CMSC 132:
Object-Oriented Programming II

Program Correctness, Exceptions

Department of Computer Science

University of Maryland, College Park

Overview

- **Program correctness is determined by the presence / absence of program defects (errors)**

- **Issues**
  - **Types of errors**
  - **Testing**
  - **Debugging**
  - **Exceptions**

Program Errors

- **Types of errors**
  - **Compile-time (syntax) errors**
  - **Run-time errors**
  - **Logic errors**

Program Errors – Compile Time

- **Compile-time (syntax) errors**
  - **Errors in code construction**
    - **Lexical (typographical), grammatical, types**
  - **Detected during compilation**
  - **Usually easy to correct quickly**
- **Examples**
  - **Misspelled keyword**
  - **Missing or misplaced symbol**
  - **Incorrect operator for variable type**

Program Errors – Run Time

- **Run-time errors**
  - Operations illegal / impossible to execute
  - Detected during program execution
    - But not detectable at compile time
  - Treated as exceptions in Java
- **Example**
  - Division by zero
  - Array index out of bounds
  - Using null pointer
  - Illegal format conversion

Program Errors – Logic

- **Logic errors**
  - Operations leading to incorrect program state
  - May (or may not) lead to run-time errors
  - Problem in design or implementation of algorithm
- **Examples**
  - Computing incorrect arithmetic value
  - Ignoring illegal input
- **Hardest error to handle**
  - Detect by testing
  - Fix by debugging

Testing

- **Run program (or part of program) under controlled conditions to verify behavior**
  - **Detects run-time error if exception thrown**
  - **Detects logic error if behavior is incorrect**
- **Issues**
  - **Selecting test cases**
  - **Testing different parts of program**
  - **Visibility of program code**
  - **Test coverage**
  - **…**

Test Coverage

- Test coverage
  - Whether code is executed by some test case
  - Automatically calculated by submit server
    - For set of tests selected (from link)
      - E.g., student tests, public tests, student+public tests
    - For conditionals, reports X/Y where
      - X = # tests executing True
      - Y = # tests executing False
    - Color
      - Green = executed by some test case
      - Pink = not executed

Test Coverage Example

Source Code

Coverage information for public test #all:

Source file	statements	conditionals	methods	total
Utilities.java	4/10	1/5	1/2	

```
 1          package utilities;
 2
 3          public class Utilities {
 4        2     public static String letterGrade(double numericGrade) {
 5      1/1         if (numericGrade >= 90.0)
 6        1             return "A";
 7      1/0         else if (numericGrade >= 80.0)
 8        1             return "B";
 9      0/0         else if (numericGrade >= 70.0)
10        0             return "C";
11      0/0         else if (numericGrade >= 60.0)
12        0             return "D";
13                  else
14        0             return "F";
15              }
16
17        0     public static boolean passingNumericGrade(double numericGrade) {
18      0/0         return numericGrade >= 70.0 ? true : false;
19              }
20          }
```

Debugging

- Process of finding and fixing software errors
  - After testing detects error
- Goal
  - Determine cause of run-time & logic errors
  - Correct errors (without introducing new errors)
- Similar to detective work
  - Carefully inspect information in program
    - Code
    - Values of variables
    - Program behavior

Debugging – Approaches

- **Classic**
  - **Insert debugging statements**
  - **Trace program control flow**
  - **Display value of variables**
- **Modern**
  - **IDE (integrated development environment)**
  - **Interactive debugger**

Interactive Debugger

- **Capabilities**
  - Provides trace of program execution
  - Shows location in code where error encountered
  - Interactive program execution
    - Single step through code
    - Run to breakpoints
  - Displays values of variables
    - For current state of program

Interactive Debugger

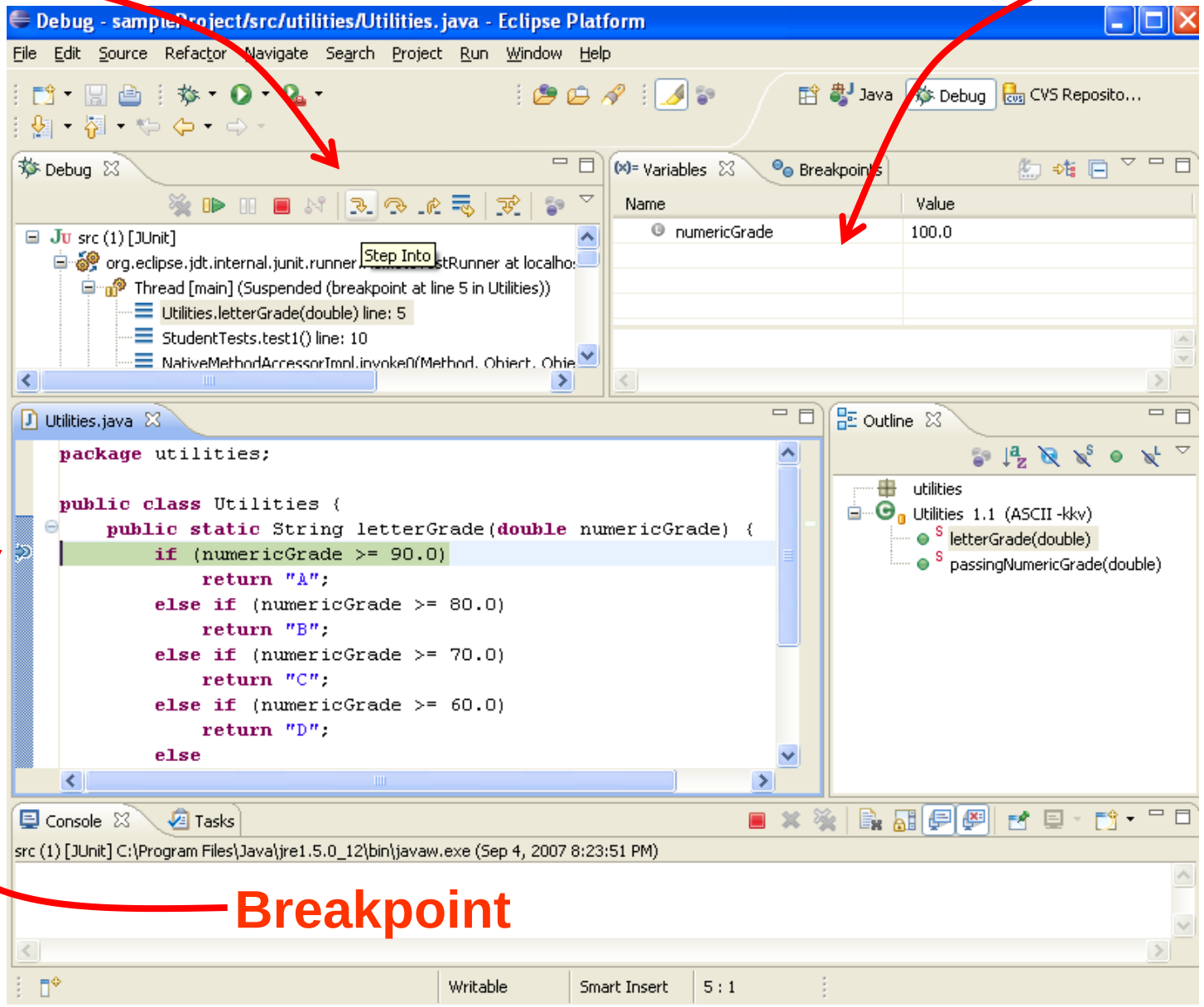
- **Single step**
  - Execute single line of code at a time
  - When executing method, can
    - Finish entire method
    - Execute first line in method
  - Tedious (or impractical) for long-running programs

- **Breakpoint**
  - Specify location(s) in code
  - Execute program until breakpoint encountered
  - Can skip past uninteresting code

Eclipse Debugger

Single Step

Data Display



Breakpoint

Exceptions

- 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

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**

```
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**

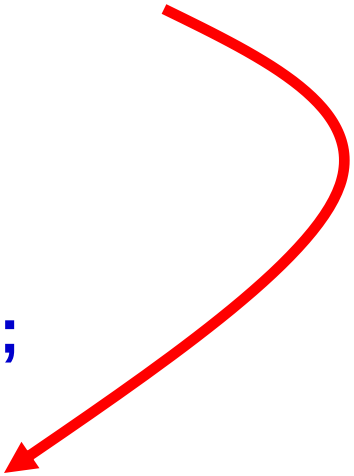
  A( ) { if (error) return (-1); }

  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**
- **Example**

```
A( ) {
    if (error) throw new ExceptionType();
}
B( ) {
    try {
        A( );
    }
    catch (ExceptionType e) { ...action... }
}
```



Java exception backtracks to caller(s) until matching catch block found

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**

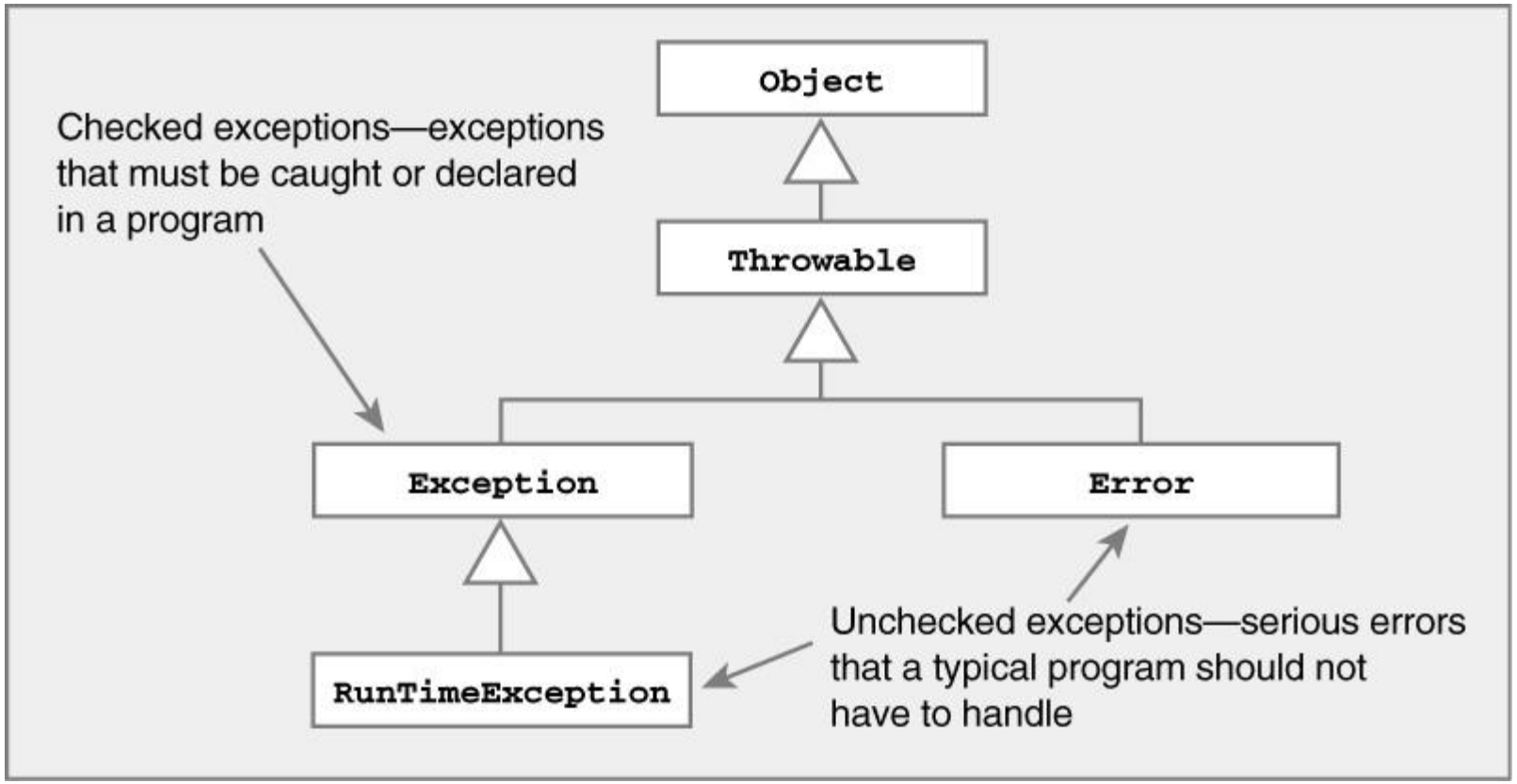
Representing Exceptions in Java

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

```
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
}
```

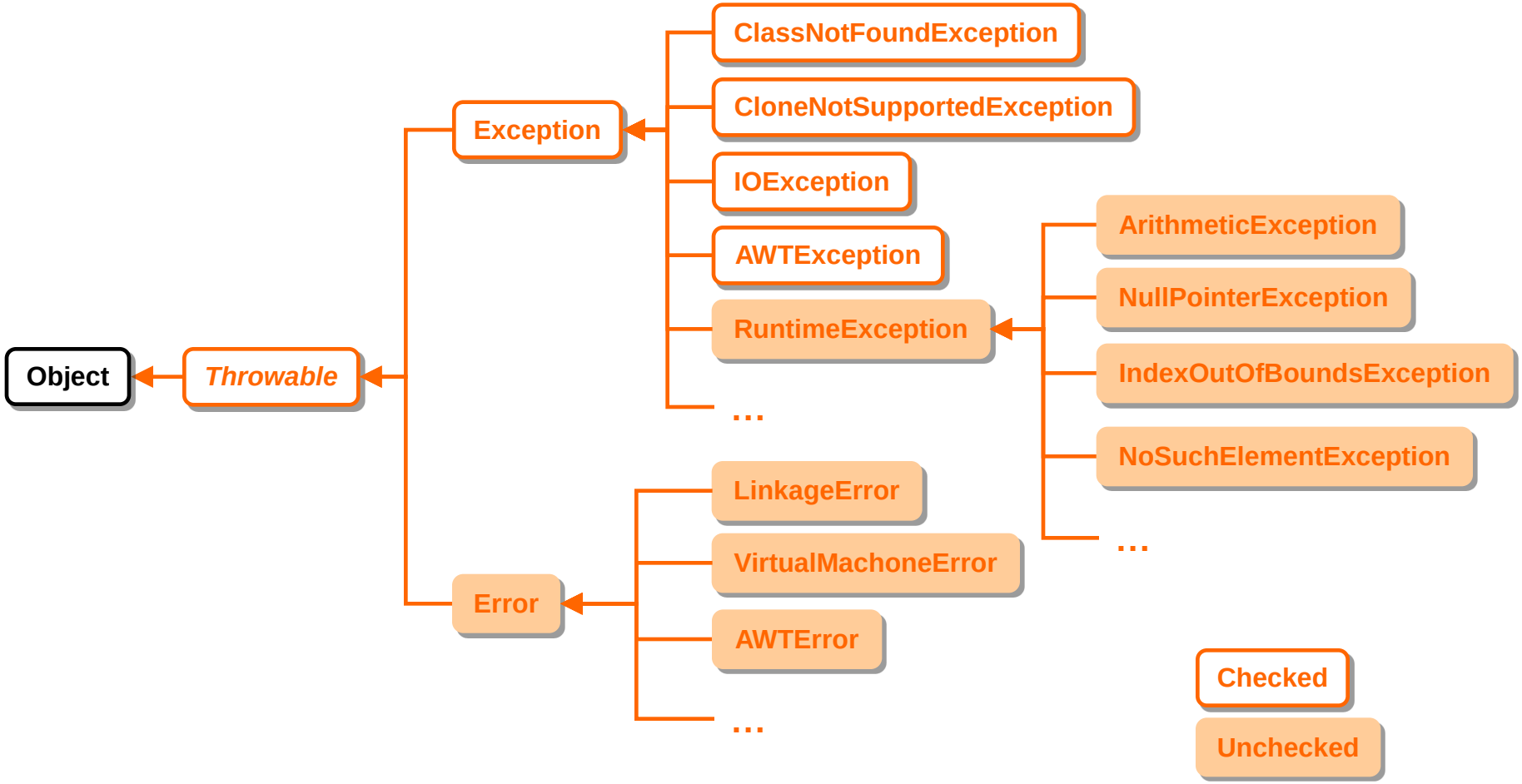
Representing Exceptions

- **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();
}
catch (eType1 e) {
    ...action...
}
catch (eType2 e) {
    ...action...
}
finally {
    ...action...
}
```

// try block encloses throws
// throw jumps to catch

// catch block 1
// run if type match

// catch block 2
// run if type match

// 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**

```
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
  - Superclass may subsume catch for subclass
    - If catch block for superclass occurs first

Java Primitive – Catch

- **Example**

```
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

```
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**

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

Exceptions – Examples

- FileNotFoundException (java.io)
  - Request to open file fails
- IllegalArgumentException (java.lang)
  - Method passed illegal / inappropriate argument
- IOException (java.io)
  - Generic I/O error
- NullPointerException (java.lang)
  - Attend to access object using null reference
- UnsupportedOperationException (java.lang)
  - Object does not provide requested operation

Exceptions – Examples

- **Used in programming project**

```
public void MethodRequiredForProject() {
    throw new UnsupportedOperationException(
      "You must implement this method.");
}
```

- **Behavior**
  - **If method is invoked during program execution**
  - **Exception is thrown**
    - **Of type UnsupportedOperationException**
    - **Message string is displayed**
  - **Program execution stops unless exception caught**