

Exceptions

Lecture 5

Daqing Hou, Winter 2007

Today:

- Exceptions
- Specifications
- Exceptions in Java
- Defensive programming
- Design Issues

An 'unexceptional' example

```
Class Array {...
public static int search(int[] a, int x) throws NullPointerException,
    NotFoundException
// REQUIRES: a is sorted in ascending order
// EFFECTS: if a is null, throws NullPointerException;
// else if x is in a, returns i such that a[i]==x;
// else throws NotFoundException.
{for (int i=0; i<a.length; ++i) {
    if (a[i]==x) return i;
    else if (a[i]>x) throw new NotFoundException();
}
... } // END of Array

try{...;
    try { x=Array.search(v,7);}
    catch(NullPointerException npe){
        throw new NotFoundException();}
} catch (NotFoundException nfe){...}
```

Motivating Exercise 1

Consider writing a specification for the procedure

```
public static int sum(int[] a)
```

that computes as its return value the sum of elements in the integer array `a`. The spec. might require a non-empty array, return 0 when the array is empty, or throw an exception.

Compare these alternatives.

Motivating Exercise 2

Consider

```
static void combine(int []a, int []b)
```

that multiplies each element of a by the sum of the elements of b; for example, if a=[1,2,3] and b=[5,6], then on return a=[11,22,33]. What should this procedure do if a or b is null or empty?

Exceptions: partial proc

- Partial procedure
 - e.g., $\text{gcd}(n,d)$ is not defined when n or d negative.
- Caller of a partial procedure must ensure REQUIRES conditions are satisfied
- Failure to ensure this may cause difficult-to-find errors
 - robust program and graceful degradation
- Total procedures enhance robustness
 - do something about exceptional inputs, or
 - let the caller know them

Exceptions: special value

- One way to indicate exceptional cases is to return special values

```
public static int fact(int n)
    // EFFECTS: if n>0 return n! else returns 0
```

- One problem with this approach is that instead of

```
z= x+Num.fact(y)
```

using code must be written as

```
int r=Num.fact(y);
if (0!=r) z=x+r; else ...
```

- Another problem is range may be used up
Vector v; ... v.get(i);

Specifications

A procedure header lists *all* exceptions it may throw

```
public static int fact(int n) throws  
    NonPositiveException
```

```
// EFFECTS: if n is non-positive, throws  
// NonPositiveException, else returns n!
```

```
public static int search(int[] a, int x) throws  
    NullPointerException, NotFoundException
```

```
// REQUIRES: a is sorted  
// EFFECTS: if a is null, throws NullPointerException;  
// else if x is in a, returns i such that a[i]==x;  
// else throws NotFoundException.
```

Java exceptions

- Java exception type hierarchy
 - Throwable(Error, Exception (RuntimeException(unchecked), checked))
 - checked and unchecked exceptions
- Defining exception types
 - new NewKindOfException("This is the reason");
 - nke.toString(): NewKindOfException: This is the reason
 - put all exceptions in a special package
 - CapitalizingFirstLetterOfWordsInExceptionNameException
- Throwing & handling exceptions

Handling exceptions

- ```
try{x=Num.fact(y);}
catch (NonPositiveException npe){
// npe can be used here
}
```
- ```
try{...;
    try { x=Arrays.search(v,7);}
    catch(NullPointerException npe){
        throw new NotFoundException();}
} catch (NotFoundException nfe){...}
```
- ```
try { x=Arrays.search(v,7);}
catch(Exception e){s.println(e); return;}
finally{System.out.println("am I executed?");}
```

# Exception propagation

# Defensive programming

- Goal: locate an error right at the location where it happens, do not let it propagate
- Check requires conditions and raise `FailureException` when they are violated
- Check an assumption at program points where it must hold and raise `FailureException` when otherwise
- Do not list `FailureException` in a procedure header as this exception indicates a programming error that should not be hidden

# Benefits of exceptions

- Separate normal cases from exceptional cases
- Draw users' attention to exceptional cases so that they cannot be missed
- Handling of special cases are separate from normal cases
- E.g., `aVector.get(int i)` may throw `IndexOutOfBoundsException`

# Design issues

---

- What is an exception, anyway?
- Exceptions are simply a means to allowing for several kinds of behavior and informing the caller about the different cases.
- Classification of one case as normal and the others as exceptional is somewhat arbitrary.

# Design issues

---

- When to use exceptions?
- Use exceptions
  - to remove conditions in requires clauses, if they are cheap to check,
  - to avoid encoding information in ordinary results,
  - and to program defensively.

# Design issues

---

- Checked and unchecked, which one to use?
  - Depends on the context of use.
- Use unchecked exceptions
  - when users will/can usually write code that ensures an exception will not happen because there is a convenient and inexpensive way to avoid it
  - or when the use is local
- Otherwise, use a checked exception