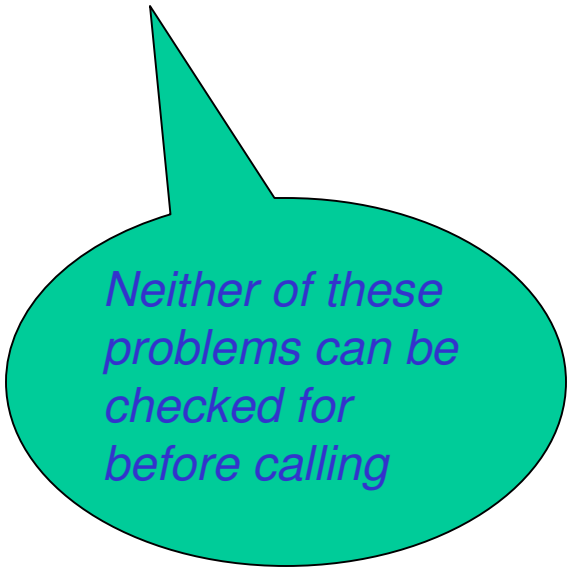
For Example: Reading

Reading an integer:

→ Input does not work

→ Input is not an Integer

→ Input is smaller/bigger than allowed



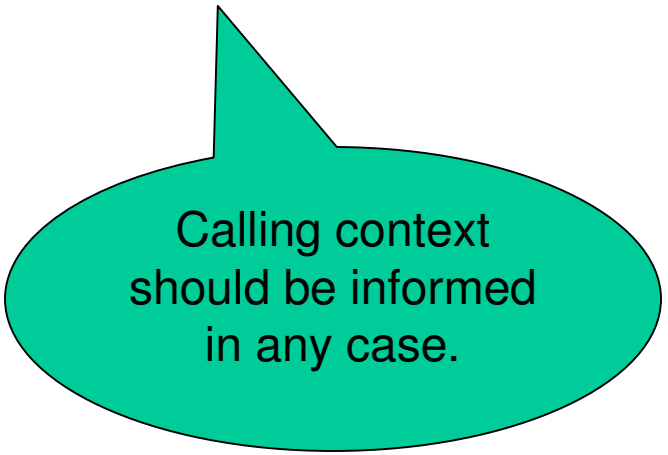
Reaction Options

→ return NULL value

→ return special value

→ signal error situation to context

→ retry automatically



Integer Reading

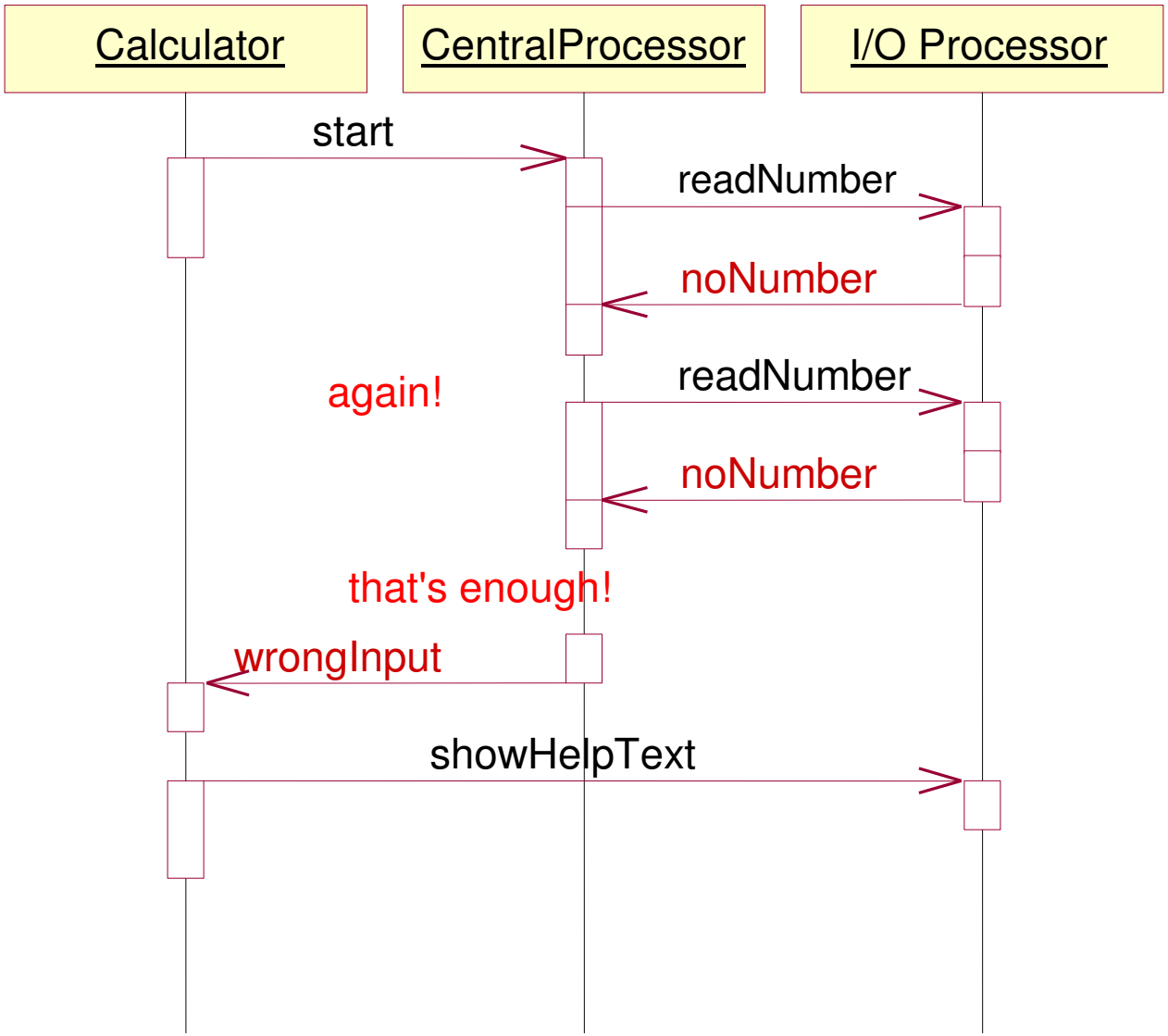
- Assume the thing read is not an integer.
- Pass this error information to the calling context
- Let the context decide on handling it:
  - repeat reading (maybe 3x)
  - use a default value instead
  - pass the error information on to ist calling context

Error Info

- Which error infos are to be passed on?
  - Error Type (for internal use)
  - Error Description (for printing)
  - Error Position in Program text
  - Values of Variables involved

- The context can use it to
  - Repair or bridge the error
  - Generate useful error messages

Example: Calculator



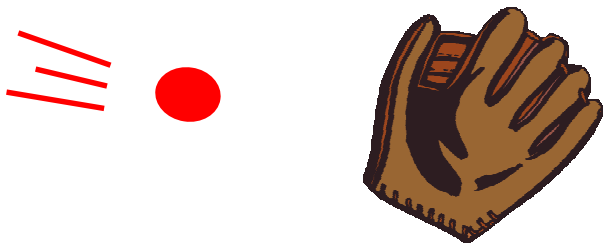
Exceptions (Ausnahmen)

Handling of exceptional situations in two steps:

- When an exception occurs:
  - cancel method execution
  - fill an error info bundle an pass it on

throw

- When you receive an error info bundle:
  - Either pick up the bundle and repair the error
  - or pass it on



catch - or not?

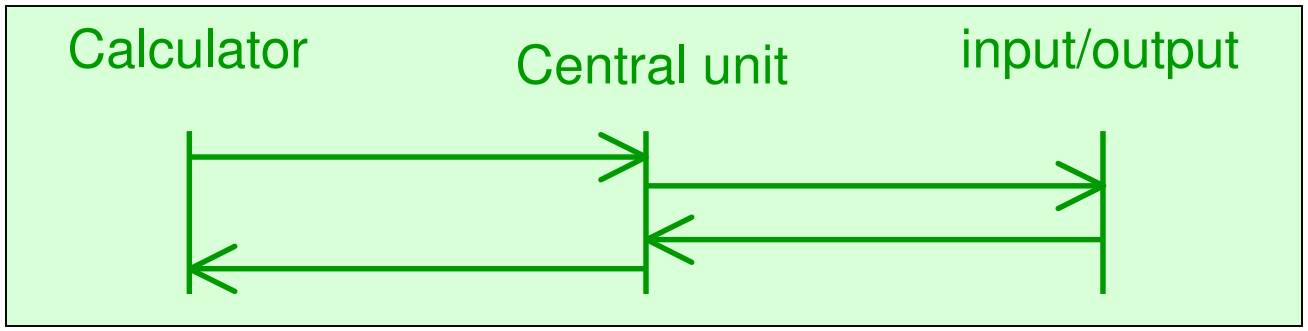
Examples

Exception	***"Handle"***	***"Reject"***
receiving a phone call with unreadable caller number	accept, display "unknown number"	reject call, info "sender number damaged"
line disturbance during password input, 1 char not readable	accept if all other chars are correct	reject, info "error reading password"
missed a train	wait for next train	give up, go home
fax input incomplete	print received amount	request re-sending
parcel received with damaged wrapping paper	accept + report	reject as damaged
coffee served with broken cookie	accept, ignore fault	reject + complain

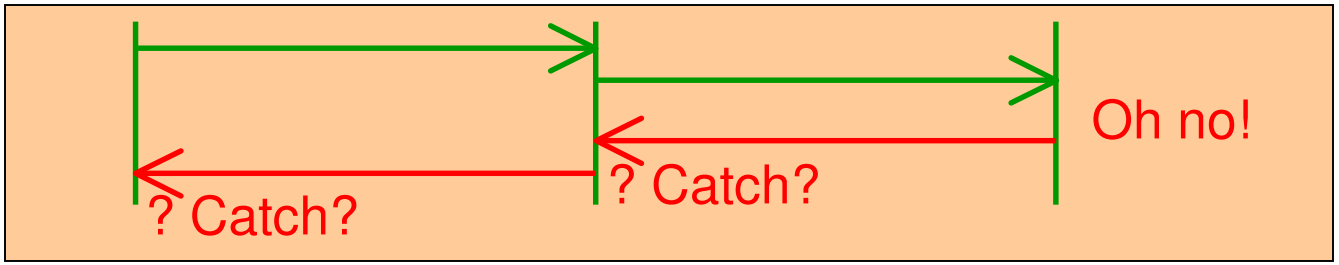
The Exception Mechanism

Example: Calculator reading number

normal flow of control



control flow with input exception



catch

Exception

throw

The Exception Class

- Exceptions are *throwable* Java objects:

```
throw new Exception()
(usage like return)
```

- The Exception class allows you to bundle information (in the constructor):
  - a text (readable with getMessage)
  - another Exception (readable with getCause)
  - Constructors
    Exception(),
    Exception(String msg),
    Exception(String msg, Exception cause)

- Typically, you extend Exception:
  - to form a specific object type, which can be distinguished on catching
  - in addition, you bundle infos into it.

Instead of "oh no!" say "throw"!

```
// add if not full
public int add(Vehicle vh) {
    if (number >= vehicles.length)
       throw new Exception("Garage full");
    vehicles[number] = vh;
    number++;
    return number-1;
}
// read if existing
public Vehicle read(int index) {
     if (index < 0 || index >= number)
        throw new Exception("Illegal Index");
     return vehicles[index];
}
```

Exception object
bundled with a message

Special Excetions are Better than Messages

```
// add if not full
public int add(Vehicle vh) {
    if (number >= vehicles.length)
       throw new FullException();
    vehicles[number] = vh;
    number++;
    return number-1;
}
// read if existing
public Vehicle show(int index) {
     if (index < 0 || index >= number)
        throw new IllegalIndexException();
     return vehicles[index];
}
```

Exception Subclass

```
class FullException extends Exception {}
class IllegalIndexException extends Exception {}
```

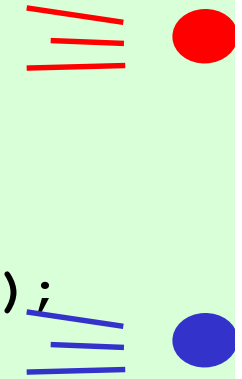
... but who catches??

```
class GarageOperator {
   VehiclePool garage = new VehiclePool();
   public boolean open = true;

   public void operate() {
      while (true) {
        if (!open) continue;
        String request = readRequest();

        if (request.isAdd()) {
            int slot = garage.add(request.car);
            issueTicket(slot);
        }
        else if (request.isShow()) {
            Vehicle veh = garage.show(request.index);
            showCar(veh);
        }


      }
  }
```



... but who catches??

```
class GarageOperator {
   VehiclePool garage = new VehiclePool();
   public boolean open = true;

   public void operate() {
      while (true) {
        if (!open) continue;
        String request = readRequest();
        try {
          if (request.isAdd()) {
            int slot = garage.add(request.car);
            issueTicket(slot);
          }
          else if (request.isShow()) {
            Vehicle veh = garage.show(request.index);
            showCar(veh);
          }
        } catch (FullException full)
            { showMessage("garage is full, sorry");  }
        catch (IllegalIndexException ilx)
            { showMessage(ilx.getMessage());  }
   }
```

Two Questions

- How do I know that I need to use try-catch?
- How can I refuse to catch and repair an exception?
  (In the fax example, a fax connector will be created even if it will not work because of an exception!)

"Attention! Lamas spit!"

```
class VehiclePool {
    ...
    // add if not full
    public int add(Vehicle vh) throws FullException {
        if (number >= vehicles.length)
                    throw new FullException();
        vehicles[number] = vh;
        number++;
        return number-1;
    }
}
```

The throws clause is part of the signature;
a possible Exception must be listed
and must not be ignored (i.e.try-catch is required)

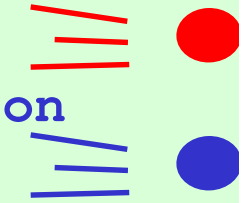
... and if catching is refused?

```
class GarageOperator {
   VehiclePool garage = new VehiclePool();
   public boolean open = true;

   public void operate() throws FullException,
                                IllegalIndexException
   {  while (true) {
         if (!open) continue;
         Request request = readRequest();

         if (request.isAdd()) {
              int slot = garage.add(request.car);
              issueTicket(slot);
         }
         else if (request.isShow()) {
              Vehicle veh = garage.show(request.index);
              showCar(veh);
         }

      }
   }
```



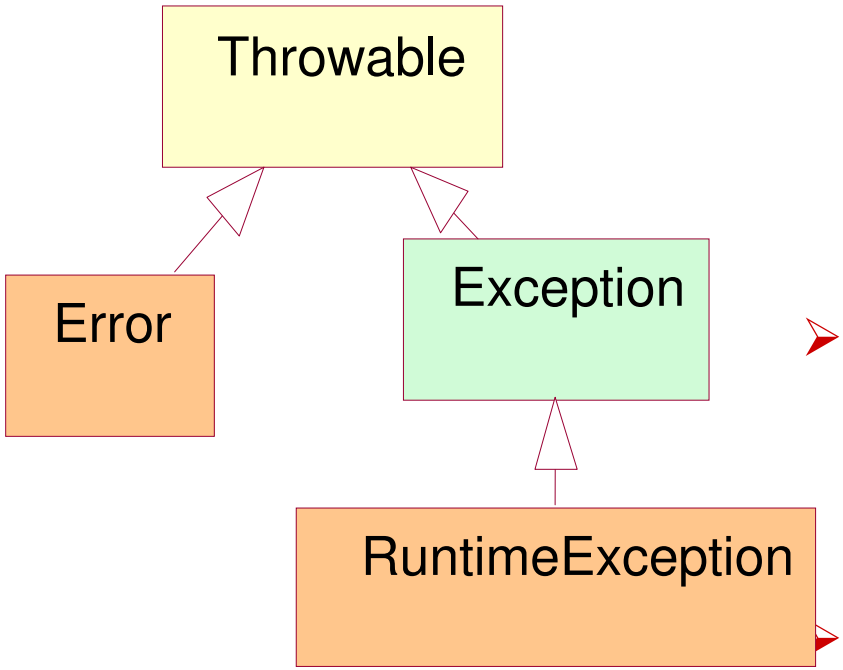
Passing on of exception responsibility through specification (throws-clause)

Reading an Integer...

```
 public int readInt()
            throws IOException, NumberFormatException
{
     int number;            // result
     String input = "";     // reading buffer

     int c = System.in.read(); // throws IOException
     while (c != '\n' & c != '\r')
     {       input += c;
             c = System.in.read(); // throws IOException
     }
     number = Integer.parseInt(input);
                                  // throws NumberFormatEx.
     return number;
 }
```

Exception Types



- Exceptions- are to be used for repairable Errors.
  They must be either caught or passed on through specification.
  → checked exceptions

- Errors –are meant for system errors; typically they cannot be handled.
  Your program may catch them, but does not need to.

- RuntimeExceptions- are meant for program errors that cannot be repaired.
  Your program may catch them, but does not need to.

Array Exceptions:

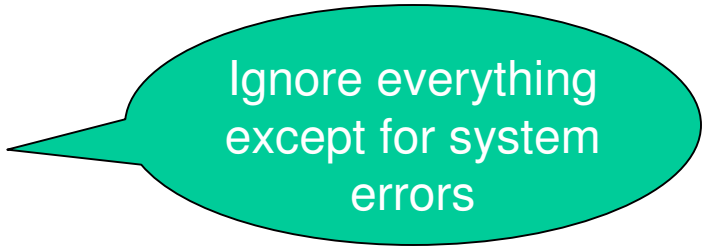
- Access with illegal index →
  ArrayIndexOutOfBoundsException.
  (Type RuntimeException – catching not enforced)

```
try { car[i] = new Car(); }
catch (ArrayIndexOutOfBoundsException e)
{ System.err.println
        ("there are only 6 garages!");
}
```

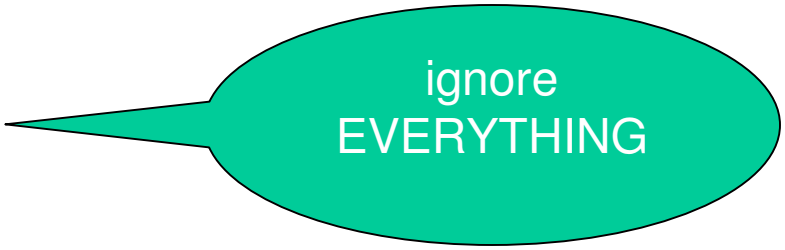
```
Storage myBills = new Storage();
int index;
// ...
try {myBills.addBill(index, new Bill("ALDI",10.39)); }
catch(ArrayIndexOutOfBoundsException x)
{ System.err.println
      (x.toString() + "in myBills.addBill at "+index);
}
```

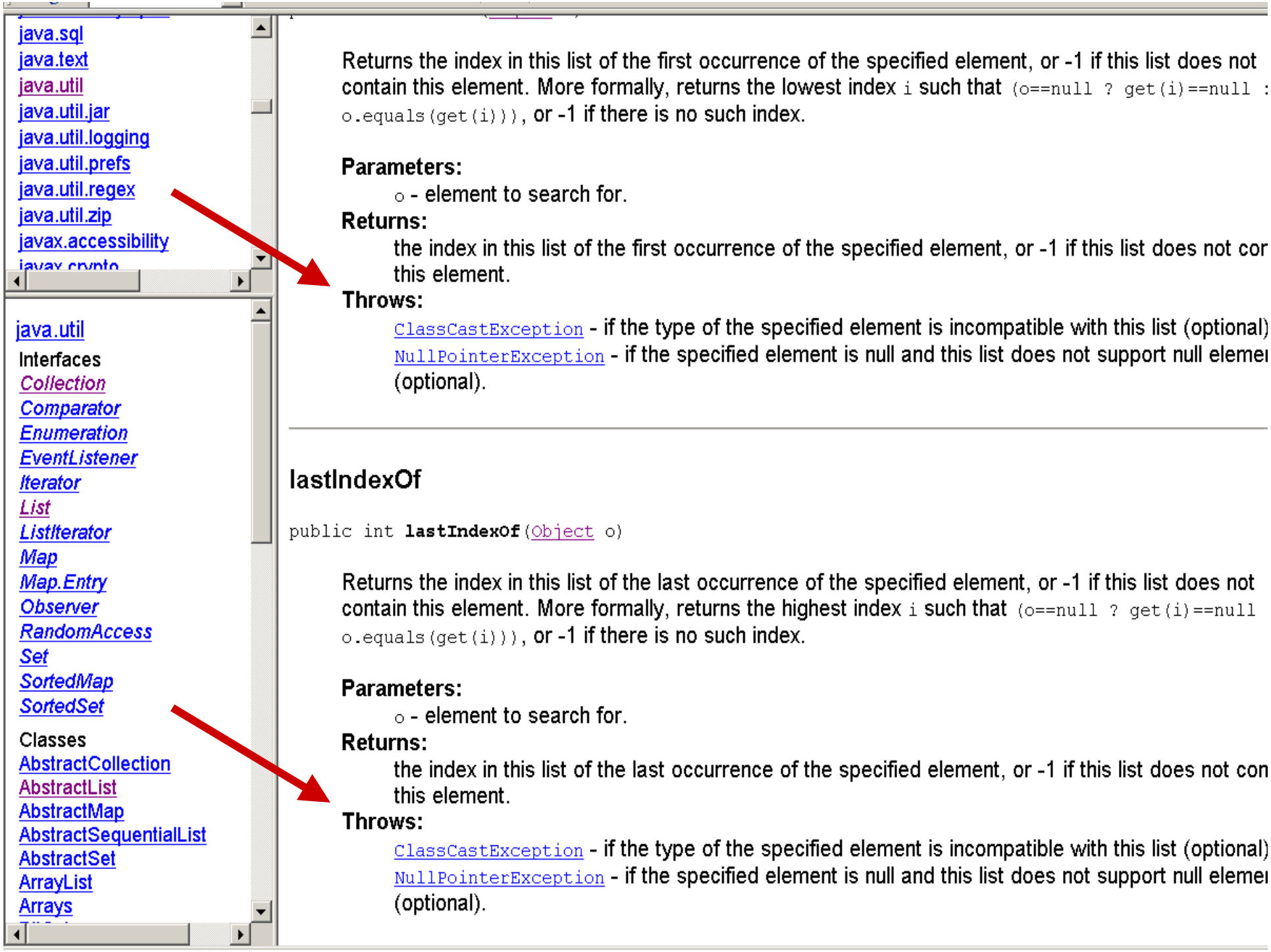
Exception-"Firewall"?

```
try {  // ...
}
catch (Exception ex) { }
```




```
try {  // ...
}
catch (Throwable th) { }
```




java.sql
java.text
java.util
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.zip
javax.accessibility
java.util
Interfaces
Collection
Comparator
Enumeration
EventListener
Iterator
List
ListIterator
Map
Map.Entry
Observer
RandomAccess
Set
SortedMap
SortedSet
Classes
AbstractCollection
AbstractList
AbstractMap
AbstractSequentialList
AbstractSet
ArrayList
Arrays
Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element. More formally, returns the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index.
Parameters:
o - element to search for.
Returns:
the index in this list of the first occurrence of the specified element, or -1 if this list does not cor
this element.
Throws:
ClassCastException - if the type of the specified element is incompatible with this list (optional)
NullPointerException - if the specified element is null and this list does not support null eleme
(optional).
lastIndexOf
public int lastIndexOf(Object o)
Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element. More formally, returns the highest index i such that (o==null ? get(i)==null o.equals(get(i))), or -1 if there is no such index.
Parameters:
o - element to search for.
Returns:
the index in this list of the last occurrence of the specified element, or -1 if this list does not con
this element.
Throws:
ClassCastException - if the type of the specified element is incompatible with this list (optional)
NullPointerException - if the specified element is null and this list does not support null eleme
(optional).

Why use Console instead of System.in?

System.in.read() throws IOException, which must be caught....

```
try {
          c = System.in.read();
}
catch (IOException ex) {
          System.out.println
                    (ex.toString() + ex.getMesage());
}
```

...That's enough about misbehaving objects in general...

Let's go back and apply it to (mis)behaving data collections.