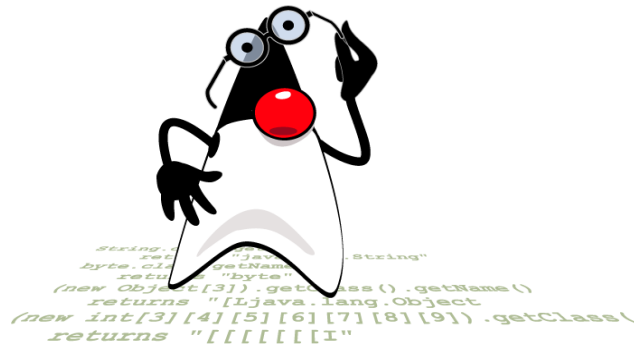


Chap 7 泛型和异常处理



7.1 泛型(Generics)

- 泛型的概念

在**Java 1.5**版本后引入泛型

软件中的错误：编译时刻错误，运行时刻错误

泛型：将类型（类和接口）作为参数

泛型的作用：使用泛型，可以将许多难以检测的运行时刻错误转变为在编译时刻就能检测出来，从而增加了代码的稳定性，另外，泛型的引入可以消除代码中的强制类型转换。

- 泛型的定义和使用

Box类（非泛型版本）

```
public class Box{  
  
    private Object object;  
  
    public void set(Object object){this.object = object;}  
    public Object get(){ return object; }  
  
}
```

```
Box b = new Box();  
b.set(new Integer(2));  
Integer x = (Integer)b.get(); // 运行正常
```

```
Box b = new Box();  
b.set(new String("hello world"));  
Integer x = (Integer)b.get(); //编译器无法检测错误，运行时出错！
```

```
Integer x = (Integer)b.get();  
注意： 需要加入强制类型转换
```

- 泛型的定义和使用

Box类（泛型版本）,引入类型变量**T**

```
public class Box <T>{  
  
    private T t;  
  
    public void set(T t){this.t = t;}  
    public T get(){ return t; }  
  
}
```

使用泛型， 需要使用具体类型替代 T

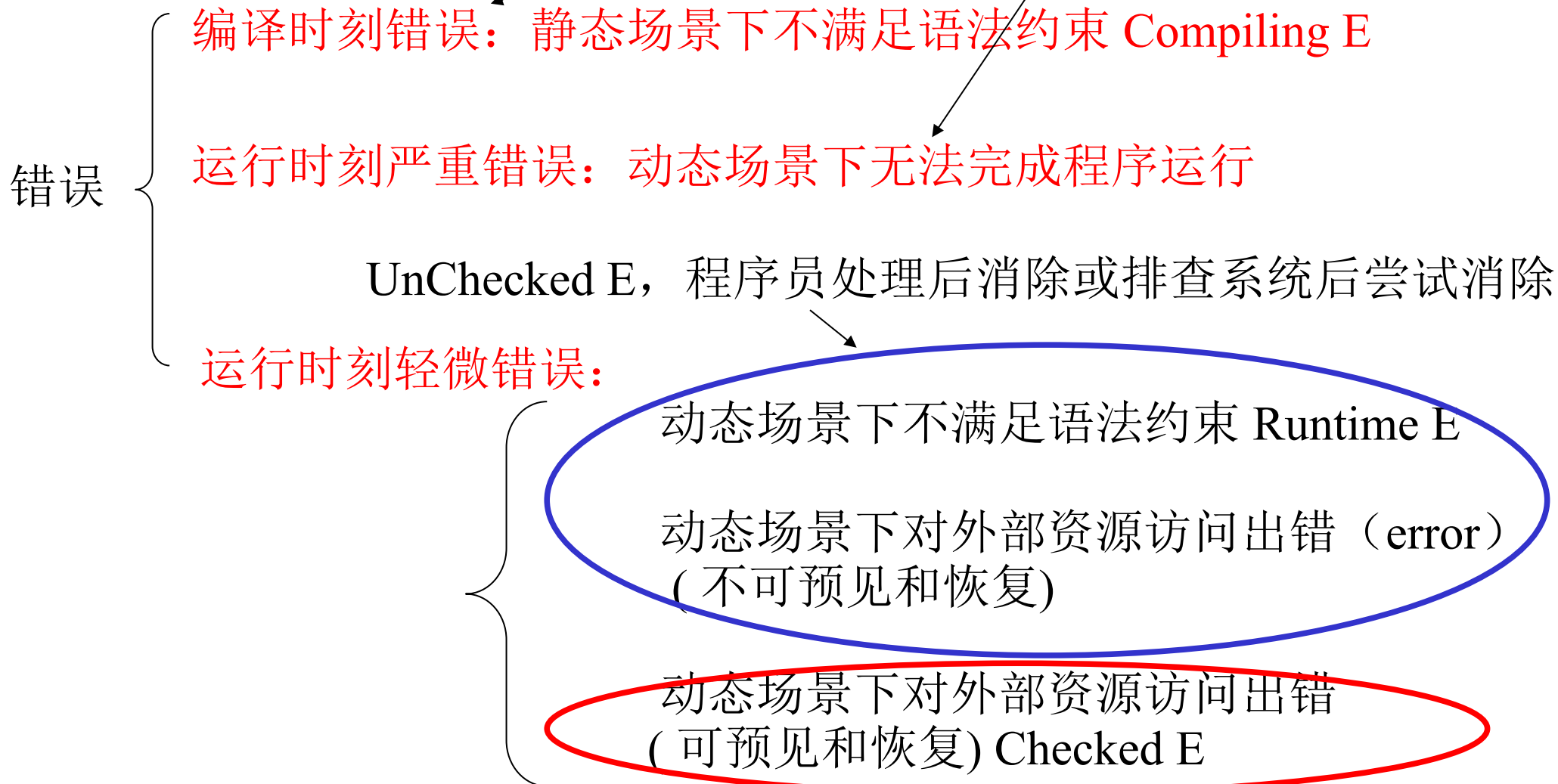
```
Box<Integer> b = new Box<Integer>();  
b.set(new Integer(2));  
Integer x = b.get(); // 运行正常
```

```
Box<Integer> b = new Box<Integer>();  
b.set(new String("hello world")); //编译器检测出错误!  
Integer x = b.get();
```

```
Integer x = (Integer)b.get();  
注意： 不再需要加入强制类型转换
```

7.2 异常(exception)的定义

异常，也称例外，是指在程序执行过程中发生的破坏程序正常指令执行流的事件。



7.2 异常处理的语法结构和处理过程

异常处理的语法结构:

Try 语句体

Catch 语句体

Finally 语句体

举例：来源 Java 白皮书

```
try{  
    startFaucet();  
    waterLawn();  
}
```

← try 语句体

```
catch(exception e)  
{  
    ....  
}  
catch(exception e){....  
    ....  
}
```

← catch 语句体

```
finally  
{  
    stopFaucet();  
}
```

← finally 语句体

- **try** 语句体

构造一个异常处理器的第一步就是将可能抛出异常的语句包含在**try**语句体中。

```
try {  
    Java statements  
}
```

```
try{  
    startFaucet();  
    waterLawn();  
}
```

- **catch** 语句体

在**try**语句体后紧跟一个或多个**catch**语句体构成对异常的处理

```
try {  
    ...  
} catch ( ExceptionType name ) {  
    ...  
} catch ( ExceptionType name ) {  
    ...  
} .
```

* 异常名称必须与需要俘获的异常一致。

* 当异常处理器被调用时，**catch**语句体中的语句被执行。运行环境将调用在调用栈中第一个其异常处理类型匹配异常类型的异常处理器

```
try{  
    startFaucet();  
    waterLawn();  
}  
catch(StartFauceException e)  
{  
    ....  
}  
  
catch(WaterLawnException e)  
{  
    ....  
}
```

- **finally**语句体

Final语句体提供了在程序流程离开异常处理器转向其他程序段前清除方法状态的处理机制，清除代码应该写在**finally**语句体中。例如，回收网络连接等不需要再使用的物理资源。

```
try{  
    startFaucet();  
    waterLawn();  
}  
catch(exception e)  
{  
    ....  
}  
catch(exception e){....  
    ....  
}  
finally  
{  
    stopFaucet();  
}
```


异常处理的动态过程:

异常对象(exception objects)

抛出异常(throwing an exception)

异常处理器(exception handler)

俘获异常(catch the exception)

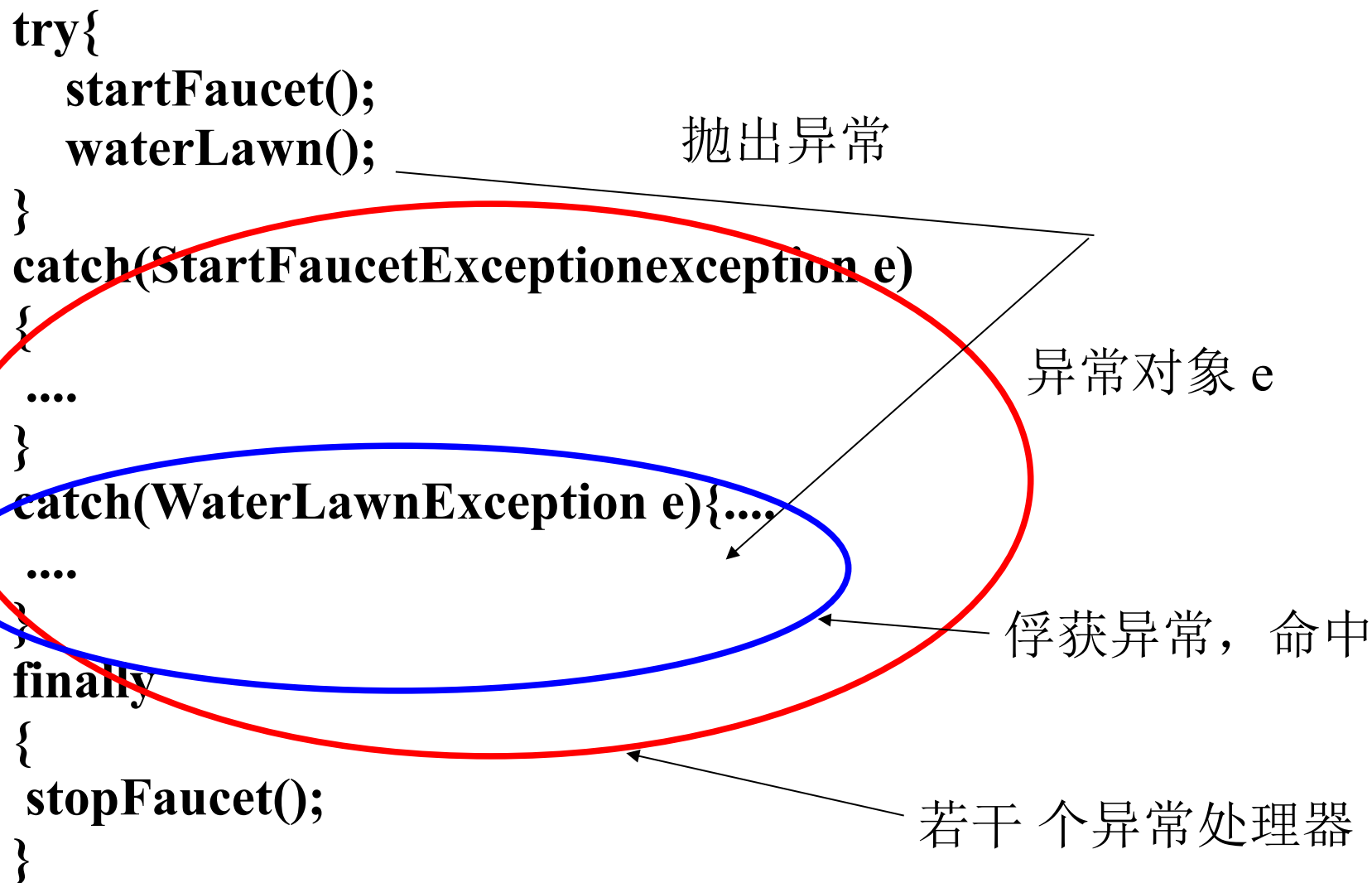
```
try{
    startFaucet();
    waterLawn();
}
catch(StartFaucetException e)
{
    ....
}
catch(WaterLawnException e){....
    ....
}
finally
{
    stopFaucet();
}
```

抛出异常

异常对象 e

俘获异常，命中

若干个异常处理器



异常的分类:

非运行时刻异常(Nonruntime exceptions): 发生在Java运行环境之外的代码中的异常,也称为已检查异常(**checked exceptions**)。例如: 打开文件时, 不存在文件名对应的文件, 抛出异常 **FileNotFoundException**,

运行时刻错误(error): 发生在Java运行环境之外的异常。通常由系统的故障引起, 也称为未检查异常(**unchecked exceptions**), 例如打开文件时, 文件存在, 但文件系统无法工作导致无法读文件, 抛出异常 **IOException** 。

运行时刻异常(Runtime exception): 发生在Java运行环境内部的异常。通常由于违反语法规则引起, 也称为未检查异常(**unchecked exceptions**), 例如**NullPointerException**. 当一个方法试图通过一个 **null**引用去访问一个对象的成员时会抛出这个异常。

注意! 编译器要求确保非运行时刻异常能被俘获或是指定, 否则编译无法通过, 以确保代码在出现非运行时刻异常时能得到处理并继续运行。对未检查异常, 运行时刻错误由 **Error**及子类表示, 运行时刻异常由 **RuntimeException**及子类表示。出现未检查异常异常时, 程序员应该根据指示, 检查程序或系统来尝试解决异常。

- 俘获和指明异常

- 俘获(Catch)

方法可以通过提供与异常匹配的异常处理器以俘获某个异常。

- 指明(Specify)

如果一个方法不俘获异常，那么它必须指定它可以抛出这个异常。通过在方法的参数表后加上**throws**语句来完成这一操作。

```
void xMethod( .... )  
{  
    yMethod(...); //y方法会抛出非运行时刻Z型异常对象e  
}
```

(1)俘获异常e

```
void xMethod( .... )  
{  
    try{  
        yMethod(...); //y方法会抛出非运行时刻Z型异常对象e  
    }  
    catch( ZException e){ //x方法提供处理器匹配Z型异常e并处理  
        .....  
    }  
}
```

```
void xMethod( .... )  
{  
    yMethod(...); //y方法会抛出非运行时刻Z型异常对象e  
}
```

(2) 指明异常e

```
void xMethod( .... ) throws ZException ; //x 方法不处理异常e,  
                                           //抛出指定的这一异常e  
{  
                                           //e将沿调用栈传递  
  
    yMethod(...); //y方法会抛出非运行时刻Z型异常对象e  
  
}
```

throws 语句可以抛出多个异常，异常列表之间用，分隔开。

编写抛出异常的代码

所有的**Java**代码的都可以抛出异常，包括：你自己写的代码，别人写的代码（包含系统给出的代码）以及**Java**运行环境

- **Throw** 语句

所有的**Java** 方法都可以使用**throw**语句抛出一个异常。**throw**语句的参数是一个**throwable**对象。在**Java**系统中，**throwable**对象是**java.lang.Throwable**类的任意一个子类的实例。

throw someThrowableObject;

举例:

```
public Object pop() throws EmptyStackException {  
    Object obj;  
  
    if (size == 0)  
        throw new EmptyStackException();  
  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```


7.3 异常处理为软件设计带来的贡献(优点)

优点1: 将错误处理代码和正常代码分离开来
举例:

ReadFile {

打开文件;
确定文件大小;
分配足够内存;
将文件读入内存;
关闭文件;

}

潜在的错误:

- * 文件打不开?
- * 无法确定文件长度?
- * 无法分配到足够大小的内存空间?
- * 读文件失败?
- * 文件无法关闭?

```
errorCodeType readFile {  
    initialize errorCode = 0;  
    open the file;  
    if (theFileIsOpen) {  
        determine the length of the file;  
        if (gotTheFileLength) {  
            allocate that much memory;  
            if (gotEnoughMemory) {  
                read the file into memory;  
                if (readFailed) {  
                    errorCode = -1;  
                }  
            } else {  
                errorCode = -2;  
            }  
        }  
        close the file;  
        if (theFileDintClose &&  
            errorCode == 0) {  
            errorCode = -4;  
        } else {  
            errorCode = -6;  
        }  
        } else {  
            errorCode = -5;  
        }  
        return errorCode; }
```

```
readFile {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) {  
        doSomething;  
    } catch (sizeDeterminationFailed) {  
        doSomething;  
    } catch (memoryAllocationFailed) {  
        doSomething;  
    } catch(readFailed){
```

```
        doSomething;  
    } catch(fileCloseFailed){  
        doSomething;  
    }  
}
```

*** 7 行代码-> 19行代码**
*** 将出错处理与程序正常逻辑流程清楚地分离开来**
*** 错误的检测，报告和处理仍需由程序员完成**



- * 7 行代码 -> 29 行代码
- * 代码的正常逻辑被破坏并且很难进行维护

优点2: 沿调用堆栈向上传递错误

举例: (嵌套的方法)

```
method1 {  
    call method2;  
}  
method2 {  
    call method3;  
}  
method3 {  
    call readFile;  
}
```

假设method1是唯一一个对readFile可能出错感兴趣的方法。

```
method1 {  
    errorCodeType error;  
    error = call method2;  
    if (error)  
        doErrorProcessing;  
    else  
        proceed;  
}  
errorCodeType method2 {  
    errorCodeType error;  
    error = call method3;  
    if (error)
```

```
        return error;  
        else  
        proceed;  
    }  
    errorCodeType method3 {  
        errorCodeType error;  
        error = call readFile;  
        if (error)  
            return error;  
        else  
            proceed;  
    }  
}
```



传统的错误通知方式要求**method2**和**method3**参与传递由**readFile**产生的错误代码.


```
method1 {  
    try {  
        call method2;  
    } catch (exception) {  
        doErrorProcessing;  
    }  
}  
method2 throws exception {  
    call method3;  
}  
method3 throws exception {  
    call readFile;  
}
```

* 运行环境反向搜寻调用栈以找到对错误感兴趣并进行处理的方法

* **Java**中的方法可以避开任何发生在其中的异常，仅仅由对异常感兴趣的方法才会考虑异常的检测和处理

* 方法通过**throws**语句通知其调用者该方法抛出的异常，由调用者决定如何处理异常

*使用异常后代码变得紧凑并且易于理解

优点 3: 分组区分错误类型

对例外进行分（组）类符合类体系构成的思想
举例：

java.io.IOException -> FileNotFoundException

方法可以通过在**catch**语句中指定任何异常的父类型来俘获到一个异常

catch (FileNotFoundException e){ ... } // 处理类型很明确

catch (IOException e){ ... }

catch (Exception e){ ... } // 处理类型过于宽泛

注意! 通常异常处理器的异常类型应尽可能明确。

7.4 使用异常处理注意的问题

1. 不要过度使用异常。

区分不可预期的异常和可预期的正常处理逻辑错误，以提高处理效率。

2. 不要使用太大的try语句体

将大的try语句体分解成若干小的try语句体，以使得代码简洁明确。

3. 异常处理时应尽可能原子化。

异常处理器应该精确匹配处理异常对象。