I/O and Exceptions

I/O

- Input
  - standard input
    - keyboard (System.in)
  - command line
  - file
- Output
  - standard output
    - Screen (System.out)
  - standard err
    - System.err
  - file

- Input ของจาวา จะเป็น stream
- จะอ่าน stream ใช้คลาส **Scanner**

Input

- Scanner
  - Scanner in = new Scanner(...input stream...);

- จะอ่านจากไหนก็ใส่ input stream เป็นอันนั้น เช่น
  - อ่านจาก keyboard
    - new Scanner(**System.in**)
  - อ่านจาก file ต้องสร้าง FileReader
    - new Scanner(**new FileReader**(**ชื่อไฟล์**))

อ่านจาก internet

```
import java.net.URL;

// ใน main
URL url =
   new URL("http://www.cp.eng.chula.ac.th");

Scanner in = new Scanner(url.openStream());

...

in.close();
```

Scanner

- hasNext()

- nextInt()
- nextDouble()
- next()
- nextLine()

Print to file

- สร้าง PrintStream หรือ
- สร้าง **PrintWriter** (แบบใหม่)

PrintWriter out = new **PrintWriter**(**ชื่อไฟล์**);

out.print(…);
out.println(…);

out.close();

Error Handling with Exceptions

- Java: “Badly-formed code will not be run”
- Ideal – catch at compile time
- Fact – not all errors can be detected at compile time. They must be handled at run time
- Run-time error handling integrated into the core of the language, enforced by the compiler

The problem

- Dealing with errors during program execution
- What causes errors?
  - Program logic (i.e.: exceeding array bounds)
    - Can be fixed by the programmer
  - Status of the environment (i.e.: network goes down)
    - out of control by the programmer

What is an exception?

- เหตุการณ์ (ผิดปกติ) ที่ทำให้การทำงานของโปรแกรมหยุดชะงัก
- ในจาวาเป็น object ซึ่งคอยส่งสัญญาณ error condition และให้ข้อมูลเกี่ยวกับ error.
- Enforced by the Java compiler (cannot be ignored).

Advantages

- แยกส่วนที่เป็น error handling code ออกจากส่วนอื่นๆ เรียกว่า exception handler ทำให้อ่านง่าย
- Propagating errors up the call stack
- Grouping error types and error differentiation

Advantage 1

- Separating error handling code from regular code and handling all the problems in only one place, called *exception handler.*

```
// pseudo-code of a function

readFile {
  open the file;
  determine its size;
  allocate that much memory;
  read the file into memory;
  close the file;
}
```

Traditional error management (1)

- Potential errors
  - What happens if the file can't be opened?
  - What happens if the length of the file can't be determined?
  - What happens if enough memory can't be allocated?
  - What happens if the read fails?
  - What happens if the file can't be closed?
- Putting error detections into your "regular" code can lead to confusing spaghetti code.

Traditional error management (2)

```
errorCodeType readFile {
  initialize errorCode = 0;
  open the file;
  if (theFileIsOpen) {
    determine the length of the file;
    if (gotTheFileLength) {
      allocate that much memory;
      if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) errorCode = -1;    // read failed
      } else errorCode = -2;               // not enough memory
    } else errorCode = -3;                 // file size can't be determined
    close the file;
    if (theFileDidntClose && errorCode == 0) {
      errorCode = -4;                      // can't close file
    } else errorCode = errorCode and -4;   // can't close file + error
  } else errorCode = -5;                   // can't open file
  return errorCode;
}
```

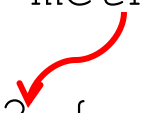
Using exception handling

```
readFile {
  try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
  } catch (fileOpenFailed) {
      doSomething;
  } catch (sizeDeterminationFailed) {
      doSomething;
  } catch (memoryAllocationFailed) {
      doSomething;
  } catch (readFailed) {
      doSomething;
  } catch (fileCloseFailed) {
      doSomething;
  }
}
```

Advantage 2

- Propagating errors up the call stack
- Suppose that only method1 is interested in error from readFile.

```
method1 {
   call method2;
}
method2 {
   call method3;
}
method3 {
   call readFile;
}
```



Without exception

```
method1 {
  errorCodeType error;
  error = call method2;
  if (error)
    doErrorProcessing;
  else
    proceed;
}
method2 {
  errorCodeType error;
  error = call method3;
  if (error)
    return error;
  else
    proceed;
}
```

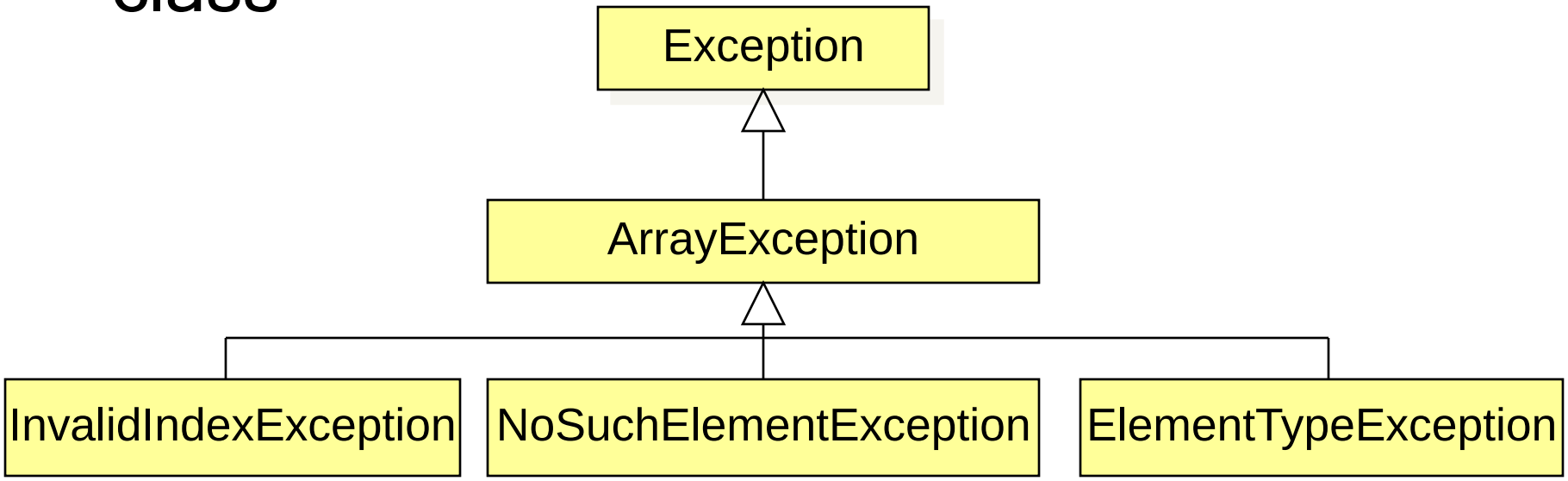
```
method3 {
  errorCodeType error;
  error = call readFile;
  if (error)
    return error;
  else
    proceed;
}
```

With exception

```
method1 {
   try {
      call method2;
   } catch (exception) {
      doErrorProcessing;
   }
}
method2 throws exception {
   call method3;
}
method3 throws exception {
   call readFile;
}
```

Advantage 3

- Grouping error types and error differentiation
- All exceptions are subclass of Throwable class



Advantage 3 (ต่อ)

- If interested in a specific exception

```
catch (InvalidIndexException e) {
   . . .
}
```

- If interested in all exceptions about array

```
catch (ArrayException e) {
   . . .
}
```

- We can catch either specific exceptions or general exceptions or both.

Throwing an Exception

- ถ้าเราไม่มีข้อมูลที่จะจัดการกับ exception ที่เกิดขึ้นได้เราก็จะโยนต่อ

- **throw** an exception:

```
if (t == null)
  throw new NullPointerException();
```

ชื่อของ class จะบอกประเภทของ exception

- Exception arguments

```
if (t == null)
  throw new NullPointerException("t=null");
```

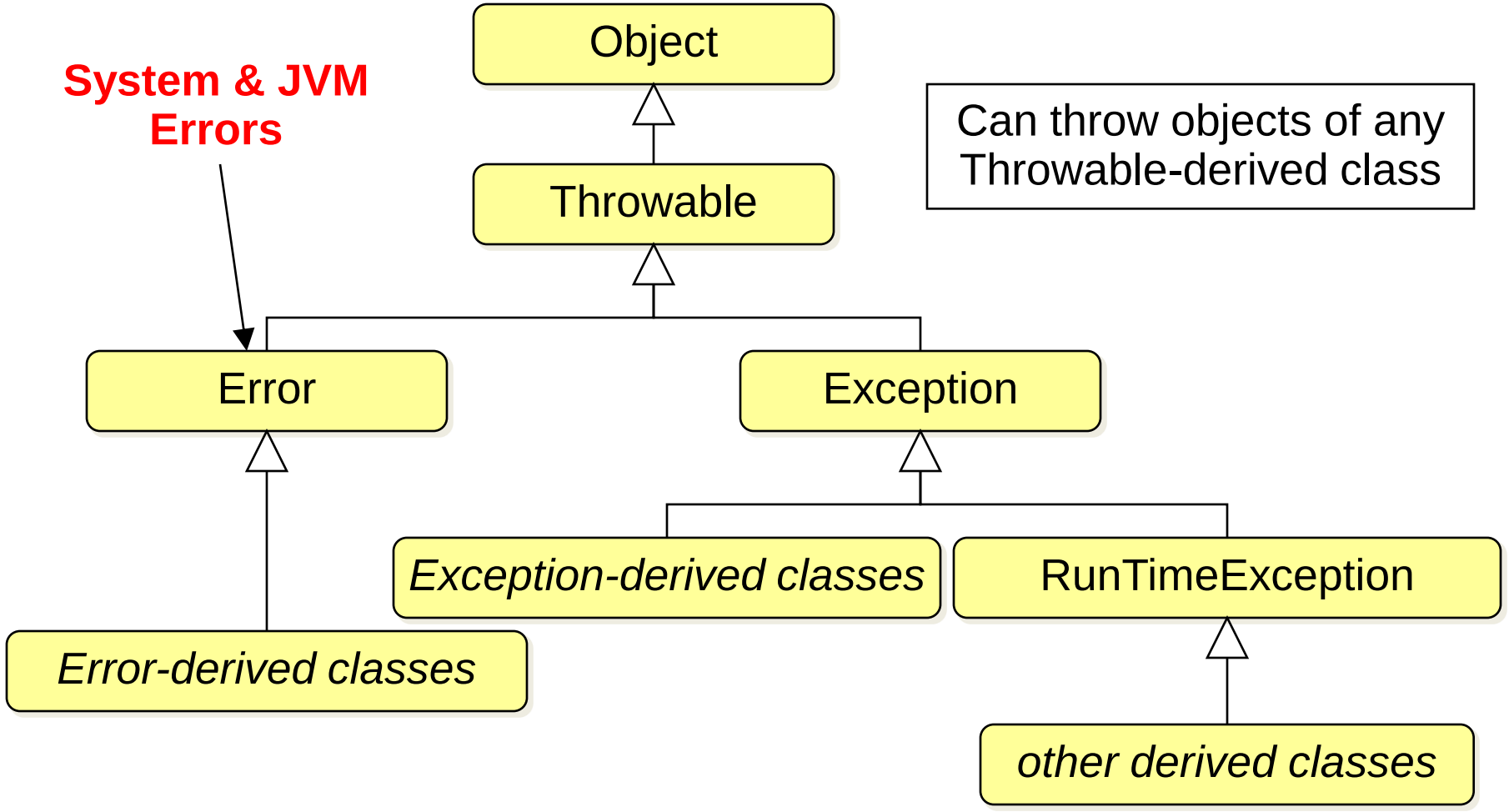
Termination vs. Resumption

- In *termination*, you don't want to deal with the error, so you throw the exception.
- If a method chooses not to catch an exception, the method must specify (in its signature) that it can throw that exception.

```
public void doXXX() throws SomeException { ... }
```

- In *resumption*, you presume that it will be successful the second time.
- Put the **try** block inside an infinite while loop until the result is satisfied.

Class hierarchy for error handling

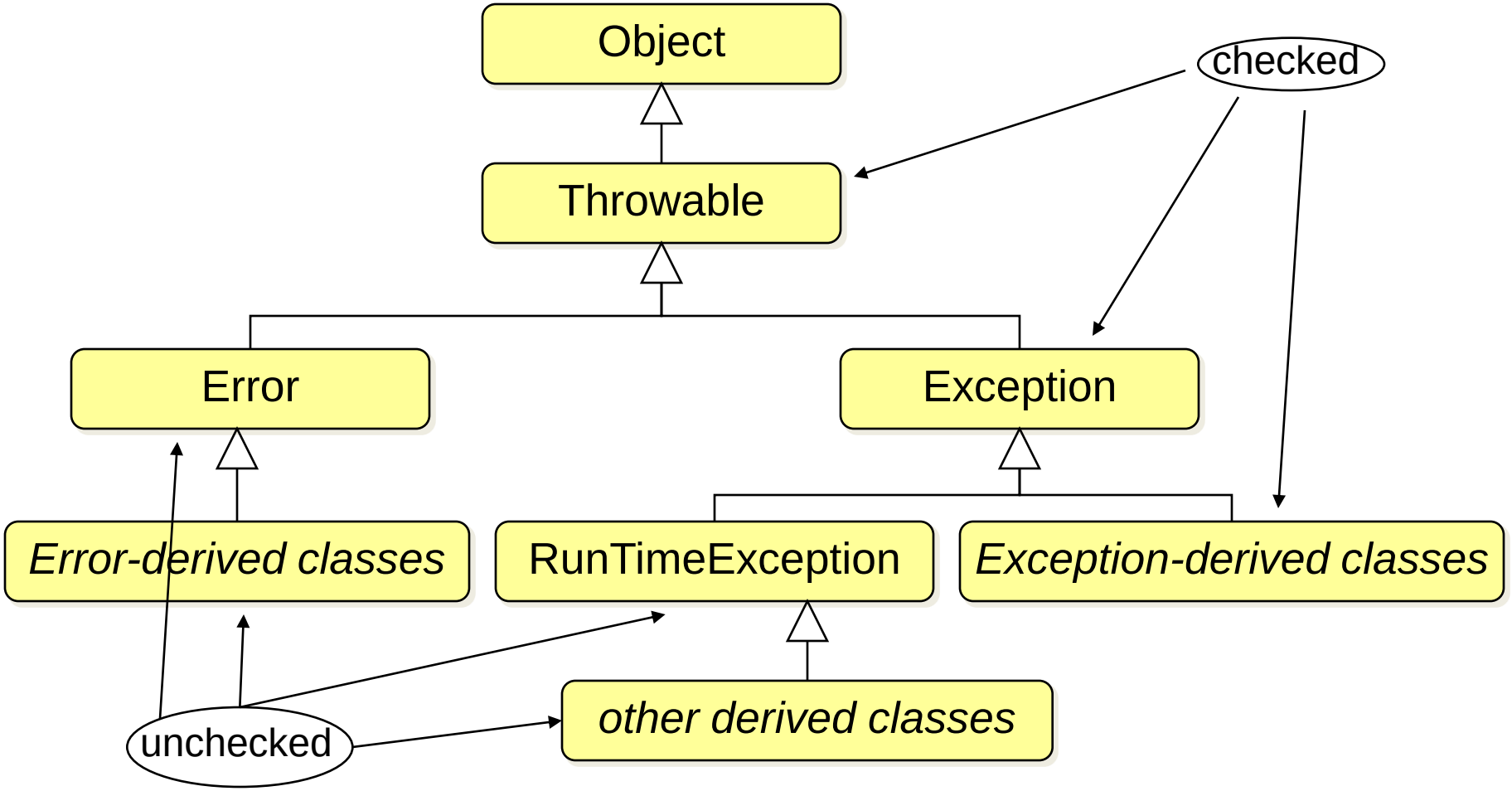


Checked vs. Unchecked

- *Checked Exceptions* must be specified to handle or pass on
  - *exceptions that are expected to happen*
  - *forced by compiler*

- *Unchecked Exceptions*
  - *exceptions derived from RuntimeException*

Class hierarchy for error handling



The exception specification

- It's a part of the method declaration, appearing after the argument list.

```
void f() throws Exception1,
                Exception2,
                . . .,
                ExceptionN { // . . .
```

Catching an Handling Exceptions (1)

- The **try** block: a guarded region

```
try {
   // code that may
   // generate exceptions
}
```

- A try statement *must* be accompanied by at least one catch block or one finally block.

Catching an Handling Exceptions (2)

- The **catch** block(s): exception handler

```
try {
   // code that might generate exceptions
} catch (Type1 id1) {
   // handle exceptions of Type1
} catch (Type2 id2) {
   // handle exceptions of Type2
} catch (Type3 id3) {
   // handle exceptions of Type3
}
// etc. . .
```

Catching an Handling Exceptions (3)

- The **finally** block: clean up mechanism

```
try {
    // code that might generate exceptions
  } catch (Type1 id1) {
    // handle exceptions of Type1
  } catch (Type2 id2) {
    // handle exceptions of Type2
  } catch (Type3 id3) {
    // handle exceptions of Type3
  }
  finally {
    // code guarantee
    // to be executed
  }
}
```

```
class ThreeException extends Exception {}

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while (true) {
      try {
        // post-increment is zero first time:
        if (count++ == 0)
          throw new ThreeException();
        System.out.println("No exception");
      } catch (ThreeException e) {
        System.out.println("ThreeException");
      } finally {
        System.out.println("In finally clause");
        if (count == 2) break;
      }
    }
  }
}
```

```
ThreeException
In finally clause
No exception
In finally clause
```

What's "finally" for?

- Always gets called, regardless of what happens with the exception and where it's caught
- To set something *other* than memory back to its original state (GC handles memory) (close files, network connections, etc.)

```
try {
   openFile();
   readFile();
   closeFile();
} catch (Exception1 e) {
   // handle exception1
   closeFile();
} catch (Exception2 e) {
   // handle exception2
   closeFile();
}
```

- duplicate `closeFile()`
- If new catch is added, `closeFile()` must be called.
- more error-prone

```
try {
   openFile();
   readFile();
} catch (Exception1 e) {
   // handle exception1
} catch (Exception2 e) {
   // handle exception2
} finally {
   closeFile();
}
```

Rethrowing an exception

- You may want to rethrow the exception that you just caught, or throw a new exception.

```
catch (Exception1 e) {
  System.err.println("An exception was thrown");
  throw e;
}
```

```
catch (Exception1 e) {
  System.err.println("An exception was thrown");
  throw new Exception2();
}
```

Creating your own exceptions

- Choosing the exception type to throw
  - Use one written by someone else
  - Write one of your own
- Choosing a superclass
  - Must be a subclass of Throwable
    - Error
    - Exception or Exception-derived class
- Naming conventions
  - Having “Exception” to the end of all Exception-derived classes
  - Having “Error” to the end of Error-derived classes

Creating your own exceptions

```
class InvalidCharException extends Exception {}
```

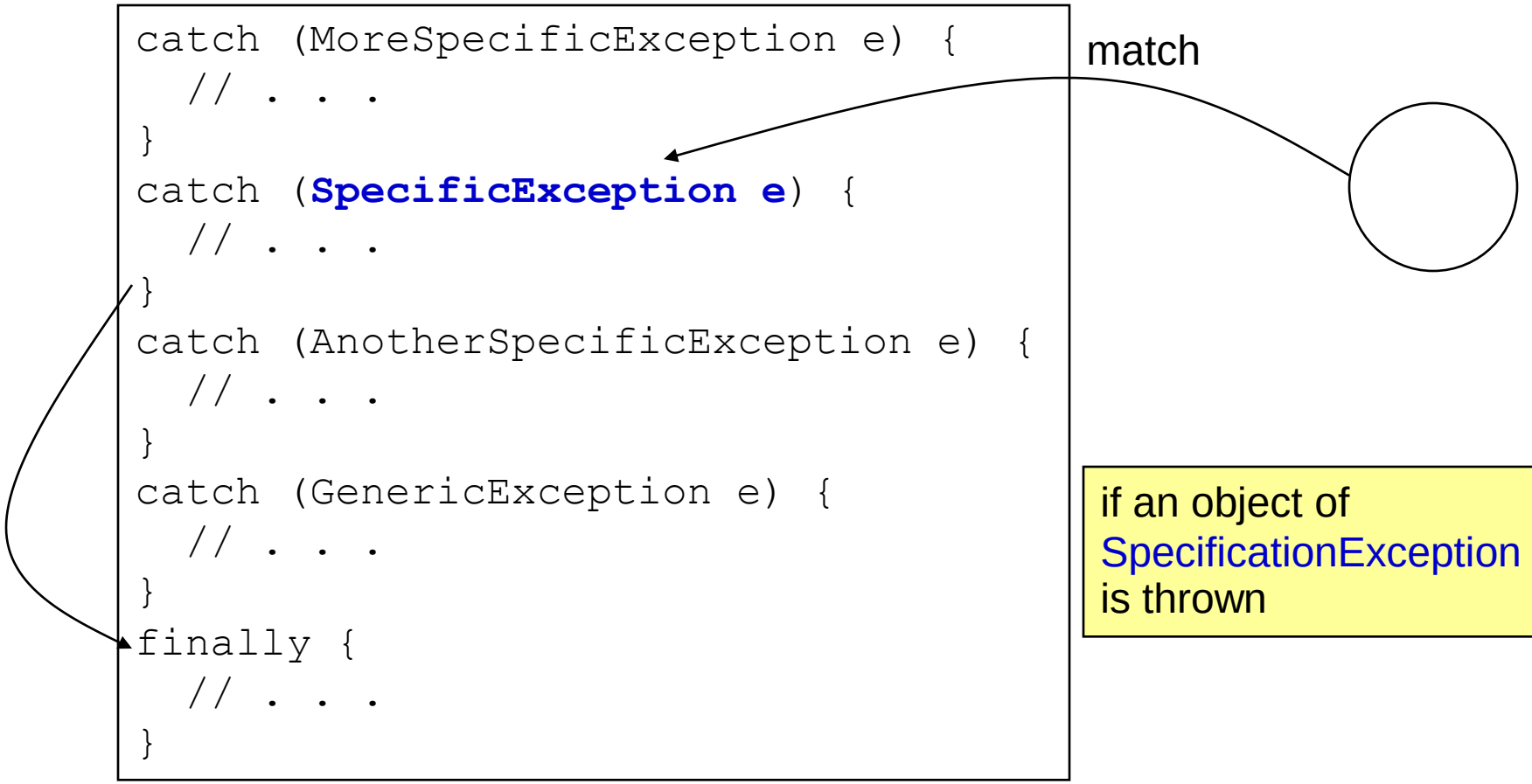
```
class InvalidCharException extends Exception {
  public InvalidCharException() {}
  public InvalidCharException(String msg) {
    super(msg);
  }
}
```

Exception matching

- Only one catch block will be executed
- Sequential search in the order they are written
- Does not required a ***perfect*** match
  - Is the thrown exception “is-a” object of the exception in the catch block?
    - No – continue searching
    - Yes – exception match

```
class GenericException extends Exception {}
class SpecificException extends GenericException {}
class AnotherSpecificException extends GenericException {}
class MoreSpecificException extends SpecificException {}
```

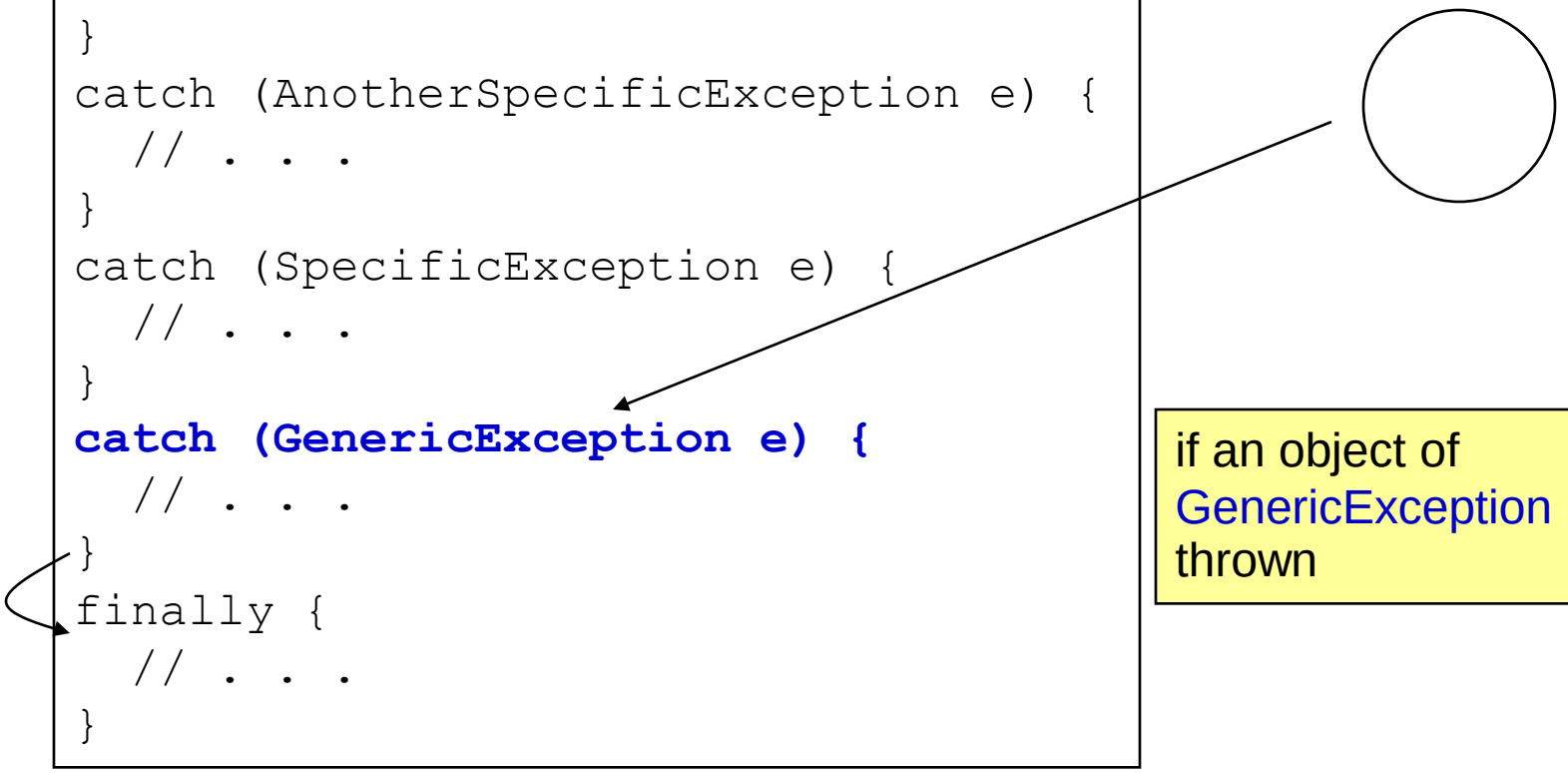
```
catch (MoreSpecificException e) {
  // . . .
}
catch (SpecificException e) {
  // . . .
}
catch (AnotherSpecificException e) {
  // . . .
}
catch (GenericException e) {
  // . . .
}
finally {
  // . . .
}
```



if an object of SpecificationException is thrown

```
class GenericException extends Exception {}
class SpecificException extends GenericException {}
class AnotherSpecificException extends GenericException {}
class MoreSpecificException extends SpecificException {}
```

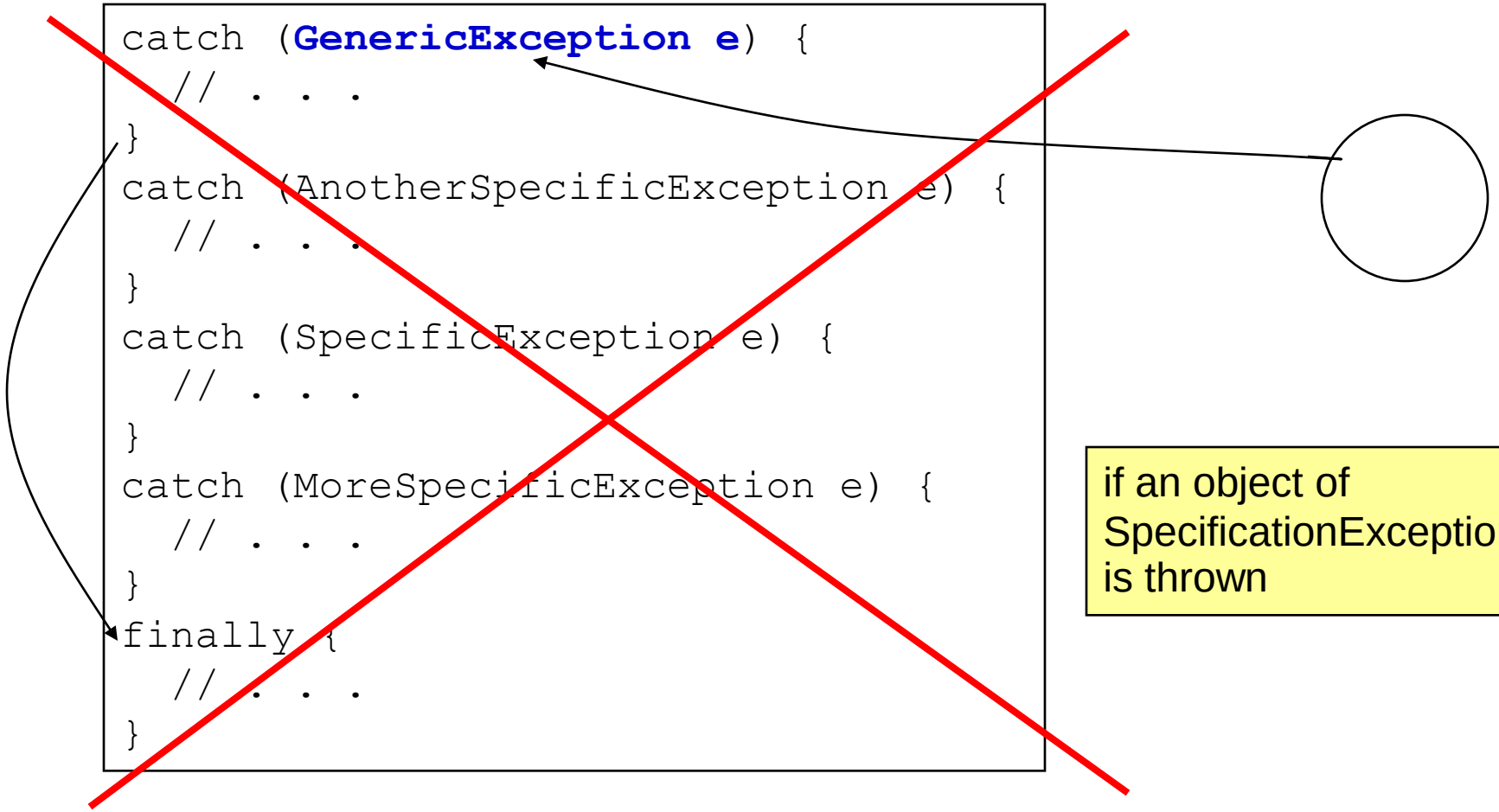
```
catch (MoreSpecificException e) {
   // . . .
}
catch (AnotherSpecificException e) {
   // . . .
}
catch (SpecificException e) {
   // . . .
}
catch (GenericException e) {
   // . . .
}
finally {
   // . . .
}
```



if an object of GenericException is thrown

```
class GenericException extends Exception {}
class SpecificException extends GenericException {}
class AnotherSpecificException extends GenericException {}
class MoreSpecificException extends SpecificException {}
```

```
catch (GenericException e) {
  // . . .
}
catch (AnotherSpecificException e) {
  // . . .
}
catch (SpecificException e) {
  // . . .
}
catch (MoreSpecificException e) {
  // . . .
}
finally {
  // . . .
}
```



if an object of SpecificationException is thrown

Exception matching

- Catch the **most specific exception** (leaf node of the hierarchy) first.
- Go up the hierarchy
- Put the most generic one (base-class) at the very last catch block
- Base-class handler will catch derived-class object

Overhead

- Exceptions are free as long as they don't get thrown
- If they are thrown, very expensive
- Don't use exceptions for normal flow of control
- Only use exceptions to indicate abnormal conditions

Guidelines

- Handle an exception only if you have enough information to correct the error
  - let the exception propagate up (throw it)
- Separate error handling code (which almost never runs) from the code that represents the normal path of execution
  - Make code more readable

Guidelines

- Handle tasks, not statements
  - put tasks inside of a try block
- Use loops to retry
- If you catch an exception, do something with it
- Clean up using **`finally`**
- Use exception in constructors
  - sometimes construction does not succeed