Exceptions and error handling

Java exception mechanism

• when an error or exceptional condition occurs, you throw an exception which is caught by an exception handler
  
  ```java
  try {
    // statements that may throw an exception..
    if (someCondition)
      throw new ExceptionName();
  } catch (ExceptionName e) {
    // executed for ExceptionName
  } finally {
    // executed whether exception or not
  }
  ```

Java exception mech. (cont.)

• when searching for a catch block to handle an exception thrown, Java goes upward through the dynamic activation stack of method calls until it finds a matching catch block

• should use exceptions only for "exceptional" conditions, not for ordinary control flow

• may enable recovery from a failure

• can ensure release of critical resources

• at least, enables graceful shutdown

On exception handling

• a library shouldn't produce diagnostic output at an end user, or independently terminate a program

• instead, throw an exception and let a caller decide

• not every program needs to handle exceptions safely:
  – a simple application may produce a suitable diagnostic, let the system release all acquired resources, and let the user re-run the program with a more suitable input
Benefits of exception handling

- separates error handling from normal processing
- provides a systematic way to report and handle exceptional situations (failures or errors)
- exceptions are necessary for making reusable libraries and components work:
  - the library component may detect an error, e.g., violation of a precondition, but cannot know how to handle it

Handling exceptions

Generally, there are many ways to handle an exception:

1. let Java handle it (but don't suppress it)
2. fix the problem that led to the exception and resume the program; but often, erroneous conditions are difficult or impossible to fix
3. log the problem and resume the program
4. log the problem and rethrow the exception
5. chain a low-level exception within a new higher-level exception (wrap inside as a parameter) and throw that
6. print/display/log out an error message and let the program terminate

Handling exceptions (cont.)

- no one-for-all solution, depends strongly on the specific requirements of the application
- it may well be OK to simply let the program terminate itself, and let the user try with better input

Java exception class hierarchy

- Checked exceptions—exceptions that must be caught or declared in a program
- Unchecked exceptions—serious errors that a typical program should not have to handle
Checked vs. unchecked exceptions

- **checked exceptions** are either caught by the method in which they occur, or you must declare that the method throws the exception.
- **checked exceptions** force the programmer to deal with exceptional conditions; error situations become explicit and in principle the program becomes more reliable.
- **unchecked exceptions** extend `RuntimeException` and, if uncaught, are handled by Java.
- use **checked exceptions** for recoverable conditions and `RuntimeExceptions` for programming errors.
- other object-oriented languages (C++, C#) don’t have checked exceptions (but Clu did).

Guidelines

- avoid *unnecessary use* of exceptions.
- exception handling is not meant to replace a simple test; for example:
  - reaching the end of a list should not throw an exception.
  - the API offers a query method (hasNext) for that.
  - but then going past the end should throw a `NoSuchElementException` is a run-time exception that indicates a programming error.
- don’t wrap every statement in a separate try block.
  - “throw early, catch late” at a higher-level system.

Guidelines (cont.)

- don’t suppress or ignore exceptions.
  - try {
      
      ... some code that throws SomeException
      
    } catch (SomeException ex) {
      
      // OK: don’t care
    }
  
- if a **checked exception** does not make sense to you, don’t hesitate to convert it into an **unchecked exception** and throw it again.
  - try {
      
      ... // actually cannot throw exceptions
    } catch (SomeException e) {
      
      assert false; // so shouldn’t get here
    }

Guidelines (cont.)

- in some situations, the programmer may “know” that the code cannot throw any exceptions whatsoever.
- of course, then it should be well documented and checked:
  - try {
      
      ... // actually cannot throw exceptions
    } catch (SomeException e) {
      
      assert false; // so shouldn’t get here
    }

Guidelines (cont.)

- chaining constructors available since version 1.4.
Guidelines (cont.)

- but don’t catch unnecessarily high-level exceptions
  
  ```java
  try {
    // ... some code that throws exceptions
  } catch (Exception ex) {
  }
  ```

- unchecked exceptions inherit `RuntimeException` class which inherits `Exception`

- the code above ignores all unchecked exceptions, as well

---

Exception translation

- for example, a `get()` method may call more primitive operations with their own exceptions; the method can perform exception translation:
  ```java
  public E get (int index) {
    ListIterator it = listIterator (index);
    try {
      return it.next () ;
    } catch (NoSuchElementException e) {
      throw new IndexOutOfBoundsException
        ("Index: " + index);
    }
  }
  ```

Guidelines (cont.)

Favor the use of standard exceptions, for example:

- `IllegalArgumentException`: parameter value is inappropriate
- `IllegalStateException`: state is inappropriate for method call (before `remove` must call `next`)
- `NullPointerException`: null value where prohibited
- `IndexOutOfBoundsException`: invalid index
- `ConcurrentModificationException`: modification of object has been detected where prohibited
- `UnsupportedOperationException`: object does not support method

Guidelines (cont.)

- document all exceptions thrown by each method using the Javadoc `@throws` tag
- for debugging, the string representation of an exception should contain the values of all objects that “contributed” to the exception
- higher layers can catch lower-level exceptions, and in their place, throw exceptions that are explainable in terms of the higher-level abstraction
**Exception chaining**

- In *exception chaining* a lower-level exception is stored by the higher-level exception for debugging purposes:

```java
try {
    lowLevelOperation ();
} catch (LowLevelException e) {
    throw new HighLevelException (e);
        // has chaining constructor
}
```

- Additionally, the class `Throwable` provides methods `getCause()` and `initCause(Throwable cause)` to get and set the original cause of the error situation.

**Failure atomicity**

- Ideally, attempt to achieve *failure atomicity*:
  - A failed method should leave the system in the state it was in prior to the call.
  - In principal, recovery is not possible if the system is left in an undetermined, possibly faulty state.
  - Failure atomicity is especially important for checked exceptions, from which the caller is expected to recover.

**Failure atomicity (cont.)**

```java
public E pop () {  // a bit trivial but illustrative
    if (size () == 0)
        throw new NoSuchElementException ();
    // now can safely modify the contents:
    E result = elements [--size];
    elements [size] = null;  // eliminate old reference
    return result;
}
```

**Using finally blocks**

- Finally clauses release resources no matter what.
- A finally clause can be used without a catch clause:

```java
Graphics g = image.getGraphics ();
try {
    code that might throw exceptions
} finally {
    g.dispose ();
}
```
Using finally blocks (cont.)

• you can decouple try/catch and try/finally to make your code more readable:

```java
InputStream in = ...;
try {
    try { // ensure that stream is closed
code that might throw exceptions
} finally {
    in.close();
}
} catch (IOException e) { // reports errors
    show error dialog
}
• this resembles the use C++ destructors
```

Problems with finally blocks

• if a finally block itself throws an exception, the original exception becomes permanently lost
• as a rule, all clean-up code such as finally should be programmed *not to throw* any exceptions
  – very hard to enforce in practice, and
  – unfortunately, not all library methods necessarily respect the right spirit of exception handling

Problems finally blocks (cont.)

• if a try block or a catch clause contains a return statement, the finally block is (as always) executed before the return
• however, if the finally block itself contains a return, it then *masks off* the original return

```java
public static f (int n) {
    try { ...
        return n * n;
    } ...
} finally {
    if (n == 2) return 0; // strange and unexpected
}
```