Exceptions, Assertions, Defensive programming and Design by Contract

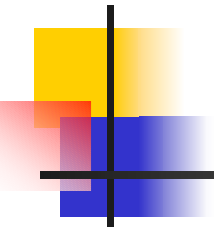
Lecture 3: OOP, autumn 2003

Why Exceptions

- Partial procedures => non-robust programs
- Solution: generalize to total procedures
- Problem: how to notify client of problems?
  - Mechanisms known so far:
    - Use special return values
    - Modify input values
  - Use a special mechanism to inform about exceptional situations

=> Exceptions

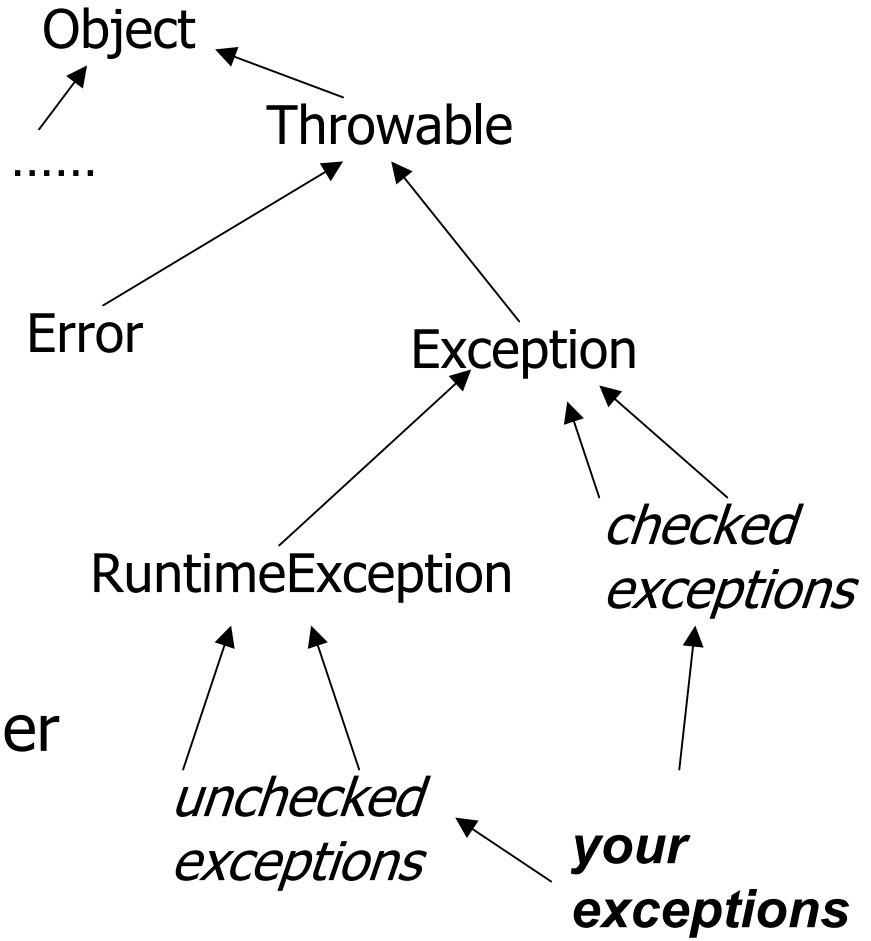
Procedure specifications with Exceptions



- **`throw`** - terminates procedure exceptionally ≠ return
  - Declare thrown exceptions **`throws`** `e1, e2, …`
  - **effects** explains reasons for each exception
  - **requires** includes inputs *not* handled through exceptions - still partial procedure
- **`try/catch`** - get exception

Exceptions in Java: checked vs. unchecked

- Errors: generated by JVM - fatal failure
- Checked:
  - must be in specification
  - must be handled by caller
- Unchecked:
  - may not be in specification
  - may not be handled by caller
- List all exceptions in header



Transfer of control: throwing exceptions

```
if <condition> throw new <ExceptionType>
```

……

```
if (balance < 0) {
  throw new NegativeBalanceException("message")
}
```

……

- Terminates procedure exceptionally
- Exceptions are not "returns"!

Transfer of control: handling exceptions

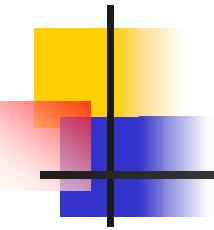
- Explicitly by try/catch (can be nested)

```
try {
   float sum = someAccount.getMoney(s);
} catch (NegativeBalanceException e) {
   // inform user
   throw new UnableToBuyException("empty account");
}
```

- Implicitly through automatic propagation
  - Exception listed in caller's header
  - Exception is unchecked

Steps of try…catch…finally

- Every try block must have at least one catch block
- If an exception is raised during a try block:
  - The rest of the code in the try block is skipped over.
  - If there is a catch block of the correct, or derived, type **in this stack frame** it is entered.
  - If there is no such block, the JVM moves up one stack frame.
- Uncaught exceptions are handled by the JVM



Programming with exceptions

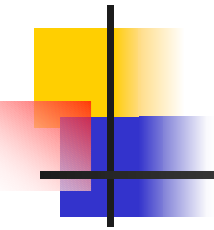
- Reflecting
  - Automatic propagation
  - Catch and throw new exception - relate exception to current abstraction
- Masking
  - Handle exception
  - Continue normally

Procedure design with exceptions

- Exceptions ≠ errors
- When to use
  - Replace **require** conditions with exceptions
    - but mind efficiency
    - may avoid when local code
  - Use exceptions in general procedures

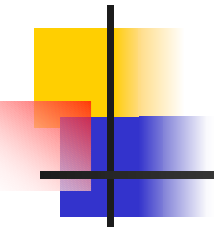
Checked vs. unchecked exceptions

- Checked
  - + compiler forces handling exceptions
  - - must handle even if unnecessary
- Unchecked
  - + simpler - may not handle, but avoid
  - + faster
  - - Error-prone - can be accidentally captured
- Rules:
  - Unchecked <= simple to avoid, local use
  - Checked <= otherwise



Defensive programming (1)

- There can always be errors (e.g. scarce resources)
- Procedures defend themselves from errors
- Use special unchecked exception FailureException
- Not listed in specification!
- Raise when some assumption doesn't hold, i.e.
  - violated **requires**
  - breach of contract



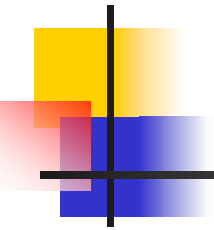
Defensive programming (2)

+ Increases software **robustness** - handling abnormal conditions

- Increases complexity

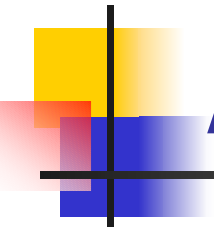
- Checks too late

- What about correctness?
  - adherence to a specification
- Not suitable to maintain correctness



Design by contract (DBC)

- DBC - check early and crash
- Shift responsibility to client
- “exceptional” => “incorrect”
- Use pairs of
  {Precondition} Action {Postcondition}
- Assertions provide the basis
  - Manual use
- Other languages/tools provide direct support for DBC
  - Eiffel
  - iContract, jContract for Java



Assertions

- Boolean expressions
- Used to check:
  - Pre-conditions
    - reflect **requires** clause
    - Test client
  - Post-conditions
    - reflect **effects** clause
    - test procedure
  - Invariants
- Include specification in the software

Invariants

- **Invariant** – “A rule, such as the ordering of an ordered list or heap, that applies throughout the life of a data structure or procedure. Each change to the data structure must maintain the correctness of the invariant”. (http://foldoc.doc.ic.ac.uk/foldoc/index.html)

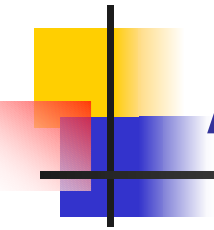
- **Class invariant** – if the “data structure” above is a class…

Invariants example

```
class CharStack {
  private char[] cArr; // internal rep
  private int i = 0;
  void push(char c) {
    cArr[i] = c;
    i++;
  }
}
```

- The invariant in this example is: “ **i** should always be equal to the size of the stack (i.e. point at one above at the top of the stack)”

Assertions in Java



- Added in JDK 1.4
- General syntax:

```
assert expression1 : expression2
```

- Added in JDK 1.4
- Examples

```
assert value >= 0;
assert someInvariantTrue();
assert value >= 0 :
  "Value must be > 0: value = " + value;
```

- `> javac -source 1.4 *.java`
- `> java -ea MyClass`

Handling assertions in Java

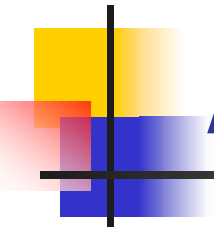
- Evaluate *expression*$_1$
  - If true
    - No further action
  - If false
    - And if *expression*$_2$ exists
      - Evaluate *expression*$_2$ and throw AssertionError(*expression*$_2$)
  - Else
    - Use the default AssertionError constructor

Care with assertions

- Side effects in assertions

```
void push(char c) {
    cArr[i] = c;
    assert (i++ == topElement());
}
```

- Change of flow in assertions
- Performance vs. correctness
  - Open issue



Assertions vs. Exceptions

In Class Sensor:

```
public void setSampleRate(int rate)
throws SensorException {
  if(rate < MIN_HERTZ  ||  MAX_HERTZ < rate)
    throw new SensorException ("Illegal rate: " + rate);
    this.rate = rate;
}

public void setSampleRate(int rate) {
  assert MIN_HERTZ <= rate  &&  rate <= MAX_HERTZ :
    "Illegal rate: " + rate;
  this.rate = rate;
}
```

Summary

- Use exceptions for program robustness
- Use assertions for program correctness
- Both are complementary
- Both have cost
- Assertions are not DBC - all depends on programmer
- Assertions can be disabled