Exception Handling

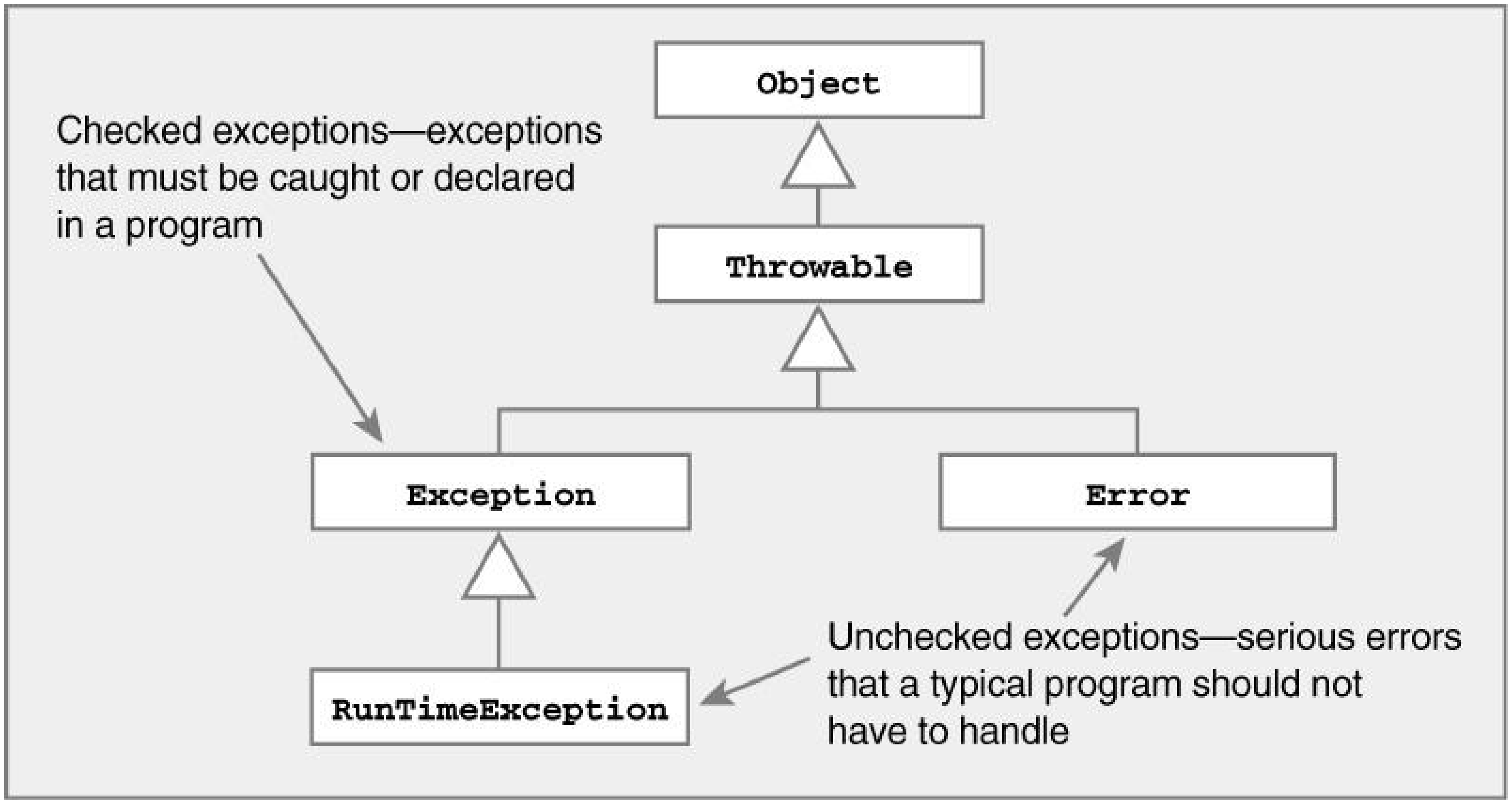
Exception Introduction

- ✓ An exception is an abnormal condition that arises in a code sequence at run time
- ✓ A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- ✓ When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- ✓ An exception can be caught to handle it or pass it on
- ✓ Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

Exception Handling

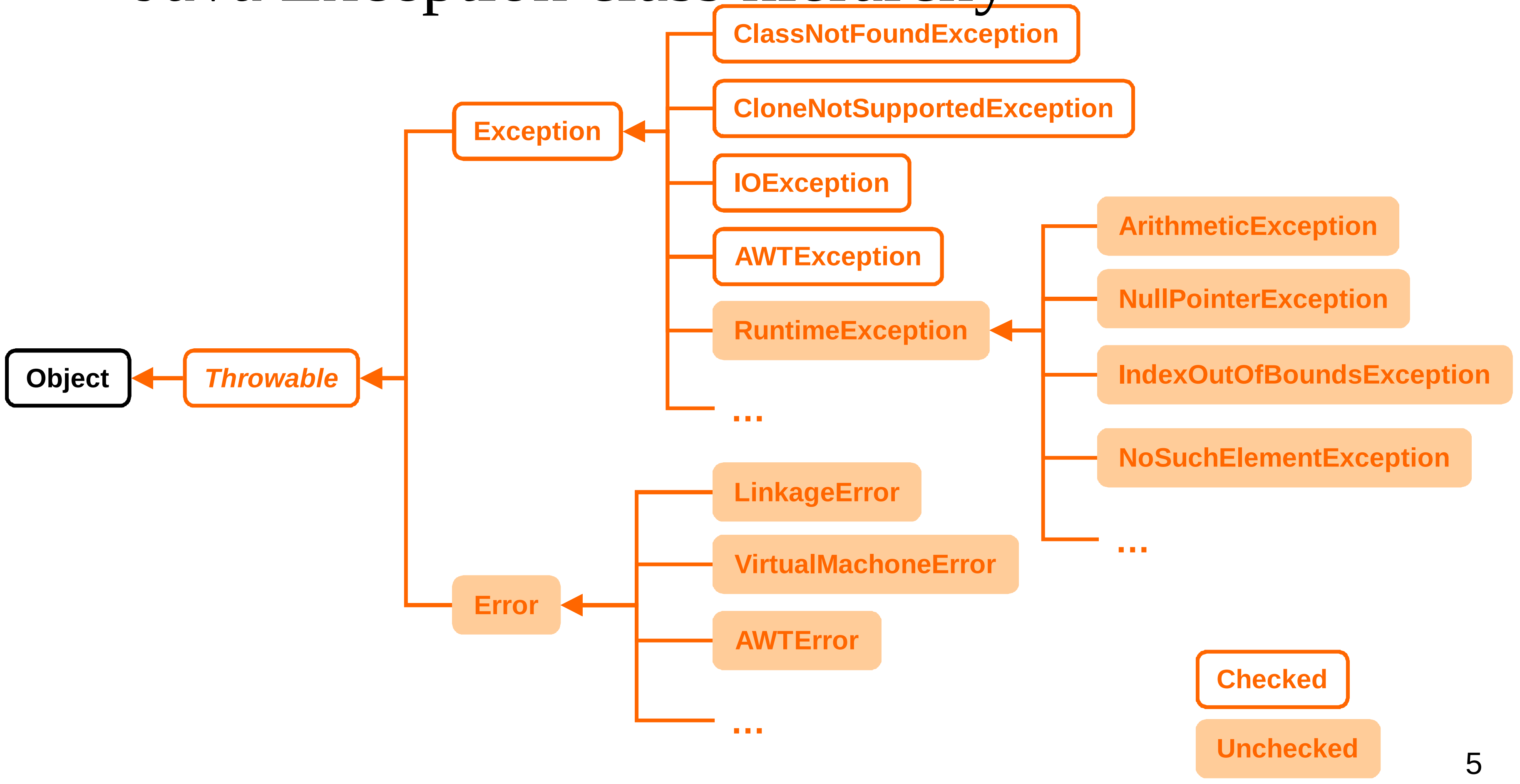
- ✓Performing action in response to exception
- ✓Examples
  - ✓Exit program (abort)
  - ✓Deal with exception and continue
    - ✓Print error message
    - ✓Request new data
    - ✓Retry action

Scope: Representing Exceptions


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



Exception Handling in Java

- ✓ Java exception handling is managed by via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- ✓ Program statements to monitor are contained within a **try** block
- ✓ If an exception occurs within the **try** block, it is thrown
- ✓ Code within **catch** block catch the exception and handle it

Example

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output:

```
Division by zero.

After catch statement.
```

try and catch statement

- ✓ The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- ✓ A **catch** statement cannot catch an exception thrown by another **try** statement.
- ✓ The statements that are protected by the **try** must be surrounded by curly braces.

Multiple Catch Clauses

- ✓ If more than one can occur, then we use multiple catch clauses
- ✓ When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- ✓ After one **catch** statement executes, the others are bypassed

Example

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Caution

- ✓ Exception subclass must come before any of of their superclasses
- ✓ A **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- ✓ For example, **ArithmeticException** is a subclass of **Exception**
- ✓ Moreover, unreachable code in Java generates error

Example

```
/*  This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

- ✓ A **try** statement can be inside the block of another try
- ✓ Each time a **try** statement is entered, the context of that exception is pushed on the stack
- ✓ If an inner **try** statement does not have a catch, then the next **try** statement’s catch handlers are inspected for a match
- ✓ If a method call within a **try** block has **try** block within it, then then it is still nested **try**

Example

```java
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

throw

- ✓It is possible for your program to to throw an exception explicitly

  throw *ThrowableInstance*

- ✓Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- ✓There are two ways to obtain a **Throwable** objects:
  - ✓Using a parameter into a catch clause
  - ✓Creating one with the **new** operator

Example -throw Statements

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // re-throw the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

Output:

```
Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo
```

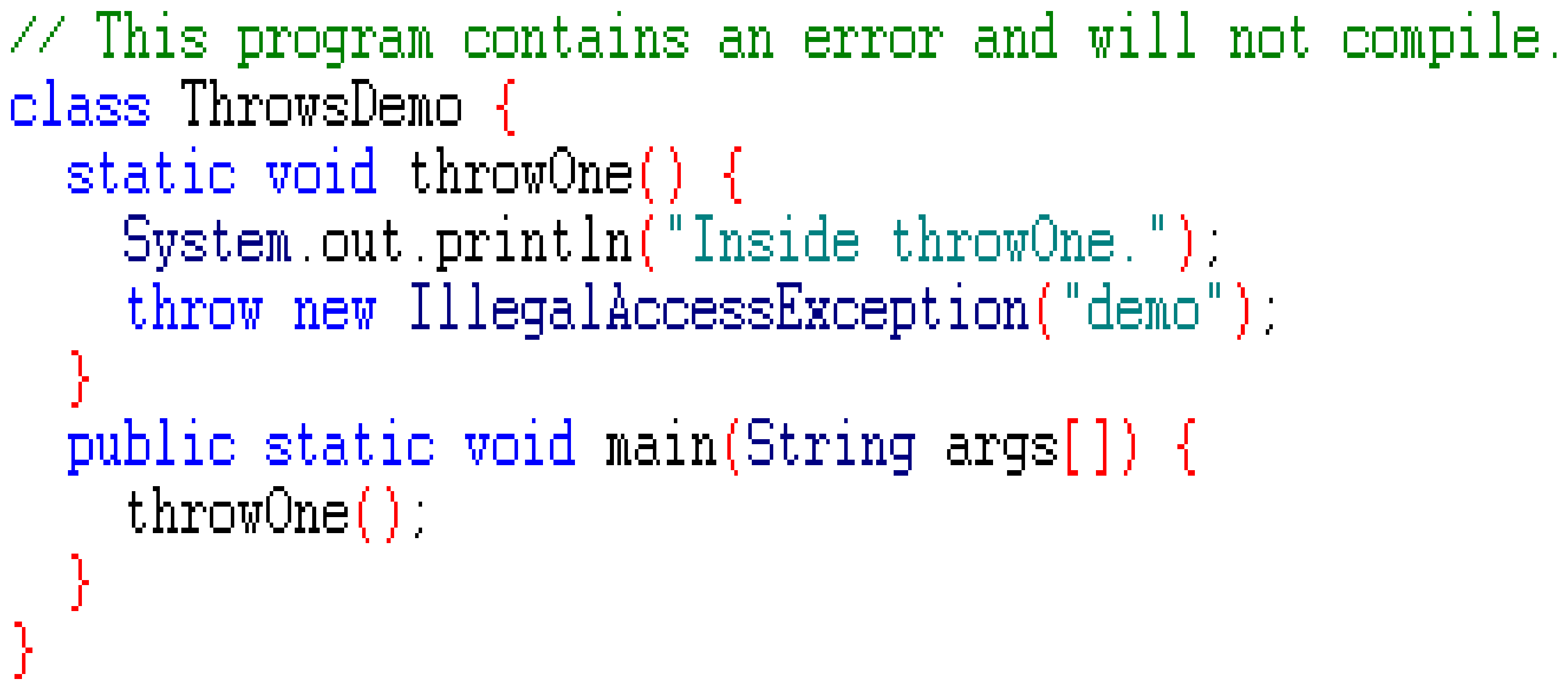
throws

- ✓ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- ✓ *type method-name parameter-list)* throws *exception-list*

```
  {
        // body of method
  }
```

- ✓ It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

Example: incorrect program



```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

Example: corrected version

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

```
Inside throwOne.

Caught java.lang.IllegalAccessException: demo
```

Finally Statement

- ✓ **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- ✓ **finally** block will be executed whether or not an exception is thrown.
- ✓ Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- ✓ Each try clause requires at least one catch or finally clause.

Example

```
// Demonstrate finally.
class FinallyDemo {
  // Through an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

Output

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Uncaught Exceptions

```
class exc0{
public static void main(String args[])
{
    int d=0;
    int a=42/d;
}
}
```

- ✓ A new exception object is constructed and then thrown.
- ✓ This exception is caught by the default handler provided by the java runtime system.
- ✓ The default handler displays a string describing the exception, prints the stack trace from the point at which the exception occurred and terminates the program.

Output:

```
java.lang.ArithmeticException: / by zero
        at exc0.main(exc0.java:4)
```

Displaying a Description of an Exception

- ✓ **Throwable** overrides the toString() method (defined by **Object**) so that it returns a string containing a description of the exception.
- ✓ Example:

```
catch(ArithmeticException e)
{
        System.out.println("Exception: "+e);
}
```

- ✓ **Output:**

```
Exception: java.lang.ArithmeticException: / by zero
```

User Defined Exception

- ✓ Define a subclass of the Exception class.
- ✓ The new subclass inherits all the methods of Exception and can override them.

```
class MyException extends Exception{
private int a;
MyException(int i) { a = i;}
public String toString (){ return “MyException[“ + a+”]”;}
}
```

Continuation of the Example

```
class test{
    static void compute (int a) throws Myexception{
            if(a>10) throw new MyException(a);
            System.out.println(“Normal Exit”);
    }
public static void main(String args[]){
    try{
            compute(1);
            compute(20);
            }catch(MyException e){ System.out.println(“Caught “ +e);
}
}
```

Example-2

```
class InvalidRadiusException extends Exception {
        private double r;
        public InvalidRadiusException(double radius){
                r = radius;
        }
        public void printError(){
                System.out.println("Radius [" +  r + "] is not valid");
        }
}
```

Continuation of Example-2

```
class Circle    {
        double x, y, r;

       public Circle (double centreX, double centreY, double radius ) throws
     InvalidRadiusException {
        if (r <= 0 ) {
                throw new InvalidRadiusException(radius);
        }
        else {
                x = centreX ; y = centreY;  r = radius;
        }
       }
}
```

Continuation of Example-2

```
class CircleTest {
        public static void main(String[] args){
         try{
                 Circle c1 = new Circle(10, 10, -1);
                 System.out.println("Circle created");
         }
         catch(InvalidRadiusException e)
         {
                 e.printError();
         }
        }
}
```