Java Exceptions, Cloning, Serialization

Nelson Padua-Perez
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Overview

- Review
  - Errors
  - Exceptions
- Java support
  - Representing exceptions
  - Generating & handling exceptions
  - Designing & using exceptions
Types of Program Errors

1. Syntax (compiler) errors
   - Errors in code construction (grammar, types)
   - Detected during compilation
2. Run-time errors
   - Operations illegal / impossible to execute
   - Detected during program execution
     - Treated as exceptions in Java
3. Logic errors
   - Operations leading to incorrect program state
   - May (or may not) lead to run-time errors
   - Detect by debugging code

Exception Handling

- Performing action in response to exception
- Example actions
  - Ignore exception
  - Print error message
  - Request new data
  - Retry action
- Approaches
  1. Exit program
  2. Exit method returning error code
  3. Throw exception
Problem

- May not be able to handle error locally
  - Not enough information in method / class
  - Need more information to decide action

- Handle exception in calling function(s) instead
  - Decide at application level (instead of library)
  - Examples
    - Incorrect data format ⇒ ask user to reenter data
    - Unable to open file ⇒ ask user for new filename
    - Insufficient disk space ⇒ ask user to delete files
  - Will need to propagate exception to caller(s)

Exception Handling – Exit Program

- Approach
  - Exit program with error message / error code

- Example
  ```java
  if (error) {
    System.err.println("Error found"); // message
    System.exit(1); // error code
  }
  ```

- Problem
  - Drastic solution
  - Event must be handled by user invoking program
  - Program may be able to deal with some exceptions
Exception Handling – Error Code

- **Approach**
  - Exit function with return value ⇒ error code

- **Example**
  
  ```
  A( ) { if (error) return (-1); }
  B( ) { if ((retval = A( )) == -1) return (-1); }
  ```

- **Problems**
  - Calling function must check & process error code
    - May forget to handle error code
    - May need to return error code to caller
  - Agreement needed on meaning of error code
  - Error handling code mixed with normal code

---

Exception Handling – Throw Exception

- **Approach**
  - Throw exception (caught in parent’s catch block)

- **Example**
  
  ```
  A( ) {
    if (error) throw new ExceptionType();
  }
  B( ) {
    try {
      A( );
    } catch (ExceptionType e) { ...action... }
  }
  ```
Exception Handling – Throw Exception

- **Advantages**
  - Compiler ensures exceptions are caught eventually
  - No need to explicitly propagate exception to caller
    - **Backtrack** to caller(s) automatically
  - Class hierarchy defines meaning of exceptions
    - No need for separate definition of error codes
  - Exception handling code separate & clearly marked

Representing Exceptions

- Exceptions represented as
  - Objects derived from class Throwable

- **Code**
  
  ```java
  public class Throwable( ) extends Object {
    Throwable( ) // No error message
    Throwable( String mesg ) // Error message
    String getMessage() // Return error mesg
    void printStackTrace( ) { ... } // Record methods
    ... // called & location
  }
  ```
Representing Exceptions

Java Exception class hierarchy

Two types of exceptions ⇒ checked & unchecked

Representing Exceptions

Java Exception class hierarchy
Unchecked Exceptions

- Class **Error** & **RunTimeException**
- Serious errors not handled by typical program
- Usually indicate logic errors
- Example
  - NullPointerException, IndexOutOfBoundsException
- Catching unchecked exceptions is **optional**
- Handled by Java Virtual Machine if not caught

Checked Exceptions

- Class **Exception** (except RunTimeException)
- Errors typical program should handle
- Used for operations prone to error
- Example
  - IOException, ClassNotFoundException
- Compiler requires “catch or declare”
  - Catch and handle exception in method, OR
  - Declare method can throw exception, force calling function to catch or declare exception in turn
- Example
  - void A() throws ExceptionType { … }
Generating & Handling Exceptions

- **Java primitives**
  - Try
  - Throw
  - Catch
  - Finally

- **Procedure for using exceptions**
  1. Enclose code generating exceptions in try block
  2. Use throw to actually generate exception
  3. Use catch to specify exception handlers
  4. Use finally to specify actions after exception

**Java Syntax**

```
try {
    throw new eType1();
} // try block encloses throws
    // throw jumps to catch
catch (eType1 e) {
    ...action...
} // catch block 1
    // run if type match
catch (eType2 e) {
    ...action...
} // catch block 2
    // run if type match
finally {
    ...action...
} // final block
    // always executes
```
Java Primitive – Try

- Forms **try** block
- Encloses all statements that may throw exception
- Scope of try block is **dynamic**
- Includes code executed by methods invoked in try block (and their descendents)

Java Primitive – Try

**Example**

```java
try {    // try block encloses all exceptions in A & B
    A( );    // exceptions may be caught internally in A & B
    B( );    // or propagated back to caller’s try block
}

void A( ) throws Exception {    // declares exception
    B( );
}

void B( ) throws Exception {    // declares exception
    throw new Exception( );    // propagate to caller
}
```
Java Primitive – Throw

- Indicates exception occurred
- Normally specifies one operand
  - Object of class Exception
- When an exception is thrown
  1. Control exits the try block
  2. Proceeds to closest matching exception handler after the try block
  3. Execute code in exception handler
  4. Execute code in final block (if present)

Java Primitive – Catch

- Placed after try block
- Specifies code to be executed for exception
  - Code in catch block ⇒ exception handler
- Catch block specifies matching exception type
- Can use multiple catch blocks for single try
  - To process different types of exceptions
  - First matching catch block executed
  - Subclass may subsume catch for subclass
    - If catch block for superclass occurs first
Java Primitive – Catch

**Example**

```java
class eType1 extends Exception { … }
try {
    … throw new eType1( ) …
}
catch (Exception e) { // Catch block 1
    …action… // matches all exceptions
}
catch (eType1 e) { // Catch block 2
    …action… // matches eType1
} // subsumed by block 1
    // will never be executed
```

Java Primitive – Catch

- Can **rethrow** exception
  - Exception propagated to caller(s)

**Example**

```java
catch (ExceptionType e) {
    … // local action for exception
    throw e; // rethrow exception
} // propagate exception to caller
```
Java Primitive – Finally

- Placed after try & all catch blocks
- Forms **finally block**
- Cleanup code
  - Executed by all exception handlers
  - Try restore program state to be consistent, legal
- Always executed
  - Regardless of which catch block executed
  - Even if no catch block executed
  - Executed before transferring control to caller
    - If exception is not caught locally

Designing & Using Exceptions

- Use exceptions only for rare events
  - Not for common cases ⇒ checking end of loop
  - High overhead to perform catch
- Place statements that jointly accomplish task into single try / catch block
- Use existing Java Exceptions if possible
Designing & Using Exceptions

- Avoid simply catching & ignoring exceptions
  - Poor software development style

- Example

  ```java
  try {
    throw new ExceptionType1();
    throw new ExceptionType2();
    throw new ExceptionType3();
  }
  catch (Exception e) { // catches all exceptions
    ... // ignores exception & returns
  }
  ```

Exceptions Summary

- Java primitives
  - Try
    - Forms try block
    - Encloses all statements that may throw exception
  - Throw
    - Actually throw exception
  - Catch
    - Catches exception matching type
    - Code in catch block ⇒ exception handler
  - Finally
    - Forms finally block
    - Always executed, follows try block & catch code
Exceptions Summary

- Programs often contain errors
- Exceptions ⇒ advanced Java language feature
- Java provides detailed support for exceptions
- Learn to use exceptions carefully

Java – Cloning

- Cloning
  - Creating an identical copy
- Cloneable interface
  - Supports clone( ) method
  - Returns copy of object
    - Copies all of its fields
    - Does not clone its fields
    - Makes a shallow copy
Java – Cloning

- **Effect of clone( )**
  - Creates new object
  - X.clone( ) != X
  - Same class
    - X.clone.getClass( ) == X.getClass( )
  - Modification to X no longer affect X.clone( )

Java Clone Comparison

- **Example (X.f = Z)**
  - X
  - X
  - X.f = Z
  - X
  - X
  - Z
  - Z
Java Clone Comparison

Example (X.f = Z)

\[ Y = X \]

\[ Y = X.clone() \]

Java Clone Comparison

Example (X.f = Z)

\[ Y = X; X.f = A \]

\[ Y = X.clone(); X.f = A \]
Java – Serializability

- Serializability
  - Ability to convert a graph of Java objects into a stream of data, then convert it back (deserialize)

- Java.io.Serializable interface
  - Marks class as Serializable
  - Supported by Java core libraries
  - Special handling (if needed) using
    - private void writeObject(java.io.ObjectOutputStream out)
      throws IOException
    - private void readObject(java.io.ObjectInputStream in)
      throws IOException, ClassNotFoundException;
  - Makes a deep copy

Serializability – Uses

- Persistence
  - Using FileOutputStream
  - Store data structure to file for later retrieval

- Copy
  - Using ByteArrayOutputStream
  - Store data structure to byte array (in memory) and use it to create duplicates

- Communication
  - Using stream from a Socket
  - Send data structure to another computer
Serializability – Deep Copy

// serialize object
ByteArrayOutputStream mOut = new ByteArrayOutputStream(  );
ObjectOutputStream serializer = new ObjectOutputStream(mOut);
serializer.writeObject(serializableObject);
serializer.flush(  );

// deserialize object
ByteArrayInputStream mIn = new ByteArrayInputStream(mOut.
toByteArray(  ));
ObjectInputStream deserializer = new ObjectInputStream(mIn);
Object deepCopyOfOriginalObject = deserializer.readObject(  );

Java Serializable Comparison

Example (X.field = Z)

Y = X
Y = X.clone( )
OS.writeObject(x)
Y = readObject(IS)