COMP 249: Object Oriented Programming II

CHAPTER 9 - EXCEPTION HANDLING

N. ACEMIAN

Run Time Errors

- Programs often don't work as intended

- We need a mechanism to determine when exceptional" conditions have occurred

Traditional Error Handling

- The C language approach
- Errors must be checked at the point where they occur
  - Typically, functions return an error code that is manually checked

```
x.doA();
x.doB();
x.doC();
…
```

```
x.doA();
if (doA went wrong)
  handle the doA problems;
else
  x.doB();
  if (doB went wrong)
    handle the doB problems
  else
    x.doC();
    if (doC went wrong)
    handle the doC problems
…
```

Traditional Error Handling

But…

- Code can become unreadable
  - Mixture of application code and error handling
  - Obscures the logic of the program
  - Returning errors codes and values is awkward

- Programmers – even “good” ones - often skip a lot of error checking

Using Exceptions Instead

- In Java, we use ***exception objects***
- An ***exception*** is an object that describes an unusual or erroneous situation
  - ex: take the log of a negative number, division by zero, a file cannot be opened, null reference, …
- A program is now separated into:
  - the ***normal*** execution flow and
  - an ***exception*** execution flow

Exception Handling

- The default behavior if an exception occurs (i.e. the programmer did not account for it):
  - the program terminates abnormally
  - and produces an appropriate message
    - The message includes a **call stack trace** that indicates:
      - the line on which the exception occurred
      - the sequence of method calls, starting at the method that threw the exception and ending at main

Example: Zero.java

```
public class Zero {
  public static void main (String[] args) {
    int numerator = 10;
    int denominator = 0;
    System.out.println(numerator / denominator);
    System.out.println("This will not be printed");
  }
}
```

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Zero.java:5)
Press any key to continue...
```

Example: StackTrace.java

```
public class StackTrace {
  public static void main (String[] args) {
   someMethod();
  }
   private static void someMethod() {
            someOtherMethod();
  }
   private static void someOtherMethod() {
    divideByZero();
  }
  private static void divideByZero() {
    int numerator = 10;
    int denominator = 0;
    System.out.println(numerator / denominator);
    System.out.println("Thi
  }
}
```

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at StackTrace.divideByZero(StackTrace.java:17)
        at StackTrace.someOtherMethod(StackTrace.java:11)
        at StackTrace.someMethod(StackTrace.java:7)
        at StackTrace.main(StackTrace.java:3)
Press any key to continue...
```

Exception Handling Mechanism

- Default behavior may not be fine for professional programs…
- **Exception Handling Mechanism** allows us to **change** the default behavior to handle the exception more gracefully

We need to:

- attempt an operation
  - normal flow is tried --> ***try*** statement
- indicate that something has gone wrong
  - in case of trouble, throw an exception --> ***throw*** statement
- deal with the exceptional case when the operation does not succeed
  - a handler catches the thrown exception and deals with it --> ***catch*** statement

Catching Exceptions

We use the **`try-throw-catch`** model

- **`try`**:
  - a block of code in which exceptions can be generated
- **`throw`**:
  - allows to generate an exception
- **`catch`**:
  - a handler for a particular type of exception

Structure of **try-catch-finally**

```
try {
   <some code>
   <either some code with a throw statement
    or a method call that might throw an exception>
   <some more code>
}
catch(ExceptionType1 e) {
   <exception handling code>
}
catch(ExceptionType2 e) {
   <exception handling code>
}
…

finally {
   <code to execute whether an exception was thrown or
    not>
}
```

`try-catch-finally`

The **`try`** block

- includes the line that throws the exception (directly or not)
- is followed by one or more **`catch`** clauses

Each **`catch`** block:

- contains code to process an exception
- has an associated exception type
- is called an *exception handler*
- If an Exception is thrown inside of a try block, the exception that is returned is forwarded as an argument to the catch block where the Exception can be handled

`try-catch-finally`

the **`finally`** block:

- optional
- The statements in the **`finally`** clause:
  - are always executed
  - Run before Java moves up the call stack
- useful to run important code that needs to be run at the end of the try block even if an exception is thrown
  - ex: closing a file opened in a **`try`**, …
- If no exception is generated,
  - the **`finally`** clause is executed after the **`try`** block completes
- If an exception is generated,
  - the **`finally`** clause is executed after the appropriate **`catch`** block completes

Example

if you call a method like **`Integer.parseInt()`**

and the input is not in the correct format, a **`NumberFormatException`** is thrown automatically.

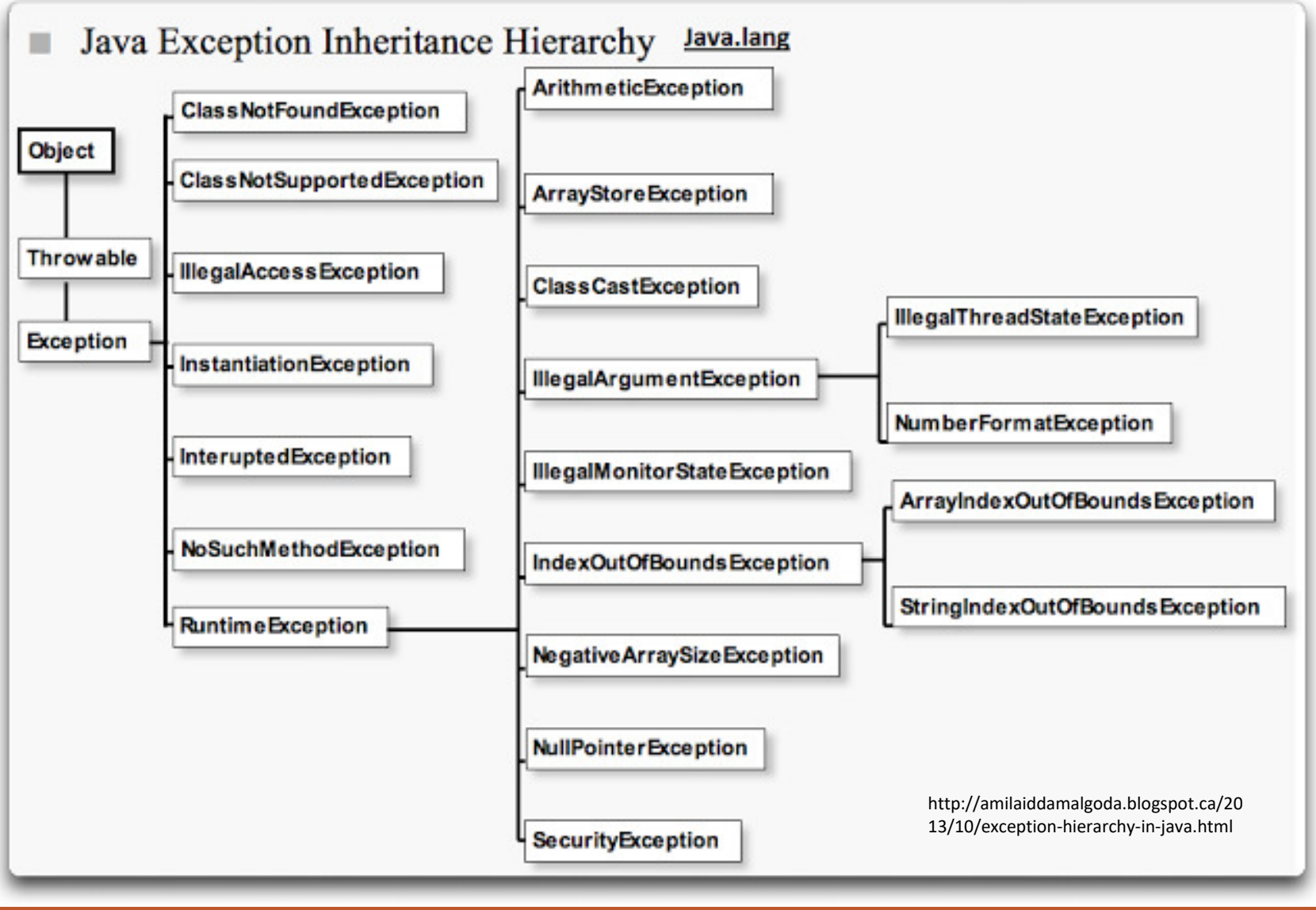
```
int x;
try {
  x = Integer.parseInt("abc");
}
catch (NumberFormatException e) {
  System.err.println("You must enter an integer");
}
```

2 things can happen

- If an exception is not thrown:
  - All of the statements in the **`try`** block executes normally.
  - The code in the **`catch`** block is not executed.
  - The code in the **`finally`** block is executed.
- If an exception is thrown:
  - In the **`try`** block, only the code up to the line that throws the exception is executed.
  - The code after the line that throws the exception is not executed.
  - The code in the appropriate **`catch`** block is executed.
  - The code in the **`finally`** block is executed.(even if ending the program)

The **Exception** Hierarchy

When an exception is thrown, one of these exception classes is instantiated



http://amilaiddamalgoda.blogspot.ca/2013/10/exception-hierarchy-in-java.html

The `Exception` Hierarchy

Example of methods in the **Exception** class:

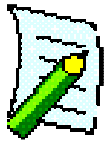
- **getCause()**
- **getMessage()** returns a string explaining why the exception was thrown.
- **printStackTrace()**

Multiple Exceptions and `catch` Blocks

- The `try` block can generate multiple exceptions
- The `catch` blocks are tested in sequence for one that catches the exception type
- Java will execute the **first** matching handler
- Will not execute other handlers once it has a match.
- …So put the `catch` blocks for the more specific exceptions early and the more general ones later

→ Catch the more specific exception first

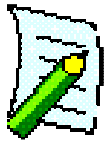
Example: MultipleExceptions.java



```
try {
 String a = "0";
 int r2 = Integer.parseInt(a) / Integer.parseInt(a);
}
catch (ArithmeticException e) {
 System.out.println("Calculation Error");
}
catch (Exception e) {
 System.out.println("General Exception");
}
finally {
 System.out.println("Finally");
}
System.out.println("Finished");
```

output?

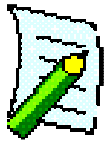
What if…



```
try {
 String a = "0";
 int r2 = Integer.parseInt(a) / Integer.parseInt(a);
}
catch (Exception e) { // general exception first
 System.out.println("General Exception");
}
catch (ArithmeticException e){
 //specific exception after
 System.out.println("Calculation Error");
}
finally {
 System.out.println("Finally");
}
System.out.println("Finished");
```

output?

Example: Which exception to catch?



```
public class MyClass {
  public static void main (String[] args)
  {
    try {
          String text = "abcde";
          System.out.print(text.charAt(10));
          int r1 = Integer.parseInt(text);
    }
    catch (ArithmeticException e) {
          System.out.println("Calculation Error!");
    }
    System.out.println("Finished!");
  }
}
```

output?

throw Syntax

```
try {
   <code to try>

   if (test condition)
     throw new Exception("Msg to display");

   <remaining code that will execute only if exception is
    not thrown>
}
catch (SomeException e) {
   <exception handling code>
}
…

finally {
   <code to execute whether the exception was thrown or not>
}
```

`throw`: What happens?

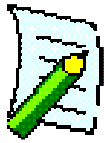
When we hit the **`throw`** statement,

1. Execution stops immediately

2. Any subsequent statements are not executed

3. Looks for matching **`catch`** statement
   - Match found --> control is transferred to that statement
   - No match --> control propagates to caller

Example: Bank Account



- First draft…

```
public class BankAccount {
  public void withdraw(double amount) {
     balance -= amount;
  }
 …
}
```

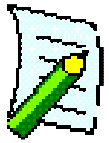
- but what if the amount to withdraw is > current balance…

```
public class BankAccount {
  public void withdraw(double amount) {




  }
 …
}
```

Example: Bank Account



- better…

```
public void withdraw (double amount) {
    try
    {
    }
    catch
    {
    }
}
```

More likely use of `try-catch`

- exception handler may or may not be inside the method that produced the problem

```
try {
  <code to try>
  someMethod();
  <more code>
}
catch (SomeException e) {
  <exception handling code>
}
<possibly more code>
```

call

normal return

```
void someMethod() throws SomeException
{
  <some code>
  if (error)
    throw new SomeException("message");
  <more code>
}
```

return from thrown exception

What if no match?

if the **`try`** clause generates an exception that you don't catch?

- either, because you did not think about it…
- or, it is more appropriate to handle the exception at a higher level
  - **Remember**: *you should place a catch block only in methods that can competently handle this particular exception*

then, the exception is passed up the call stack

- i.e. exception *propagates* up through the method calling hierarchy until it is caught and handled or until it reaches the **`main`** method
- Goes all the way back to **`main`** if necessary
- Program terminates with a stack trace

so an exception is either:

- caught or
- ends up at the JVM who halts the program (default behavior)

throws statement

A **throws**-clause is used when defining a method, to declare

- that it may **throw** an exception
- but the exception is not *caught* in the method

```
public class String {
   public char charAt(int index) throws IndexOutOfBoundsException {
      ...
      throw new IndexOutOfBoundsException();
      ...
      return c;
   }
```

Use a **throws**-clause to:

- "pass the buck" to whatever method calls it
  - i.e. pass the responsibility for the **catch** block to calling the method
  - the calling method can also pass the buck… but eventually some method must catch it
- tells other methods *"If you call me, you must handle any exceptions that I throw."*

Passing the Buck

<u>Good programming practice:</u>

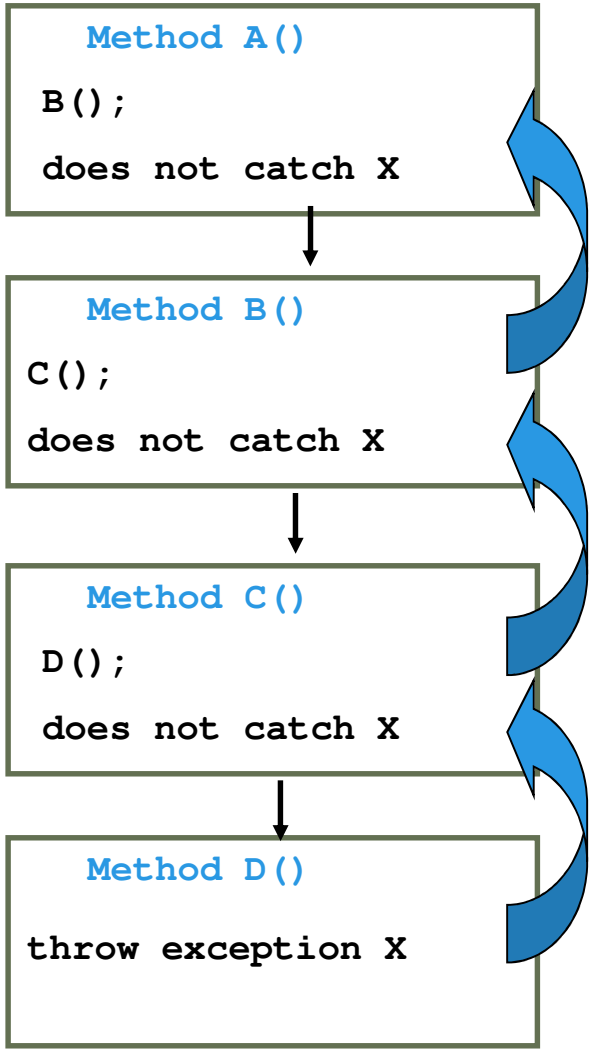
Every exception thrown should eventually be caught in some method

Normally exceptions are either:

- caught in a **`catch`** block or
- *deferred* to the calling method in a **`throws`**-clause (passing the buck)

If a method throws an exception:

- the catch block must be <u>in</u> that method
- unless it is deferred by a **`throws`**-clause
- the calling method can also defers with a **`throws`**-clause, etc., up the line all the way to **`main`**, until a **`catch`** block is found or the program terminates



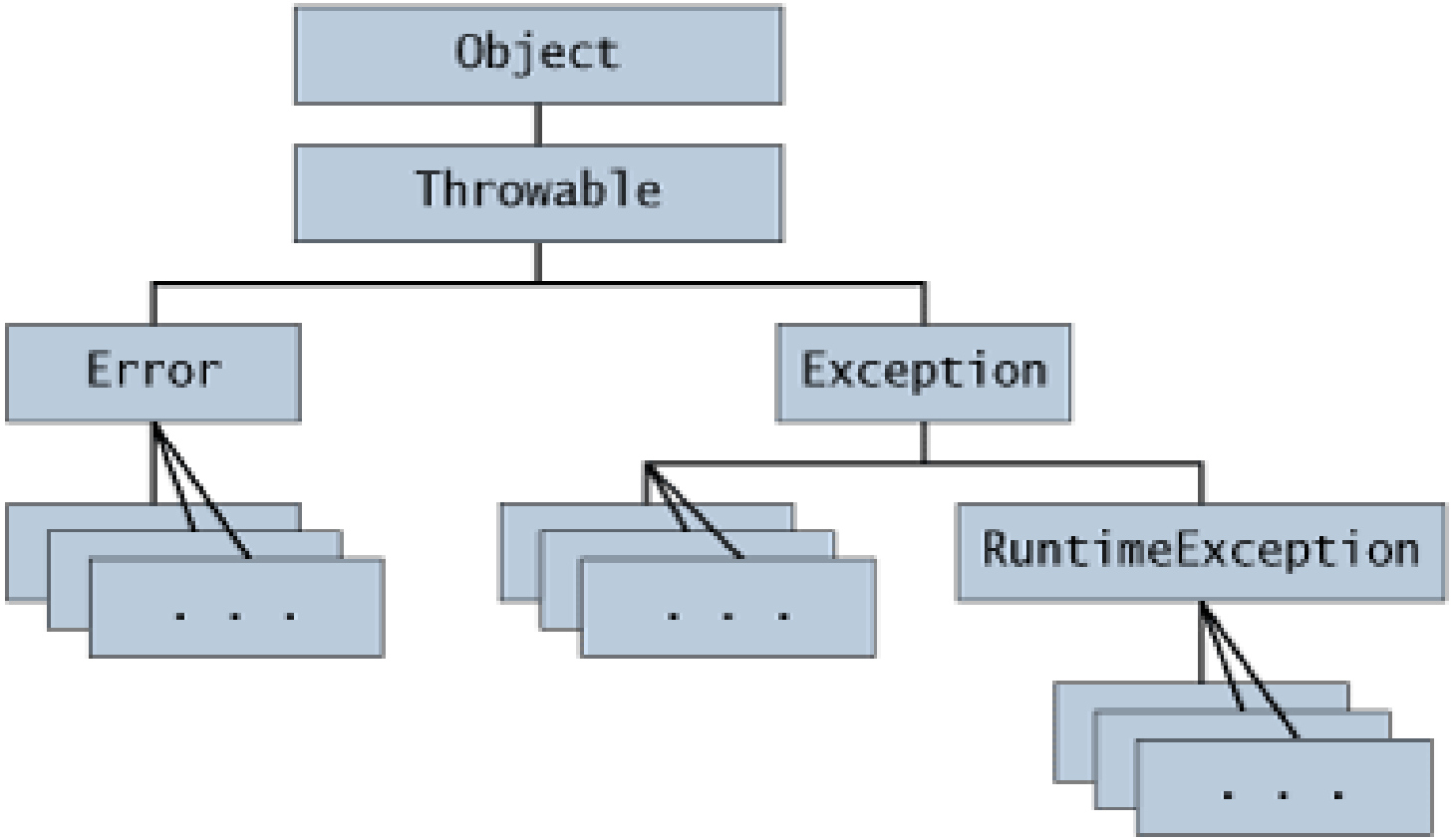
Recap:

You have 2 choices with exceptions

1. Handle the exception locally
   - Use `try/catch` clause

2. Pass the exception up the stack
   - If you can't *competently* handle it
   - Add a `throws` clause to method signature
   - No `catch` clause
   - Let some other ancestor method handle it
   - If no one handles it, program will terminate

The **Throwable** Hierarchy



What's the Difference?

`Error`

- Serious system problem
  - ex: out of memory, linkage error, …

`RunTimeException`

- Programming errors
  - ex: Divide by zero, array index out of bounds, null pointer, …
- You should find them in your testing phase

Defining your own Exception Class

- If you use a predefined, more general exception class
  - --> then your **`catch`**-block could also catch exceptions that should be handled somewhere else.

- A specific **`catch`**-block for your own exception class will catch the exceptions it should and pass others on.

- You can create new exceptions if you need them
  - Usually over-ride the **`Exception`** class or some sub-class

- Can add other methods if you like

Constructors for your exception class

Two basic constructors

```
class myException extends Exception {
        // Default constructor
        public myException() {}

        // Constructor with message
        public myException(String msg){
                super(msg);
        }
}
```

Example

```
public class DivideByZeroException extends Exception
{
    public DivideByZeroException() {
      super("Hey! you're trying to divide by zero!");
    }
    public DivideByZeroException(String message) {
      super(message);
    }
}
```

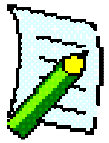
- this code only defines the exception class…
- to use it, you must throw and catch the exception where you need it

Example

```
class InsufficientFundsException extends RuntimeException
{
   public InsufficientFundsException () {}

   public InsufficientFundsException (String message) {
      super(message);
      System.err.println("The bank will contact you.");
  }
}
```

Example: Bank Account



- even better…

Can create your own methods

see **`ExtraStuff.java`**

Checked and Unchecked Exceptions

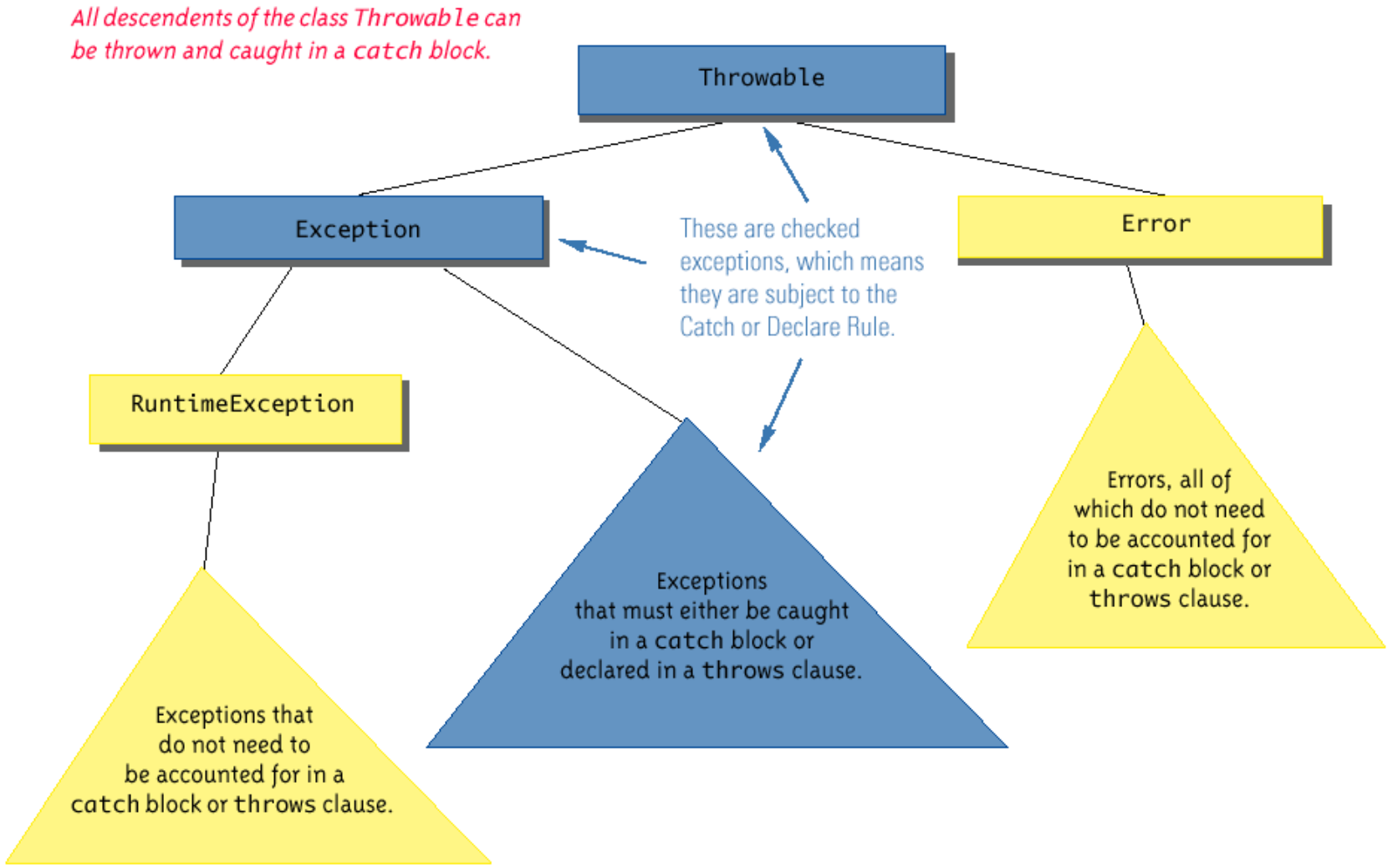
For some kinds of exceptions, the compiler makes tight checks that a **`throw`** clause has a corresponding **`catch`** statement

1. **Checked** exception (checked by compiler)
   - the compiler requires you to deal with them
   - *catch or declare rule:* if a method throws an exception, then it **must handle it**
     - i.e. it must *catch it* or *declare it* in the header with a **`throws`** clause
     - otherwise --> compilation error.

2. **Unchecked** exception (<u>not</u> checked by compiler)
   - the compiler does not require you to deal with them
   - if a method states that it can throw an exception (with a **`throw<u>s</u>`** clause)
     - the calling methods **do not have to** handle the exception
   - a method can **`throw`** an exception even if it is not stated in the header with a **`throws`** clause

The **Throwable** Hierarchy

Display 9.10 Hierarchy of Throwable Objects



Unchecked Exceptions

- Catch or Declare Rule <u>does not</u> apply
- May result from
  - **`RuntimeException`:**
    - can occur very frequently, but are the programmer's fault
      - ex: null pointer, number format exception, index out of bounds, …
    - programmer's choice to deal with them or not
  - **`Error`**:
    - Serious external conditions that are difficult to fix
      - ex: out of Memory, linkage error, …
- You should refrain from throwing these

Checked Exceptions

- Catch or Declare Rule applies

- Typically due to:
  - Unexpected conditions -- often when dealing with I/O
    - ex: file not found, disk error, broken network, …

  - Not programmer's fault
    - likely to occur no matter how careful you are

Example 1: **`NumberFormatException`**

```
int number = Integer.parseInt(someString);
```

- **`parseInt`** can throw a **`NumberFormatException`** **(unchecked)**

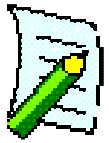
1. so, we can just ignore it…
2. or provide a catch:

```
private void theMethod() {
   try {
      …
      int number = Integer.parseInt(someString);
      …
   }
   catch (NumberFormatException e) {
      System.err.print(e.getMessage());
   …
```

3. or, pass the exception up to the calling method:

```
private void theMethod() throws NumberFormatException {
   …
   int number = Integer.parseInt(someString);
   …
}
```

Example 2: **`ArrayIndexOutOfBounds`**



```
int a[] = new int[10];
for (int n=0; n <=10; n++) {
  a[n] =0;
}
```

- can throw a _____ (checked?)
- so?

Rethrowing Exceptions

You can “rethrow” an exception after catching it and processing it

```
try {
   String text = “text”;
   System.out.println(text.charAt(10));
}
catch(IndexOutOfBoundsException e) {
   System.err.println(“Index out of bounds”);
   e.printStackTrace();
   throw e;
}
```

Useful if current method wants to catch the exception, but the calling method too.

`throws` and overriding

- The **`throws`** clause in the declaration of a method must be compatible with *all* its overridden declarations
  - the overriding method (in the child class)
    - May have **less** Exceptions
    - May have **subclasses** of the Exceptions thrown by the superclass method.

- You cannot add a new checked exception, that was not in the original throws

Throws and overriding

see: **ExceptionsAndOverRiding.java**

```
public class Parent {
 public void someMethod() throws Exception
 {…}
}
public class Child extends Parent {
 public void someMethod() throws java.io.IOException
 {…}
}
```

```
public class GrandChild extends Child {
       public void someMethod()
       {…}
}
public class BadChild extends Child {
       public void someMethod() throws Exception
       {…}
}
```

A Big Example

see:

BankTest.java

BankAccount.java

InsufficientFundsException.java

UnavailableTransactionException.java