南京大学

契约式设计 vs. 异常

DbC vs. Exception

异常 (Exception)

- 问题
  - 何谓“异常（Exception）”？
    - 一种情况要“异常” 到什么程度才算“异常” ？
  - 为什么要引入异常处理机制？
    - Robustness? Readability?
  - 如何进行异常处理？
    - Mandatory or Optional
- 不同的认识，不同的答案
  - Java/C++/C#
  - Eiffel

Java异常机制

- Java 对 Exception 的界定较宽
    - 因而需认真区分不同类型的 Exceptions
- Java的Exception机制回顾
    - try/catch/finally
    - throw Exceptions
    - 自定义Exceptions
- Java Exception与Design by Contract
- Exception的转换

什么是异常？

Many “exceptional” things can happen during the running of a program, e.g.:

checked

- User mis-types input
- Web page not available
- File not found

unchecked

- Array index out of bounds
- Method called on a null object
- Divide by zero

sys errors

- Out of memory
- Bug in the actual language implementation

Exceptions are unexpected conditions in programs.

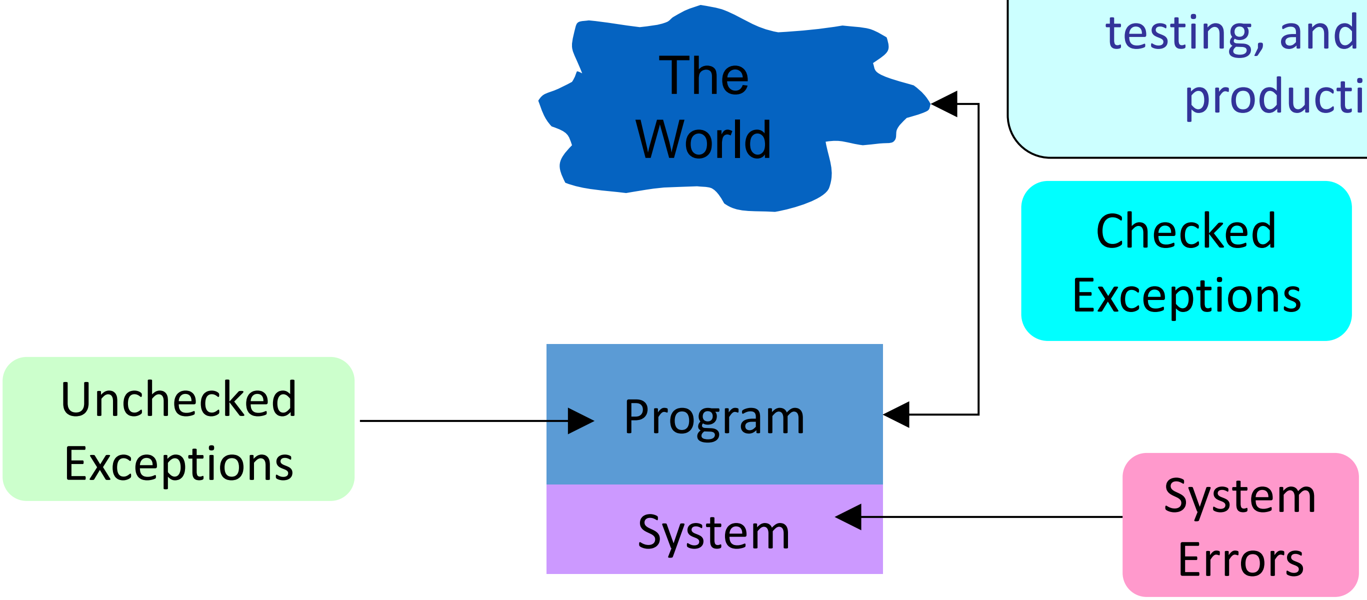
三类异常

- *checked* exceptions — Problems to do with the program's interaction with “the world”.

The world is unpredictable, so we would expect these things to happen in production code, and so need to handle them.

- *unchecked* exceptions — Problems within the program itself (i.e. violations of the contract, or bugs).
- *system errors* — Problems with the underlying system. These are outside our control.

These should be removed by testing, and not occur in production code.



The World

Checked Exceptions

Unchecked Exceptions

Program

System

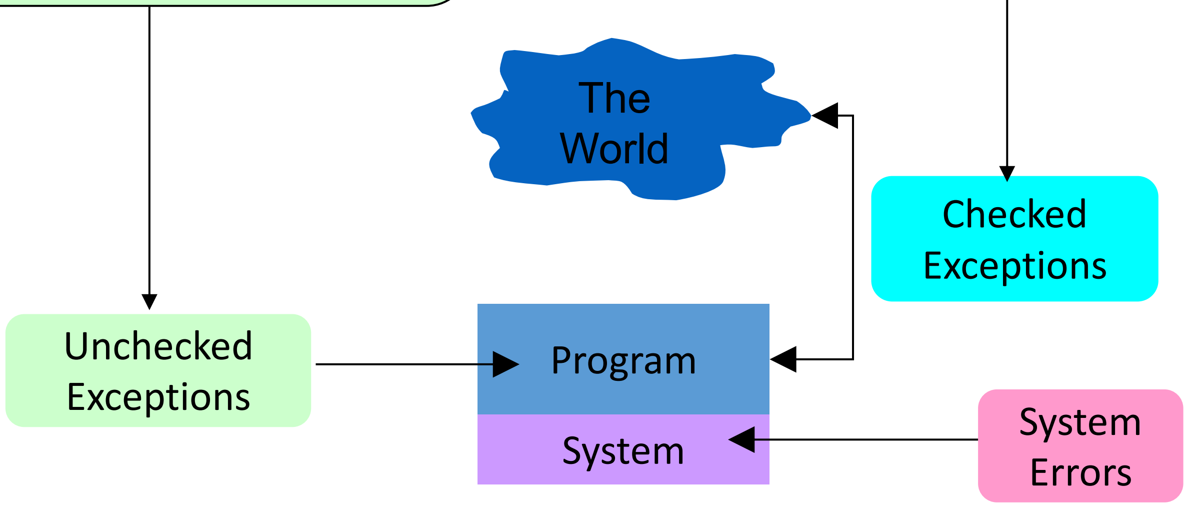
System Errors

受检和非受检异常

an important distinction, which the Java **Exception** class hierarchy does make, but in a rather confusing way

it's normal to let these just crash the program so we can debug it.

we would normally check for these, and deal with them when they occur.



The World

Checked Exceptions

Unchecked Exceptions

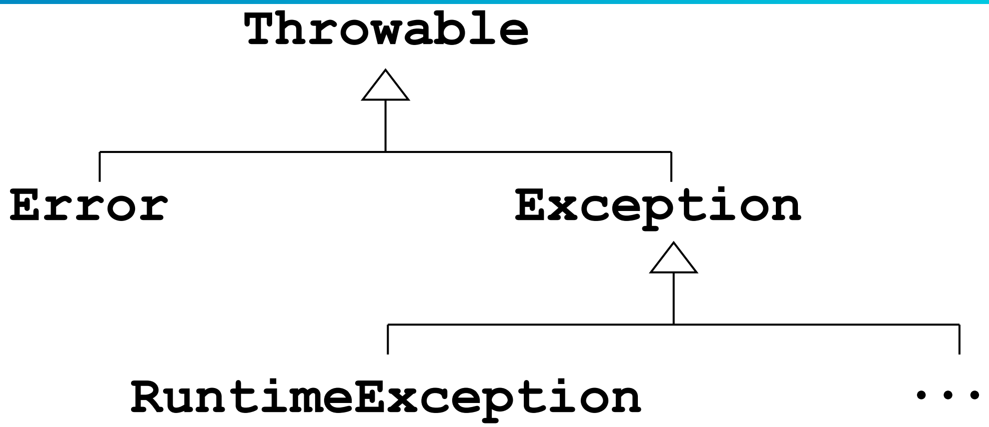
Program

System

System Errors

Exception handling is the business of handling these things appropriately.

Java中的异常层次



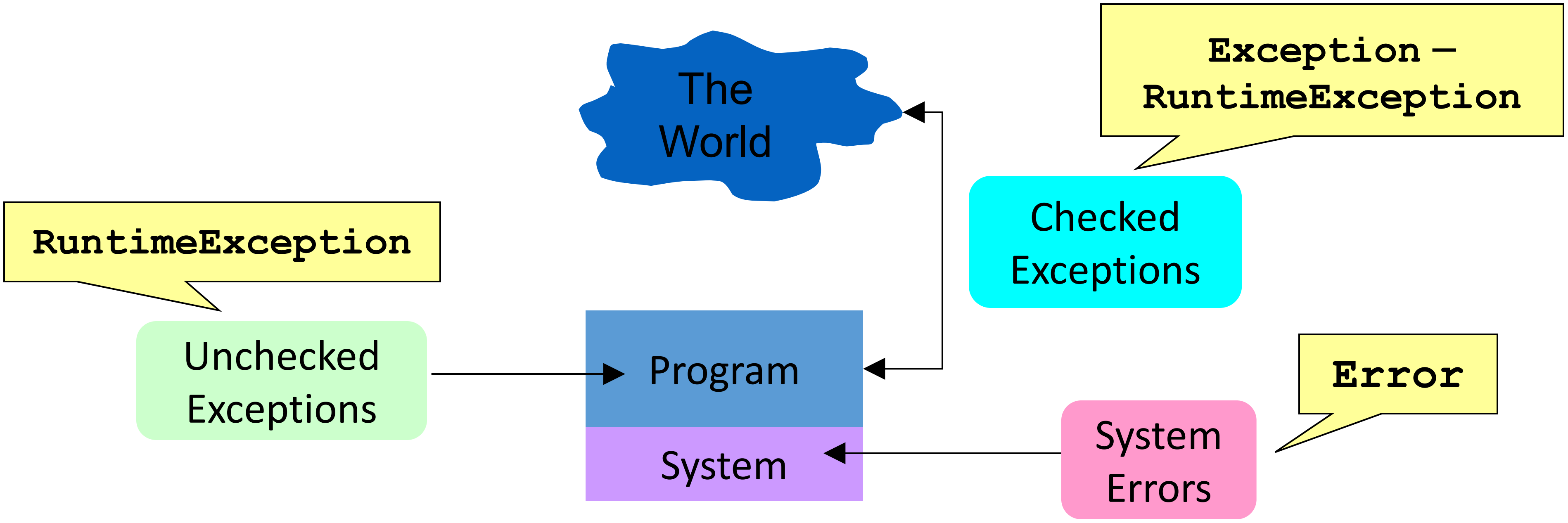
Throwable

Error

Exception

RuntimeException

…



Exception – RuntimeException

Checked Exceptions

The World

RuntimeException

Unchecked Exceptions

Program

System

Error

System Errors

我们希望什么样的异常？

e.g. no pointers to deallocated memory

Ideally, a language (and its implementation) should:

- Restrict the set of possible exceptions to “reasonable” ones
- Indicate where they happened, and distinguish between them

and not map them all to “bus error” etc.

- Allow exceptions to be dealt with in a different place in the code from where they occur

so normal case code can be written cleanly without having to worry about them

so we **`throw`** exceptions where they occur, and **`catch`** them where we want to deal with them.

Ideally, we don't want non-fatal exceptions to be thrown too far — this breaks up the modularity of the program and makes it hard to reason about.

Java中的异常回顾

In Java, the basic exception handling construct is to:

- **`try`** a block of code which normally executes ok
- **`catch`** any exceptions that it generates
- **`finally`** do anything we want to do irrespective of what happened before.

If a thrown exception is not caught, it propagates out to the caller and so on until **`main`**.

If it is never caught, it terminates the program.

If a method can generate **(checked) exceptions** but does not handle them, it has to explicitly declare that it throws them so that clients know what to expect.

Java中的异常回顾

```
class UnhandledException {
        public static void main(String[] args) {
                throw new IOException();
        }
}
```

Checked exception，编译通不过

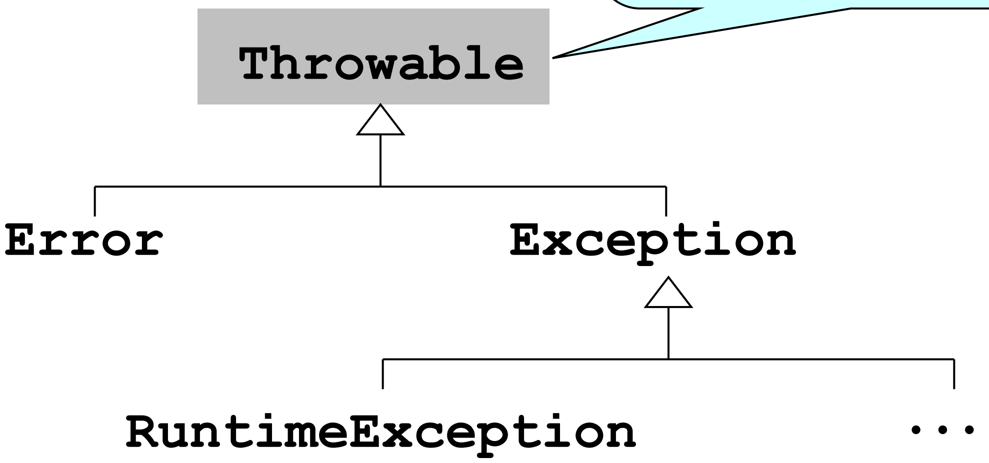
```
class UnhandledException {
        public static void main(String[] args) {
                throw new NullPointerException();
        }
}
```

Unchecked exception，编译可以通过

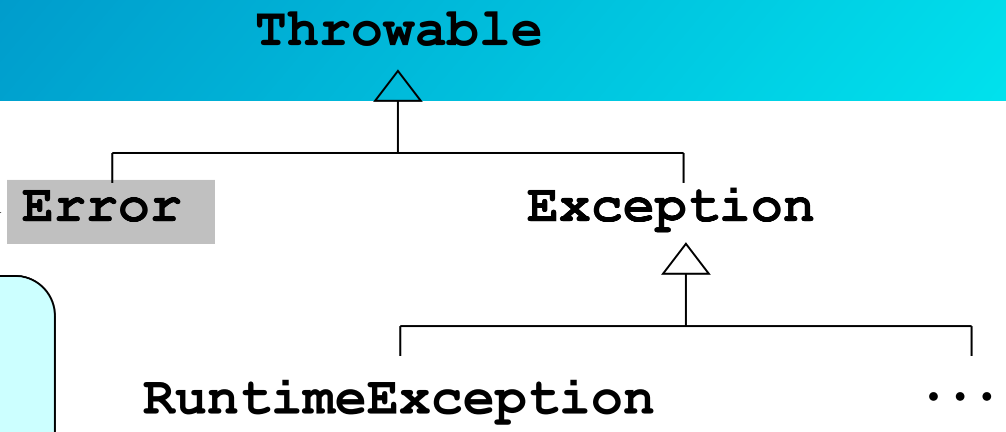
Throwable Class 的层次

The standard API defines many different exception types

- basic ones in **java.lang**
- others in other packages, especially **java.io**



Top level class is not **Exception** but **Throwable**.



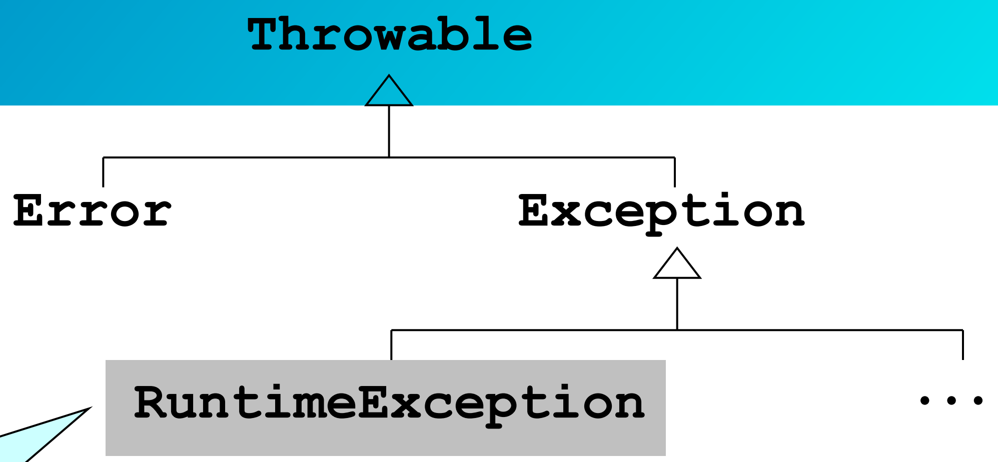
Problems with the underlying Java platform, e.g. **`VirtualMachineError`**.

User code should never explicitly generate **`Errors`**.

This is not always followed in the Java API, and you may occasionally need to say **`catch (Throwable e)`** to detect some bizarre error condition

In general, you want to catch specific sorts of (checked) exceptions, and saying **`catch (Exception e)`** is bad style.

never mind **`Throwable`**!



Throwable

Error

Exception

RuntimeException

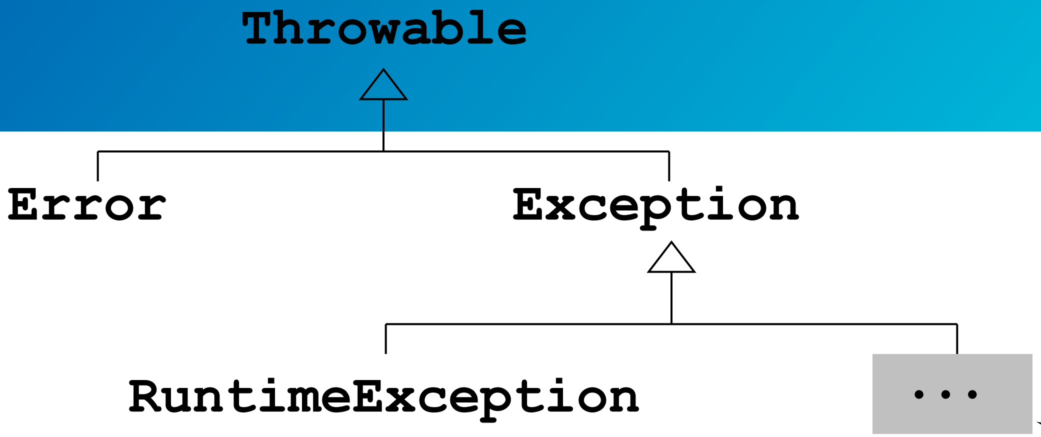
...

Unchecked exceptions are subclasses of **RuntimeException**

It would be much clearer if this was called **UncheckedException**

These include our old friends **NullPointerException** and **ArrayIndexOutOfBoundsException**.

Virtually any method can in principle throw one or more of these, so there's no requirement to declare them in **throws** clauses.



Throwable

Error

Exception

RuntimeException

...

All other exceptions are checked exceptions

It would be nice if there was a class **CheckedException**.

The most common ones are **IOExceptions** such as **FileNotFoundException**.

If you write code which can throw one of these, you must either catch it or declare that the method **throws** it.

The compiler goes to enormous lengths to enforce this, and related constraints, e.g. that all overridden versions of a method have consistent **throws** clauses.

异常处理和DbC

Exceptions are about dealing with things going wrong at runtime.

DbC is about statically defining the conditions under which code is supposed to operate.

(The two are nicely complementary.)

- Unchecked exceptions are “what happens when the contract is broken”

- Checked exceptions are expected to happen from time to time so are not contract violations.

e.g. if a precondition that an array has at least one element is broken, an **`ArrayIndexOutOfBoundsException`** will probably occur.

异常处理和DbC

So if we're going to use DbC, we ought to structure our code into:

- code which deals with The World

Methods doing this will have no (or weak) preconditions, and will deal with problems via exception handling.

- code which is insulated from The World.

This can have strong preconditions and won't throw exceptions (apart from unchecked ones during debugging).

E.g. an application involving form-filling should have

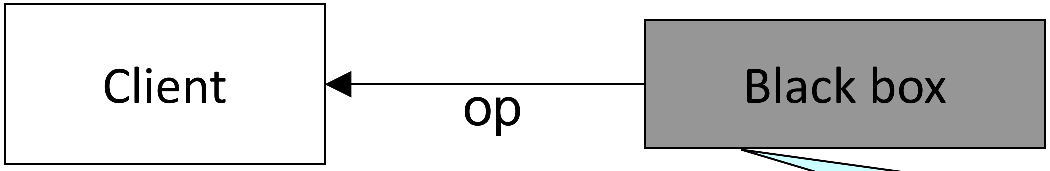
- code which validates the user input for type-correctness etc.
- code which relies on the input being correct to process it

If these two sorts of operations are mixed together, the application will probably be harder to maintain and more error-prone.

异常转换

Good OO design decouples things.

E.g. in most patterns there's a **`Client`** class which delegates operations to some “black box” in the pattern.



What happens if the black box throws a (checked) exception?

some application-specific code

may have its own exception types, but they need to be application-specific.

some application-independent library

exceptions thrown from within the library will be application-independent.

```
try {...
  obj.delegateOp(...); ...    }
catch (ApplicationIndependentException e){
throw new ApplicationSpecificException(e); }
```

`ApplicationSpecificException` will therefore have a constructor which

- takes an `ApplicationIndependentException`
- extracts the relevant information and re-casts it in application-specific terms.

例子

Consider a compiler for an oo language

In such a compiler, it's useful to have an explicit representation of the inheritance hierarchy of the source program as a tree (or graph).

which has an application-independent `Hierarchy` class which represents a hierarchy of arbitrary key-value pairs.

The key is the class name and the value is the definition of the class.

If something goes wrong in building or using a `Hierarchy` an `InvalidHierarchyException` is thrown.

e.g. a node has no parent, or there's a cycle

例子

The **`InvalidHierarchyExcetpion`** is converted to a (checked) **`IllegalInheritanceException`** resulting in a suitable error message being displayed to the user.

- If it happens later, an (unchecked) **`CompilerBrokenExpection`** is thrown instead, because it means there's a bug in the compiler.

总结

- Exception handling is an important, highly integrated, part of the Java system, one of the Really Neat Things About Java.
- Unchecked exceptions routinely occur during debugging, and make that process much easier. By product shipping time, they should no longer occur.
- Checked exceptions happen when the program's interaction with the world departs from the normal case. Make sure you handle them appropriately.

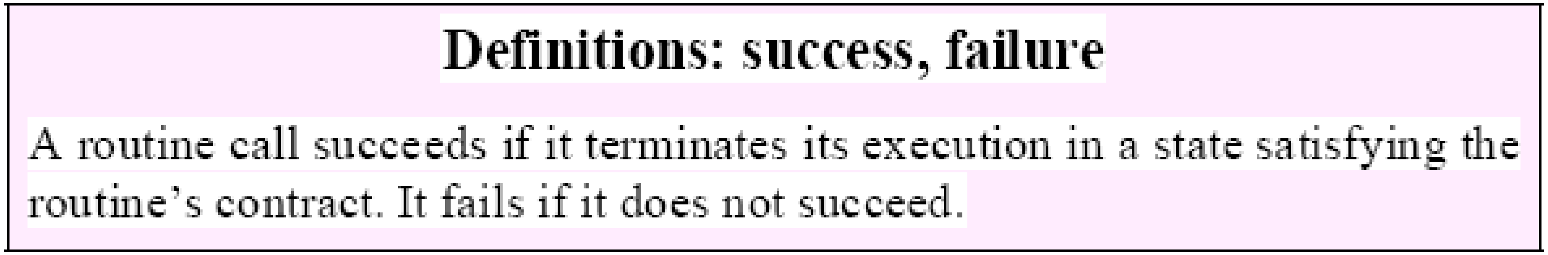
In particular, think hard about *where* to handle an exception, so as to simplify the normal-case code, but not to do the handling so far away that modularity is compromised.

- Application-independent exceptions need to be converted to application-specific ones.

Eiffel Exception机制简介

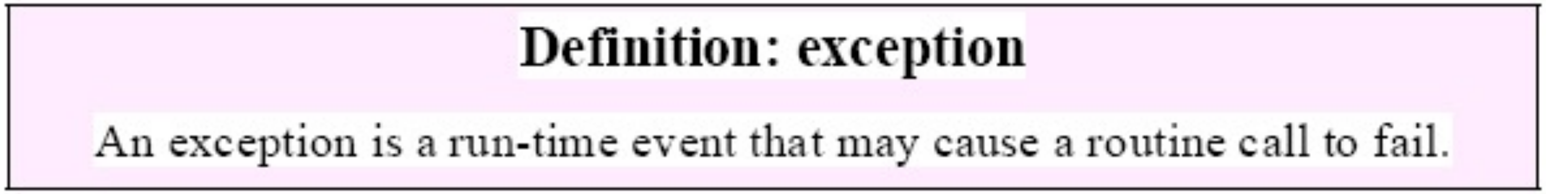
- Eiffel观点：
  - 契约破坏才会有Exception
  - Eiffel Exception机制的设计基于此观点

基本概念



Definitions: success, failure

A routine call succeeds if it terminates its execution in a state satisfying the routine's contract. It fails if it does not succeed.

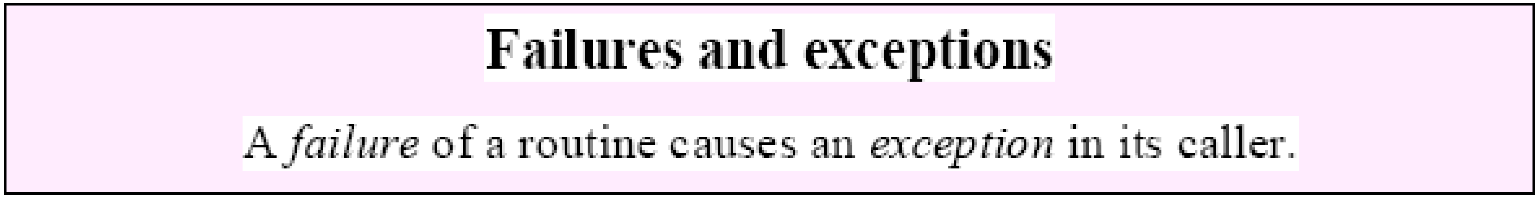


Definition: exception

An exception is a run-time event that may cause a routine call to fail.

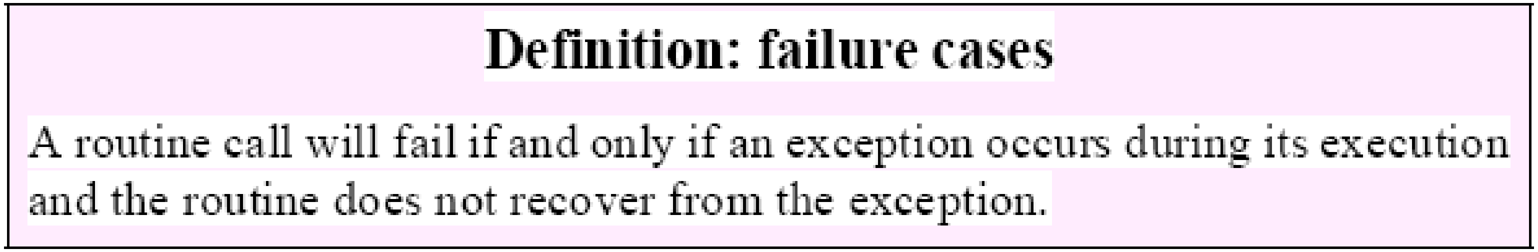
The need for exceptions arises when the contract is broken.

基本概念



Failures and exceptions

A *failure* of a routine causes an *exception* in its caller.

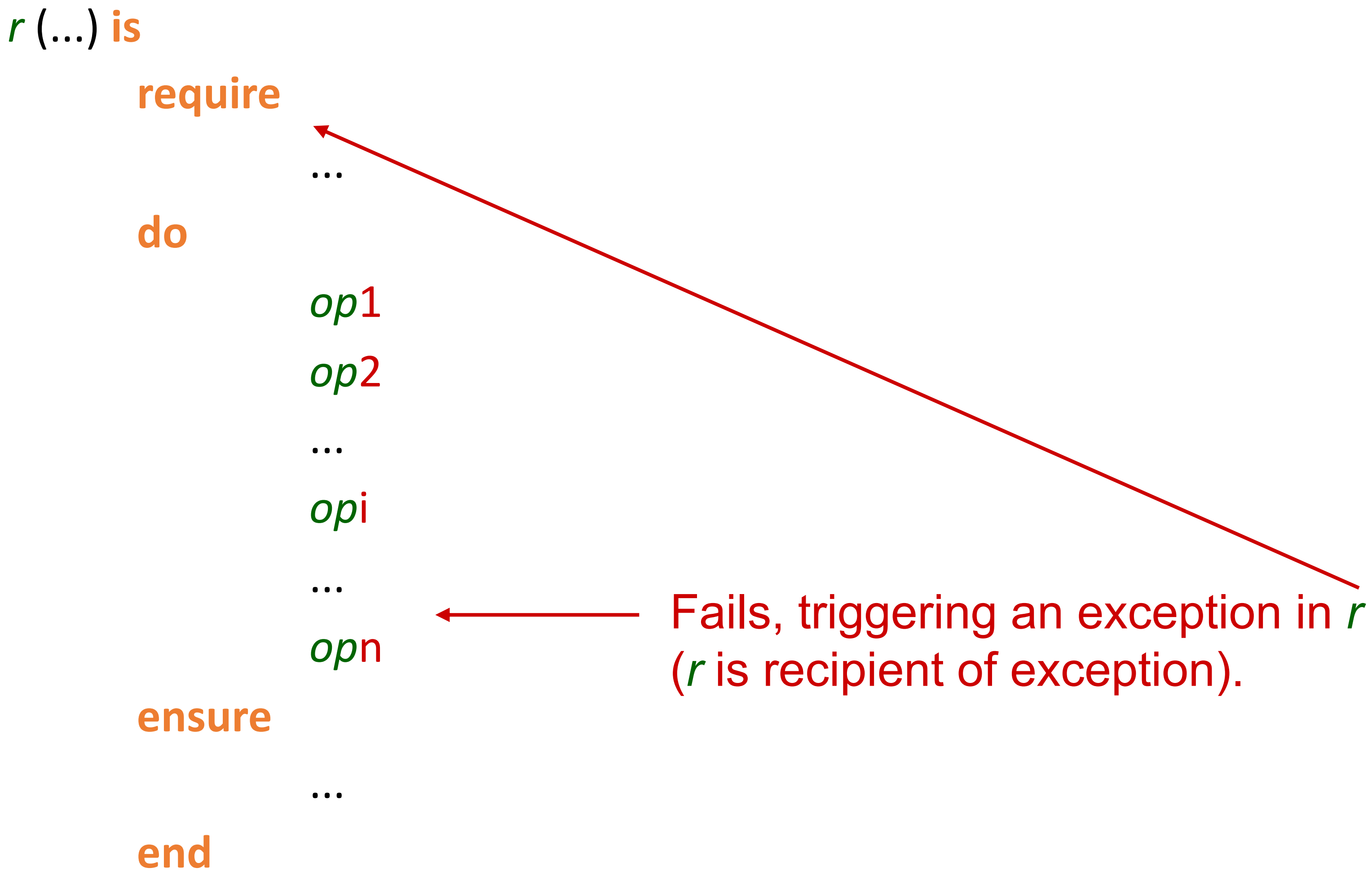


Definition: failure cases

A routine call will fail if and only if an exception occurs during its execution and the routine does not recover from the exception.

Exception: an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.

原始机制

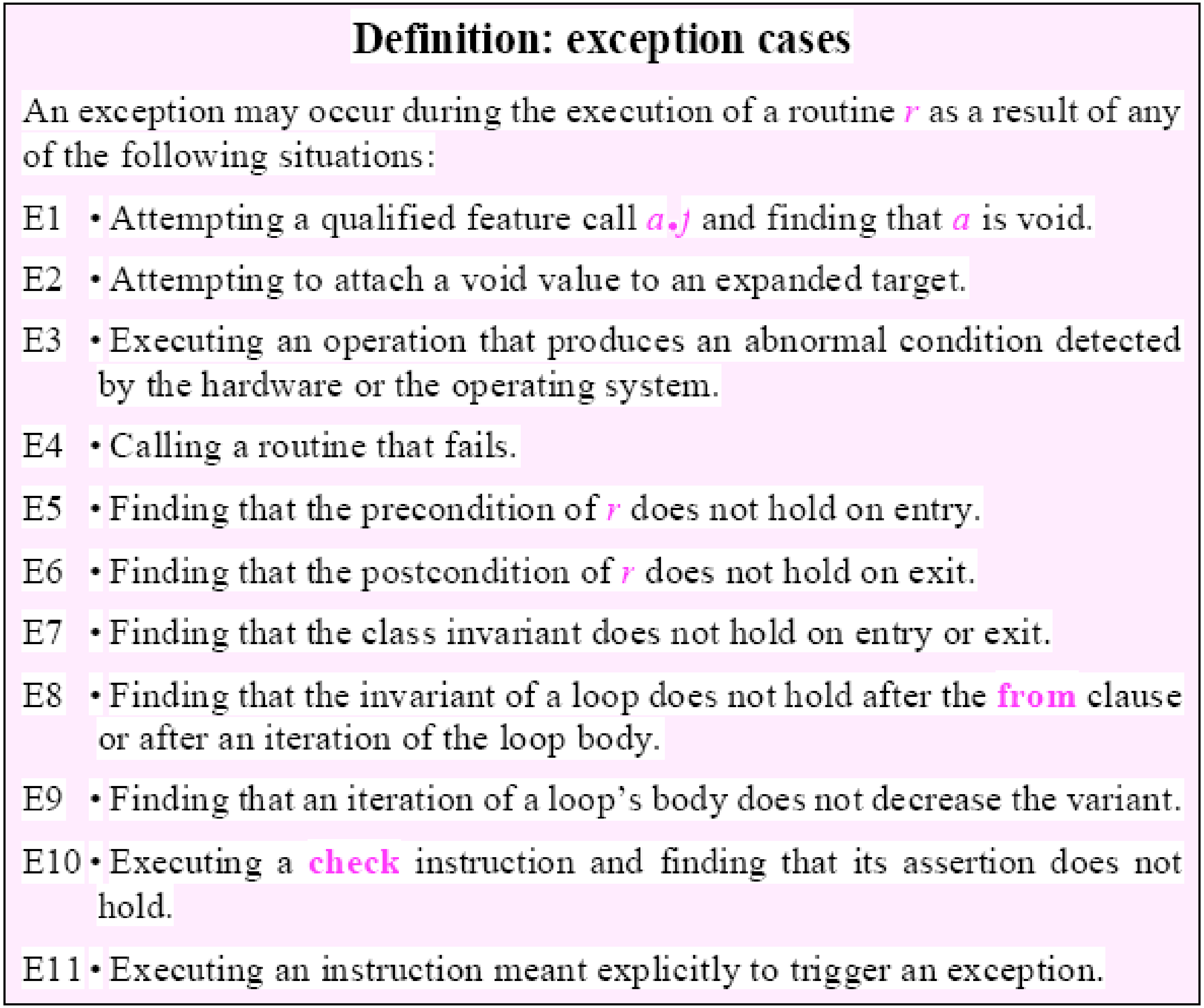


```
r (...) is
        require
                ...
        do
                op1
                op2
                ...
                opi
                ...
                opn
        ensure
                ...
        end
```

Fails, triggering an exception in *r*
(*r* is recipient of exception).

异常产生的原因

- Assertion violation
- Void call (*x*.*f* with no object attached to *x*)
- Operating system signal (arithmetic overflow, no more memory, interrupt ...)



Definition: exception cases

An exception may occur during the execution of a routine *r* as a result of any of the following situations:

E1 • Attempting a qualified feature call $a.f$ and finding that *a* is void.

E2 • Attempting to attach a void value to an expanded target.

E3 • Executing an operation that produces an abnormal condition detected by the hardware or the operating system.

E4 • Calling a routine that fails.

E5 • Finding that the precondition of *r* does not hold on entry.

E6 • Finding that the postcondition of *r* does not hold on exit.

E7 • Finding that the class invariant does not hold on entry or exit.

E8 • Finding that the invariant of a loop does not hold after the **from** clause or after an iteration of the loop body.

E9 • Finding that an iteration of a loop's body does not decrease the variant.

E10 • Executing a **check** instruction and finding that its assertion does not hold.

E11 • Executing an instruction meant explicitly to trigger an exception.

合理的处理异常

- Safe exception handling principle:
  - There are only two acceptable ways to react for the recipient of an exception:
    - Failure (Organized Panic 合理组织的“恐慌”): clean up the environment, terminate the call and report failure to the caller.
      - Panic: making sure that the caller gets an exception
      - Organized: restoring a consistent execution state
    - Retrying: Try again, using a different strategy (or repeating the same strategy).

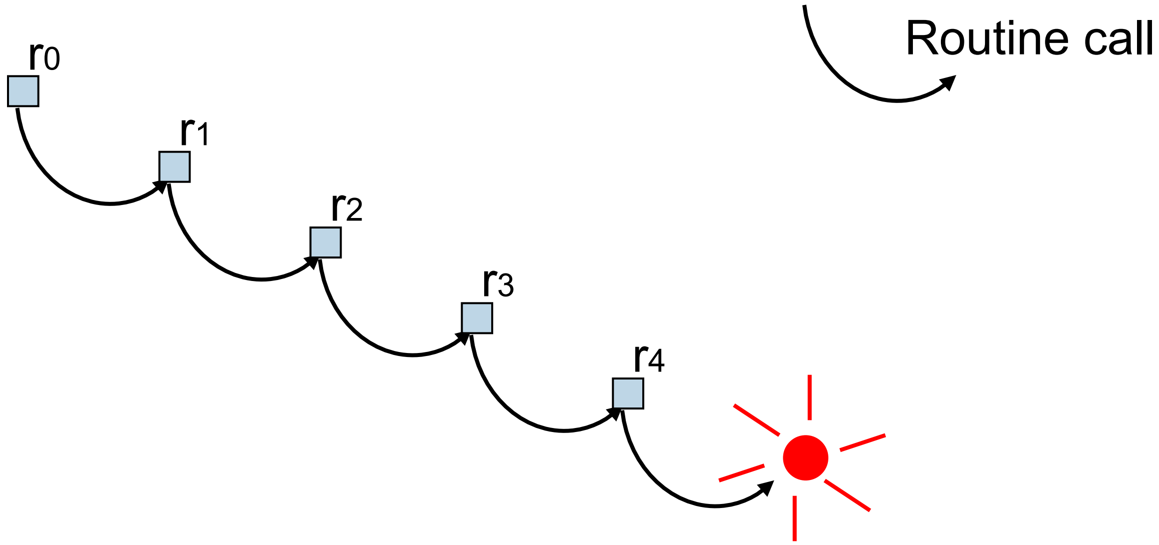
How not to do things?

(From an Ada textbook)

```
sqrt (x: REAL) return REAL is
        begin
                if x < 0.0 then
                        raise Negative;
                else
                        normal_square_root_computation;
                end
        exception
                when Negative =>
                        put ("Negative argument");
                        return;
                when others => . . .
        end; -- sqrt
```

Give up and return to the caller as if everything was fine, although not

The call chain



Exception mechanism

- Two constructs:
  - A routine may contain a rescue clause.
  - A rescue clause may contain a retry instruction.
- A rescue clause that does not execute a retry leads to failure of the routine (this is the organized panic case). → **Failure Principle**
  - If an exception occurs in a routine without rescue clause it will cause the routine to fail, trigger an exception in its caller.

Transmitting over an unreliable line (1)

```
Max_attempts: INTEGER is 100
attempt_transmission (message: STRING) is
                -- Transmit message in at most
                -- Max_attempts attempts.
        local
                failures: INTEGER
        do
                unsafe_transmit (message)
        rescue
                failures := failures + 1
                if failures < Max_attempts then
                        retry
                end
        end
```

Transmitting over an unreliable line (2)

```
Max_attempts: INTEGER is 100
failed: BOOLEAN
attempt_transmission (message: STRING) is
                -- Try to transmit message;
                -- if impossible in at most Max_attempts
                -- attempts, set failed to true.
        local
                failures: INTEGER
        do
                if failures < Max_attempts then
                        unsafe_transmit (message)
                else
                        failed := True
                end
        rescue
                failures := failures + 1
                retry
        end
```

If no exception clause (1)

- Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

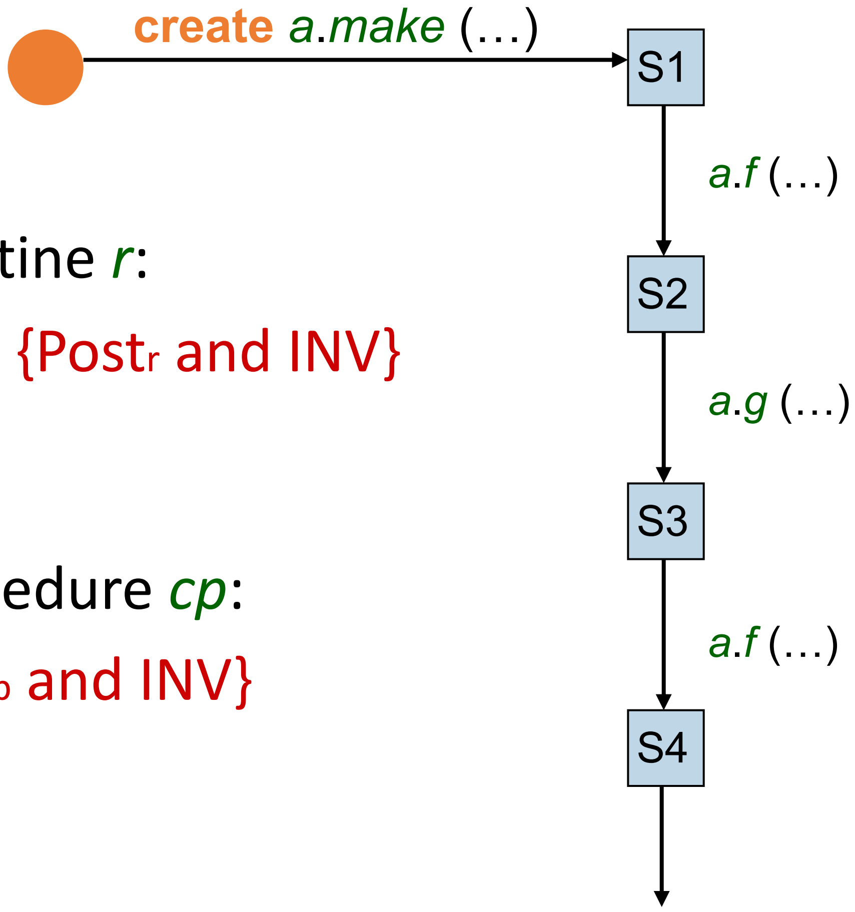
```
f (...) is
        do
                ...
        end
```

is an abbreviation for

```
f (...) is
        do
                ...
        rescue
                -- Nothing here (empty instruction list)
        end
```

- (This is a provisional rule; see next.)

The correctness of a class



- For every exported routine *r*:

  {INV and Pre$_r$} do$_r$ {Post$_r$ and INV}

- For every creation procedure *cp*:

  {Pre$_{cp}$} do$_{cp}$ {Post$_{cp}$ and INV}

Exception correctness: A quiz

- For the normal body:

  {INV and Pre$_r$} do$_r$ {Post$_r$ and INV}

- For the exception clause:

  { ??? } rescue$_r$ { ??? }

Weakest?

Strongest?

Should it be lazy?

The stronger the precond,
the easier the job

Quiz answers

- For the normal body:

  $\{INV \text{ and } Pre_r\}\ do_r\ \{Post_r \text{ and } INV\}$

- For the rescue clause:

  $\{True\}\ rescue_r\ \{INV\}$

- For the retry-introducing rescue clause:

  $\{True\}\ retry_r\ \{INV \text{ and } Pre_r\}$

Separation of roles

- Normal body: ensure the routine's contract; not directly to handle exceptions
- Rescue clause: handle exceptions, returning control to the body or (in the failure case) to the caller; not to ensure the contract.

If no exception clause (2)

- Absence of a rescue clause is equivalent to a default rescue clause:

```
f (...) is
        do
              ...
        end
```

is an abbreviation for

```
f (...) is
        do
              ...
        rescue
              default_rescue
        end
```

- The task of *default_rescue* is to restore the invariant.

For finer-grain exception handling

- Use class *EXCEPTIONS* from the Kernel Library.

- Some features:
  - *exception* (code of last exception that was triggered).
  - *assertion_violation*, etc.
  - *raise* (“exception_name”)

小结

- 使用Eiffel Exception机制来构造Robust的软件
- 使用Java/C++/C＃ Exception机制来构造
  - Robust
  - Correct
  - Easy-to-Read

Java Exception

- Java Exception机制的不当使用：
  - 引自网络论坛

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { //（5）
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery(
            "select uid, name from user");
    while (rs.next())
    {
      out.println("ID：  " + rs.getString("uid") //（6）
      "，姓名：  " + rs.getString("name"));
    }
    conn.close(); //（3）
    out.close();
}
catch(Exception ex) //（2）
{
    ex.printStackTrace(); //（1），（4）
}
```

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // （5）
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery(
            "select uid, name from user");
    while (rs.next())
    {
      out.println("ID：　" + rs.getString("uid") // （6）
      "，姓名：　" + rs.getString("name"));
     }
  conn.close(); // （3）
  out.close();
}
catch(Exception ex) // （2）
{
   ex.printStackTrace(); //（1），（4）
}
```

(1)丢弃异常！

Anti-pattern

- 丢弃异常-改正方案

1. 处理异常。针对该异常采取一些行动，例如修正问题、提醒某个人或进行其他一些处理，要根据具体的情形确定应该采取的动作。再次说明，调用printStackTrace算不上已经“处理好了异常”。
2. 重新抛出异常。处理异常的代码在分析异常之后，认为自己不能处理它，重新抛出异常也不失为一种选择。
3. 把该异常转换成另一种异常。大多数情况下，这是指把一个低级的异常转换成应用级的异常（其含义更容易被用户了解的异常）。
4. 不要捕获异常。

- 结论一：既然捕获了异常，就要对它进行适当的处理。不要捕获异常之后又把它丢弃，不予理睬。

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // (5)
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery(
            "select uid, name from user");
    while (rs.next())
    {
      out.println("ID：  " + rs.getString("uid") // (6)
      "，姓名：  " + rs.getString("name"));
    }
  conn.close(); // (3)
  out.close();
}
catch(Exception ex) // (2)
{
   ex.printStackTrace(); //(1)，(4)
}
```

(2)不指定具体的异常！

Anti-pattern

- 不指定具体的异常—改正方案
  - 找出真正的问题所在，IOException? SQLException?
- 结论二：在catch语句中尽可能指定具体的异常类型，必要时使用多个catch。不要试图处理所有可能出现的异常。

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // (5)
  Statement stat = conn.createStatement();
  ResultSet rs = stat.executeQuery(
         "select uid, name from user");
  while (rs.next())
  {
     out.println("ID：  " + rs.getString("uid") // (6)
     "，姓名：  " + rs.getString("name"));
   }
   conn.close(); // (3)
   out.close();
}
catch(Exception ex) // (2)
{
   ex.printStackTrace(); //(1)，(4)
}
```

(3)占用资源不释放！

Anti-pattern

- 占用资源不释放—改正方案
  - 异常改变了程序正常的执行流程。如果程序用到了文件、Socket、JDBC连接之类的资源，即使遇到了异常，也要正确释放占用的资源。
  - try/catch/finally
- 结论三：保证所有资源都被正确释放。充分运用finally关键词。

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // (5)
   Statement stat = conn.createStatement();
   ResultSet rs = stat.executeQuery(
          "select uid, name from user");
   while (rs.next())
   {
     out.println("ID：  " + rs.getString("uid") // (6)
     "，姓名：  " + rs.getString("name"));
    }
    conn.close(); // (3)
    out.close();
}
catch(Exception ex) // (2)
{
    ex.printStackTrace(); //(1)，（4）
}
```

(4)不说明异常的详细信息

Anti-pattern

- 不说明异常的详细信息—改正方案
  - printStackTrace的堆栈跟踪功能显示出程序运行到当前类的执行流程，但只提供了一些最基本的信息，未能说明实际导致错误的原因，同时也不易解读。
- 结论四：在异常处理模块中提供适量的错误原因信息，例如当前正在执行的类、方法和其他状态信息，包括以一种更适合阅读的方式整理和组织printStackTrace提供的信息使其易于理解和阅读。

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // ⑸
   Statement stat = conn.createStatement();
   ResultSet rs = stat.executeQuery(
          "select uid, name from user");
   while (rs.next())
   {
     out.println("ID：  " + rs.getString("uid") // ⑹
     "，姓名：  " + rs.getString("name"));
   }
  conn.close(); // ⑶
  out.close();
}
catch(Exception ex) // ⑵
{
  ex.printStackTrace(); //⑴，⑷
}
```

(5)过于庞大的try块

Anti-pattern

- 过于庞大的try块—改正方案
  - 庞大的原因？偷懒？

- 结论五：分离各个可能出现异常的段落并分别捕获其异常，尽量减小try块的体积。

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try { // (5)
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery(
            "select uid, name from user");
    while (rs.next())
    {
      out.println("ID：  " + rs.getString("uid") // (6)
      "，姓名：  " + rs.getString("name"));
    }
    conn.close(); // (3)
    out.close();
}
catch(Exception ex) // (2)
{
    ex.printStackTrace(); //(1)，(4)
}
```

如果循环中出现异常，
如何?

(6)输出数据不完整

Anti-pattern

- 输出数据不完整—改正方案
  - 对于有些系统来说，数据不完整可能比系统停止运行带来更大的损失
  - 方案1：向输出设备写一些信息，声明数据的不完整性；
  - 方案2：先缓冲要输出的数据，准备好全部数据之后再一次性输出。
- 结论六：全面考虑可能出现的异常以及这些异常对执行流程的影响。

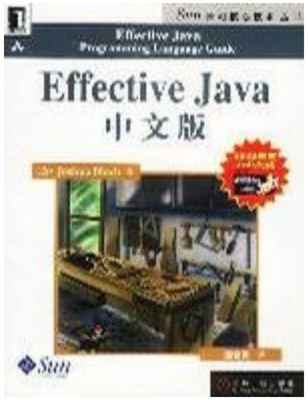
```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try {
  Statement stat = conn.createStatement();
  ResultSet rs = stat.executeQuery(
  "select uid, name from user");
  while (rs.next())
  {
    out.println("ID：  " + rs.getString("uid") +
    "，  姓名: " + rs.getString("name"));
  }
}
catch(SQLException sqlex)
{
  out.println("警告：  数据不完整");
  throw new ApplicationException(
  "读取数据时出现SQL错误", sqlex);
}
catch(IOException ioex)
{
  throw new ApplicationException(
  "写入数据时出现IO错误", ioex);
}
```

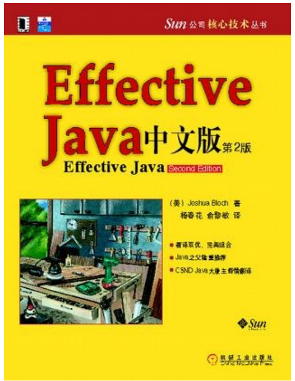
```
finally
{
  if (conn != null) {
    try {
      conn.close();
    }
    catch(SQLException sqlex2)
    {
      System.err(this.getClass().getName() +
      ".mymethod - 不能关闭数据库连接: " +
      sqlex2.toString());
    }
  }
  if (out != null) {
    try {
      out.close();
    }
    catch(IOException ioex2)
    {
      System.err(this.getClass().getName() +
      ".mymethod - 不能关闭输出文件" +
      ioex2.toString());
    }
  }
}
```

Effective Java Exceptions

- 高效Java异常处理机制：
  - 引自<<Effective Java^{TM} *second edition*>>(Joshua Bloch)
    - 语法
    - 词汇
    - **用法**

→ 高效、灵活、鲁棒、可重用的程序





9大原则

- Use exceptions only for exceptional conditions

只针对不正常的条件才使用异常

```
//Horrible abuse of exceptions. Don't ever do this!
try{
        int i = 0;
        while(true)
                range[i++].climb();
}catch(ArrayIndexOutOfBoundsException e) {
}
```

通过ArrayIndexOutOfBoundsException的手段来达到终止无限循环的目的

```
for (Mountain m : range)
        m.climb();
```

9大原则

- Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

  对于可恢复的条件使用被检查的异常，对于程序错误使用运行时异常

Java编译器会对"被检查的异常"进行检查，而对"运行时异常"不会检查。

- 也就是说，对于被检查的异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。（通过程序处理恢复运行）
- 而对于运行时异常，倘若既“没有通过throws声明抛出它”，也“没有用try-catch语句捕获它”，还是会编译通过。（通过调试避免发生）

9大原则

- Avoid unnecessary use of checked exceptions

避免不必要地使用被检查的异常

过分使用被检查异常会使API用起来非常不方便。

- 如果一个方法抛出一个或多个被检查的异常，那么调用该方法的代码则必须在一个或多个catch语句块中处理这些异常，或者必须通过throws声明抛出这些异常。 无论是通过catch处理，还是通过throws声明抛出，都给程序员添加了不可忽略的负担。

- 适用于"被检查的异常"必须同时满足两个条件：第一，即使正确使用API并不能阻止异常条件的发生。第二，一旦产生了异常，使用API的程序员可以采取有用的动作对程序进行处理。

9大原则

- Favor the use of standard exceptions

尽量使用标准的异常

重用现有的异常有几个好处：
第一，它使得你的API更加易于学习和使用，因为它与程序员原来已经熟悉的习惯用法是一致的。

第二，对于用到这些API的程序而言，它们的可读性更好，因为它们不会充斥着程序员不熟悉的异常。

第三，异常类越少，意味着内存占用越小，并且转载这些类的时间开销也越小。

9大原则

- Throw exceptions appropriate to the abstraction

抛出的异常要适合于相应的抽象

如果一个方法抛出的异常与它执行的任务没有明显的关联关系，这种情形会让人不知所措。
为了避免这个问题，高层实现应该捕获低层的异常，同时抛出一个可以按照高层抽象进行介绍的异常。这种做法被称为"异常转译(exception translation)"。

```
public E get(int index) {
    try {
        return listIterator(index).next();
    } catch (NoSuchElementException exc) {
        throw new IndexOutOfBoundsException("Index: "+index);
    }
}
```

将NoSuchElementException转译成了IndexOutOfBoundsException异常

9大原则

- Document all exceptions thrown by each method

每个方法抛出的异常都要有文档

要单独的声明被检查的异常，并且利用Javadoc的@throws标记，准确地记录下每个异常被抛出的条件。

如果一个类中的许多方法处于同样的原因而抛出同一个异常，那么在该类的文档注释中对这个异常做文档，而不是为每个方法单独做文档，这是可以接受的。

9大原则

- Include failure-capture information in detail messages

在细节消息中包含失败——捕获信息

简而言之，当我们自定义异常或者抛出异常时，应该包含失败相关的信息。

当一个程序由于一个未被捕获的异常而失败的时候，系统会自动打印出该异常的栈轨迹。在栈轨迹中包含该异常的字符串表示。典型情况下它包含该异常类的类名，以及紧随其后的细节消息。

9大原则

- Strive for failure atomicity

努力使失败保持原子性

当一个对象抛出一个异常之后，我们总期望这个对象仍然保持在一种定义良好的可用状态之中。

- 设计一个非可变对象。
- 对于在可变对象上执行操作的方法，获得"失败原子性"的最常见方法是，在执行操作之前检查参数的有效性。如下(Stack.java中的pop方法)：

```
public Object pop() {
    if (size==0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;
    return result;
}
```

9大原则

- Strive for failure atomicity

努力使失败保持原子性

当一个对象抛出一个异常之后，我们总期望这个对象仍然保持在一种定义良好的可用状态之中。

- 与上一种方法类似，可以对计算处理过程调整顺序，使得任何可能会失败的计算部分都发生在对象状态被修改之前。
- 编写一段恢复代码，由它来解释操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。
- 在对象的一份临时拷贝上执行操作，当操作完成之后再把临时拷贝中的结果复制给原来的对象。

9大原则

- Don’t ignore exceptions

不要忽略异常

```
try {
    …
} catch (SomeException e) {
}
```

空的catch块会使异常达不到应有的目的，异常的目的是强迫你处理不正常的条件。忽略一个异常，就如同忽略一个火警信号一样。

Keep in mind...

- 勿以恶小而为之，勿以善小而不为

参考书籍

- Bertrand Meyer, ***Object-Oriented Software Construction***, Second Edition, Prentice Hall, 1997.（ Chapter 12 ）

作业

- 解释checked exception, unchecked exception和error三者的定义以及使用的区别。
- 从DbC的角度看，Java方法声明中的throws子句反映了类（supplier ）和其使用者（client）之间的怎样的权利/义务关系？Java子类若重定义父类中的方法，其throws的异常有何限制？用DbC的Contract继承原则解释这个限制。

提交作业到教学立方（3月24号24点截止）