CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Exceptions

Department of Computer Science
University of Maryland, College Park

Exceptions (Rare Events)

- Rare event outside normal behavior of code
  - Usually, a run-time error
- Examples
  - Division by zero
  - Access past end of array
  - Out of memory
  - Number input in wrong format (float vs. integer)
  - Unable to write output to file
  - Missing input file

Dealing with Exceptions (Rare Events)

- What to do when this kind of event occurs?
  - Ignore the problem
  - Print error message
  - Request data
  - Exit method returning error code caller must check
  - Exit program
- Exiting method returning error code has disadvantages
  - Calling method may forget to check code
  - Agreement on error codes
  - Error handling code mixed with normal code
- Preferred approach: **Exception Handling** (e.g., Java's exception mechanism)

Exception Handling 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 in Java

- Exceptions represented as
  - Objects derived from class Throwable
- Code

```
public class Throwable  {
    Throwable()                          // No error message
    Throwable(String mesg)               // Error message
    String getMessage()                  // Return error mesg
    void printStackTrace( ) { … }        // Record methods
   …                                     // called & location
}
```

Java Exceptions

- Any code that can potentially throw an exception can been closed in a
  - **try { } block**
- Exception handlers are specified using catch
  - **catch(ExceptionType e) { }**
- You can have several catch clauses associated with a try block

Java Exceptions

- When an exception is thrown
    - Control exits the try block
    - Control proceeds to closest matching exception handler after the try block
        - Java exceptions backtrack to caller until matching block is found
    - Execute code in exception handler
    - Execute code in finally block (if present)
- **Example:** Fundamentals.java
- Scope of try is dynamic
    - Includes code executed by methods invoked in try block (and their descendants)

Java Exceptions

- **Throwing exceptions**
  - In previous example the exception was thrown for you
  - You can throw exceptions too
    - throw <Object of class exception>
  - Example:

```
throw new UnsupportedOperationException("You must implement this method.");
```

Java Exceptions

- **Finally block**
  - Code that is executed no matter what
    - Regardless of which catch block
    - Even if no catch block is executed
    - Executed before transferring control to caller
  - Placed after try and all catch blocks
  - Tries to restore program state to be consistent, legal (e.g., closing files)
- **Example:** ReadNegativeValue.java

Propagation

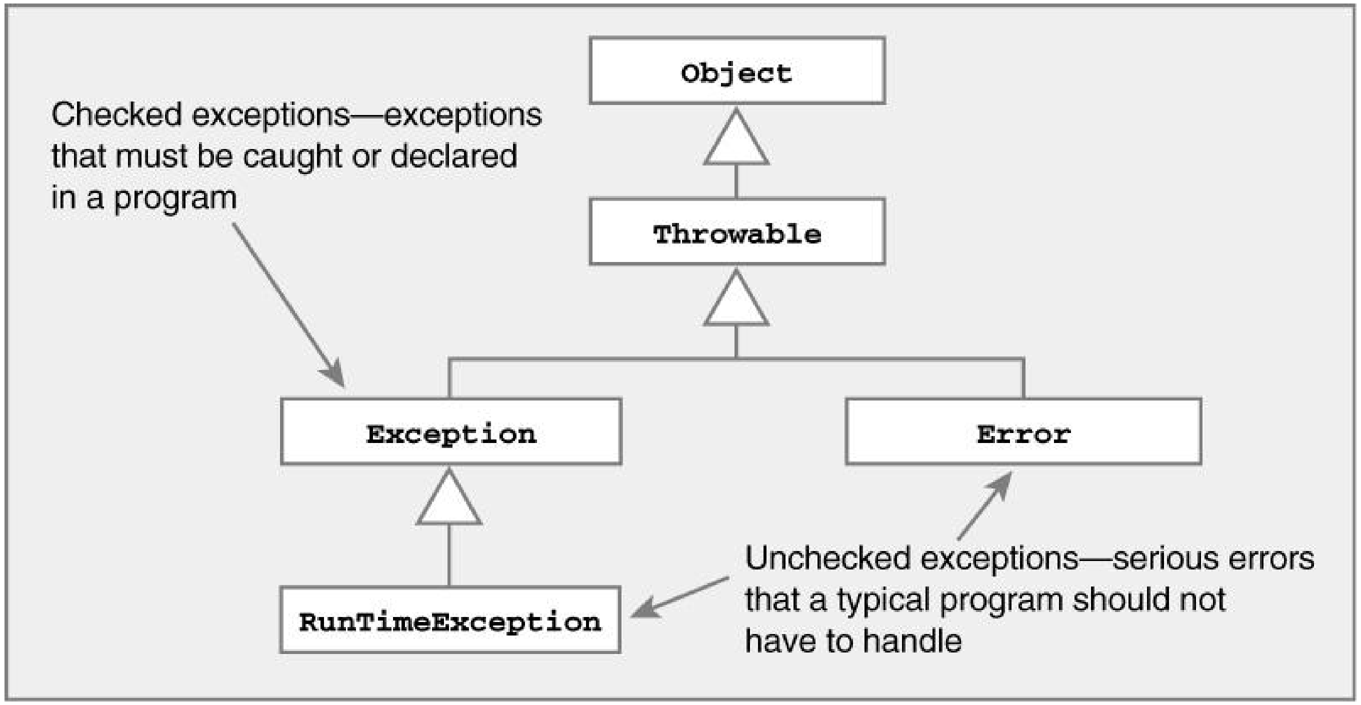
- Control proceeds to closest matching exception handler after the try block
  - Java exceptions backtrack (propagation) to caller until matching block is found
- **Example:** Propagation.java

Several Catch Clauses

- **Example:** SeveralCatchClauses.java

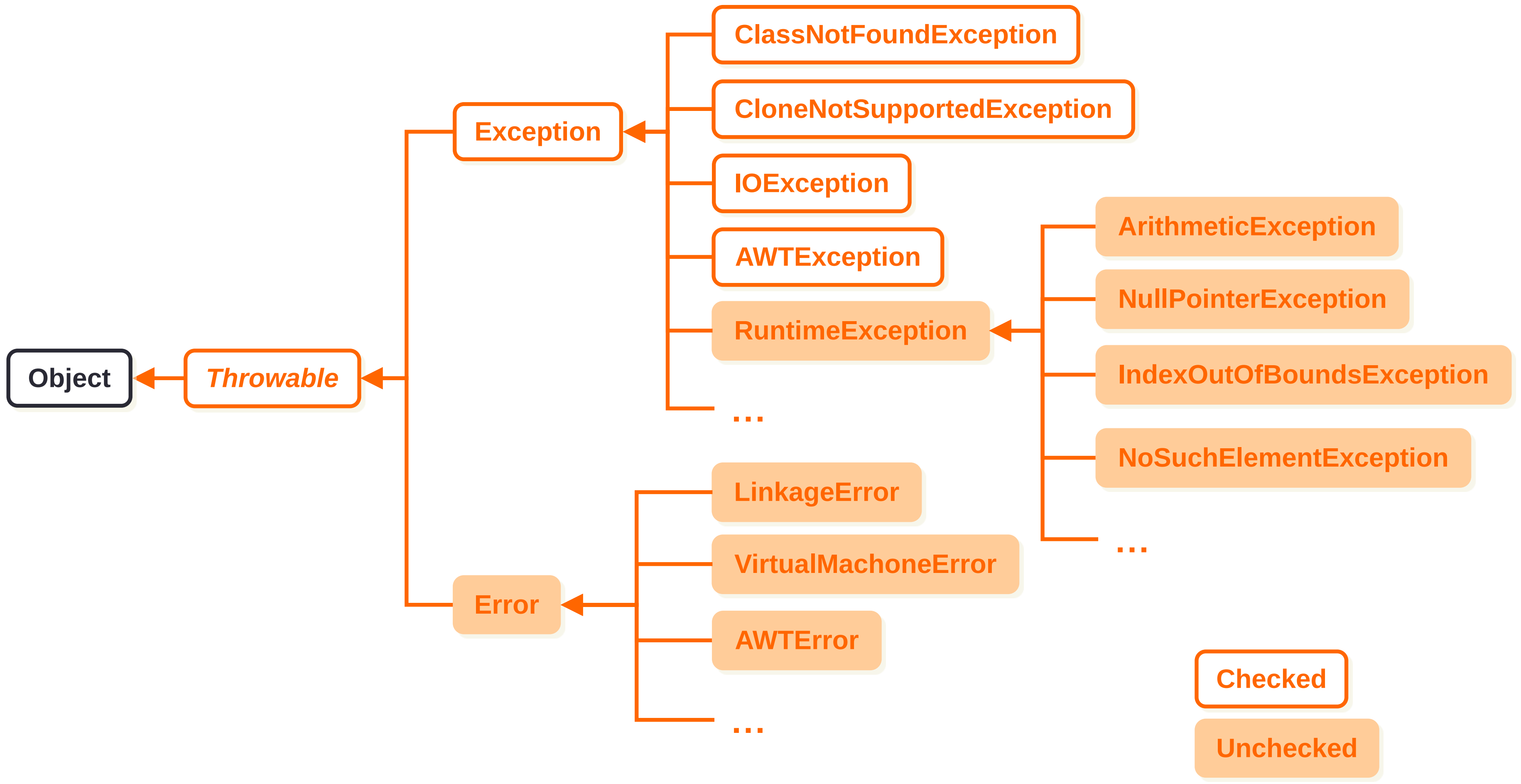
Representing Exceptions

- Java exceptions class hierarchy
  - Two types of exceptions - checked & unchecked
  - **Unchecked** - Serious errors not handled by typical program
  - **Checked** - Errors typical program should handle (e.g., file not found)


Object
Throwable
Exception
Error
RunTimeException
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

Representing Exceptions

- Java Exception class hierarchy



Checked and Uncheck Exceptions

- **Unchecked**
  - Serious errors not handled by typical program
  - They are your fault ☺ (your code is wrong)
  - Usually indicate logic errors
  - Examples → NullPointerException, IndexOutOfBoundsException
  - Catching unchecked exceptions is optional (handled by JVM if not caught)

Checked and Uncheck Exceptions

- **Checked**
  - Errors typical program should handle. Describes problem that may occur at times, regardless how careful you are
  - Used for operations prone to error
  - Examples → IOException, ClassNotFoundException
  - Compiler requires "catch or declare"
    - Catch and handle exception in method, **OR**
    - Declare method can throw exception, forcing calling function to catch or declare exception in turn
  - **Example:** Caught.java, Declared.java

Miscellaneous

- Use exceptions only for rare events
  - Not for common cases (e.g., checking end of loop)
  - High overhead to perform catch
- Use existing Java Exceptions if possible
- Avoid simply catching & ignoring exceptions
  - catch (Exception e) { } // Nothing in between { }
  - Poor software development style
- An exception can be rethrown

```
catch (ExceptionType e) {
        throw e;
}
```

- **Example:** ReadNegativeValueRethrow.java
- Example: Additional exceptions examples in **otherExamples** package