Exception Handling

Run-time Errors

- Sometimes when the computer tries to execute a statement something goes wrong:
  - Trying to divide an integer by zero
  - Trying to read a file that doesn't exist
  - Calling a method with improper arguments

- In these cases the instruction would fail
  - We say that a run-time error had occurred

Dealing with Run-time Errors …

```
float inverse(double x){
   return 1.0/x;
   }
```

Dealing with Run-time Errors …

The client needs to be informed of the error

```
float inverse(double x){
   if (x!=0.0) {
      return 1.0/x;}
   else {
      System.out.println
      ("Division by zero");
      return 0;}
```

Dealing with Run-time Errors …

- Faced with a request it cannot and should not fulfill, how should an object react?
  - Print an error message and do nothing
  - Terminate the program (with or without an error message)
  - Silently ignore the request and do nothing
  - Perform the request anyway
  - Fake an appropriate or modified action
  - Request a correction interactively
  - Return the wrong answer
  - Return an error indication

Return an Error Indication

- Every method should report errors to the caller...

In this case, this provides an error return

```
float inverse(double x){
   if (x!=0.0) {
      return 1.0/x;}
   else {
      System.out.println
      ("Division by zero");
      return 0;}
```

Problems with an Error Return

- Complex logic
  - The caller has to check the error and take action, including telling its caller
- Limited information
  - A single error return variable can contain only a limited amount of information
- No returns
  - Constructors can’t return any value

Java Exceptions

- Java's exception-handling mechanism is an attempt to resolve the dilemma faced by an object when it is asked to perform a task that is neither unwilling or unable to perform
  - If a method wants to signal that something went wrong during its execution it throws an exception
- Exceptions are run-time events which indicate an exceptional situation
  - An exception is a failure indication that interrupts (maybe in a non fatal way) the flow of a program
  - The exceptional condition is not usually part of the primary behavior of the program
- There are two kinds of exceptions in JAVA:
  - Implicit (built-in) exceptions which are signals from the Java Virtual Machine to the program indicating a violation of a semantic constraint of the Java language
  - Explicit exceptions which are intended to capture possible errors anticipated by the program designer. To define such an exception, a new exception class may have to be defined

Throwing an Exception

```
static double inverse(double x)
                              throws Exception {
   if (x!=0.0) {return 1/x;}
   else {
    throw new
     Exception("inverse:divide by zero");
   }
 }
```

Throwing an Exception

```
static double inverse(double x)
                              throws Exception {
   if (x!=0.0) {return 1/x;}
   else {
    throw new
     Exception("inverse:divide by zero");
   }
 }
```

Indicates that an Exception object can be thrown back to caller

Throwing an Exception

```
static double inverse(double x)
                              throws Exception {
   if (x!=0.0) {return 1/x;}
   else {
    throw new
     Exception("inverse:divide by zero");
   }
 }
```

Halt's execution of this method
and passes control back to caller

Throwing an Exception

```
static double inverse(double x)
                              throws Exception {
   if (x!=0.0) {return 1/x;}
   else {
    throw new
     Exception("inverse:divide by zero");
   }
 }
```

Creates exception object to be passed back to the caller that encloses information about the occurred problem

More on Exceptional Situations

- Exceptions deal with unusual situations
  - Consider opening, reading, writing, and closing a file
    - GET A FILENAME
      OPEN THE FILE
      IF THERE IS NO ERROR OPENING THE FILE
        READ SOME DATA
        IF THERE IS NO ERROR READING THE DATA
          PROCESS THE DATA
          WRITE THE DATA
          IF THERE IS NO ERROR WRITING THE DATA
            CLOSE THE FILE
            IF THERE IS NO ERROR CLOSING FILE
              RETURN

The Java Exception Programming

- Using exceptions the code looks like this
  - TRY TO DO THESE THINGS:
    GET A FILENAME
    OPEN THE FILE
    READ SOME DATA
    PROCESS THE DATA
    WRITE THE DATA
    CLOSE THE FILE
    RETURN
    IF THERE WAS AN ERROR OPENING THE FILE THEN DO ...
    IF THERE WAS AN ERROR READING THE DATA THEN DO ...
    IF THERE WAS AN ERROR WRITING THE DATA THEN DO ...
    IF THERE WAS AN ERROR CLOSING THE FILE THEN DO ...

Exception Advantages

- Exceptions allow you to
  - Make normal logic of an action clear
  - Decide whether to handle or defer handling an error
  - Handle errors in library code on a custom basis

- To summarize:
  - improved readability
  - easier to modify code
  - more robust code

The Exception Object

- The information about the problem that occurred is enclosed in a real object, the exception object

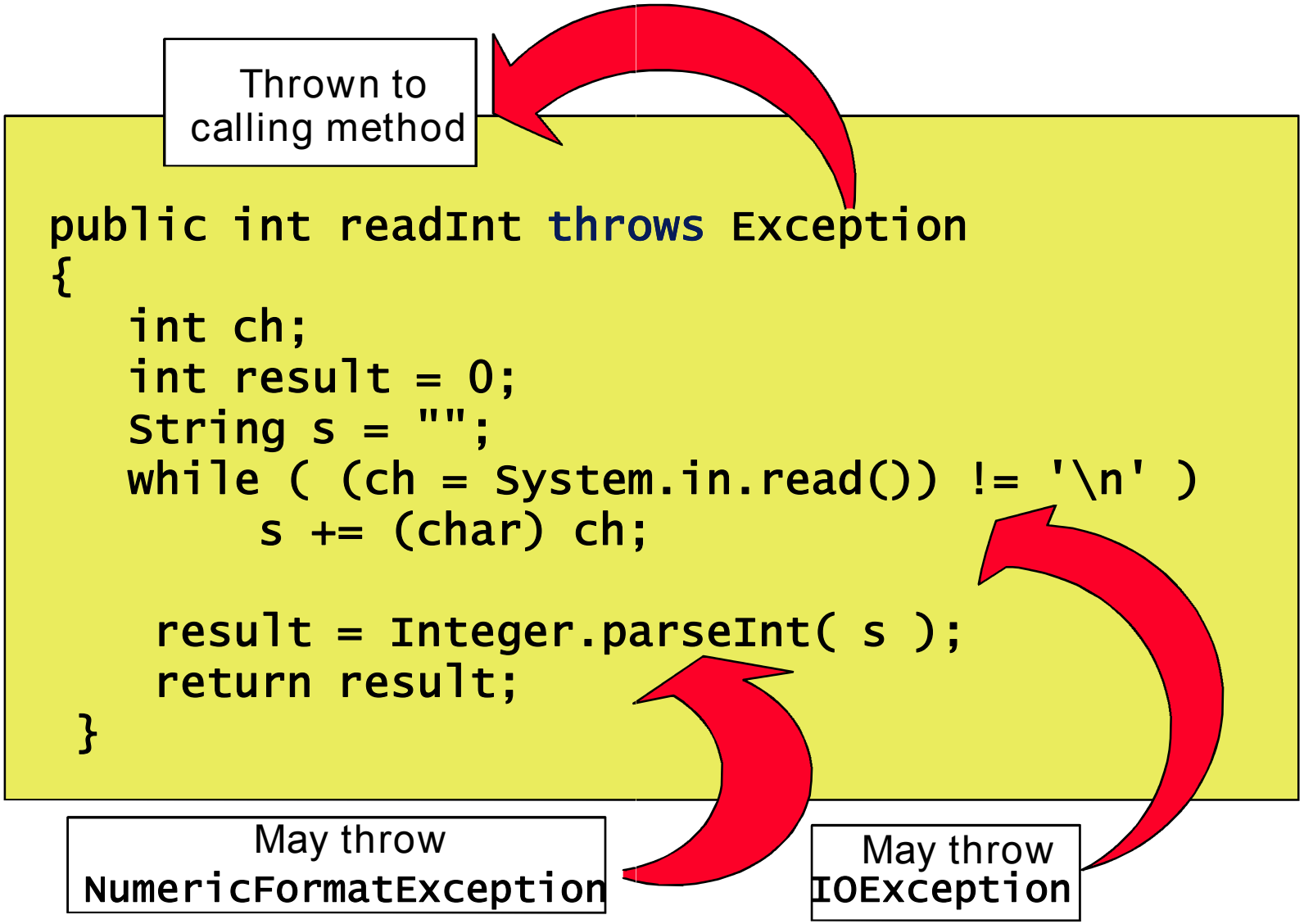
- This information includes:
  - The type of the problem
  - The place in the code where the exception occurred
  - The state of the run-time stack
  - ... other information

The Type of the Exception Object

- The most important information is the type of the exception
  - This is indicated by the class (i.e., type) of the exception object
- The Java API defines classes for many types of exceptions
  - You can define more of your own

- Exception Class Methods
  - `Exception.getMessage()` returns the string passed to its constructor
  - `Exception.printStackTrace()` useful for debugging
  - `Exception.printStackTrace(PrintStream s)`
  - `Exception.toString()` String representation of the object's class and its diagnostic message

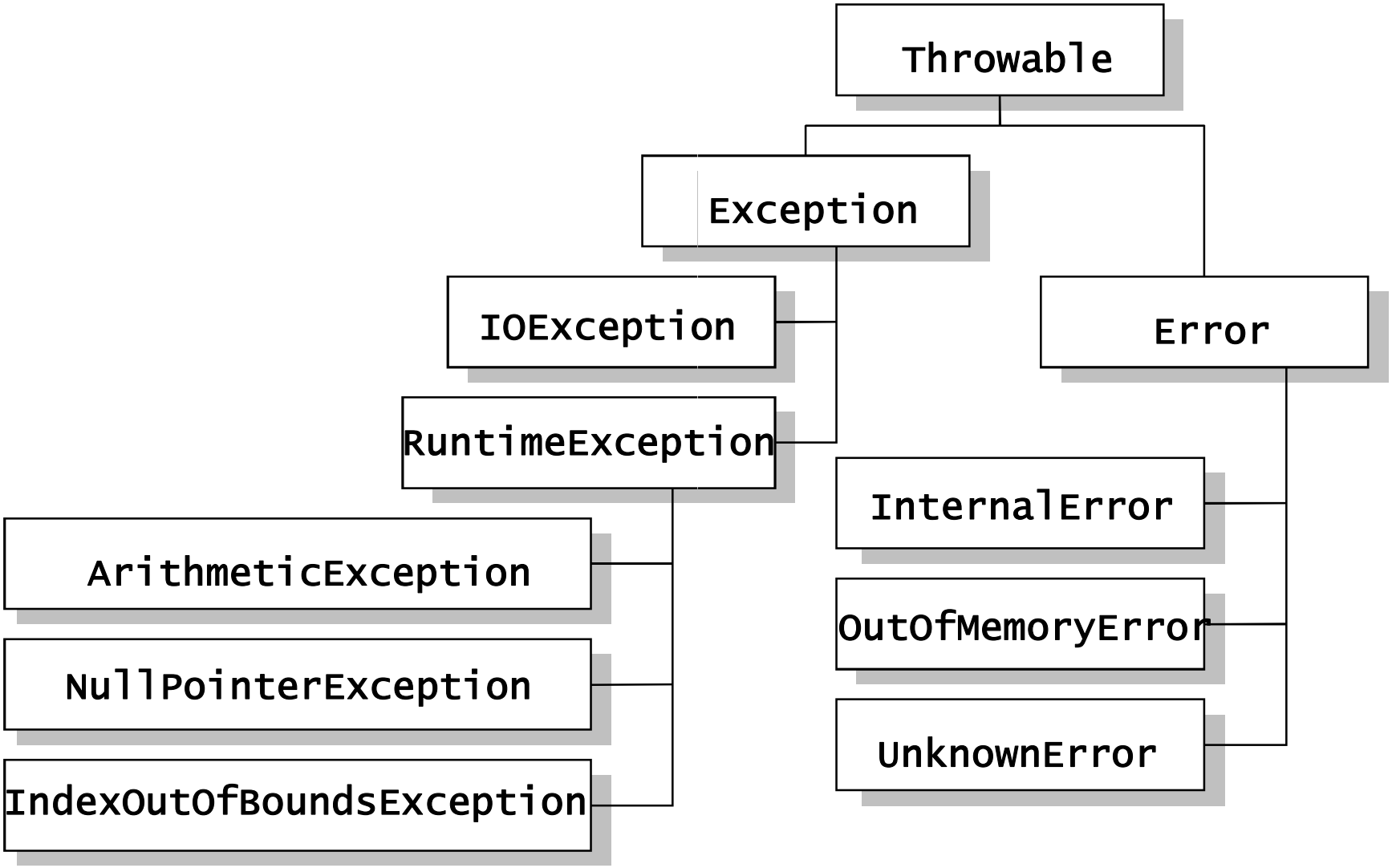
More on Exception Types



Java Common Exception Types

- `ArithmeticException`; thrown when an attempt is made to perform an integer division by zero
- `IndexOutOfBoundsException`; thrown by an array when an out-of-bounds index is used
- `NegativeArraySizeException`; thrown by an array when a negative dimension is given
- `NullPointerException`; thrown by the runtime interpreter when trying to refer to an object through a reference variable whose value is null
- `ClassCastException`; thrown by the runtime interpreter when an inappropriate cast is used on an object reference
- `IllegalArgumentException`; thrown by methods when passing an illegal argument
- `NumberFormatException`; thrown by methods of the numerical wrapper classes when a String does not contain a valid number
- `SecurityException`; thrown by a method performing an illegal operation that is a security violation. This might be trying to read a file on the local machine from an applet
- `IllegalStateException`; thrown by a constructor when environment or application is in incorrect state

Exception Class Hierarchy



java.lang

Class Exception

java.lang.Object

  java.lang.Throwable

    java.lang.Exception

All Implemented Interfaces:

Serializable

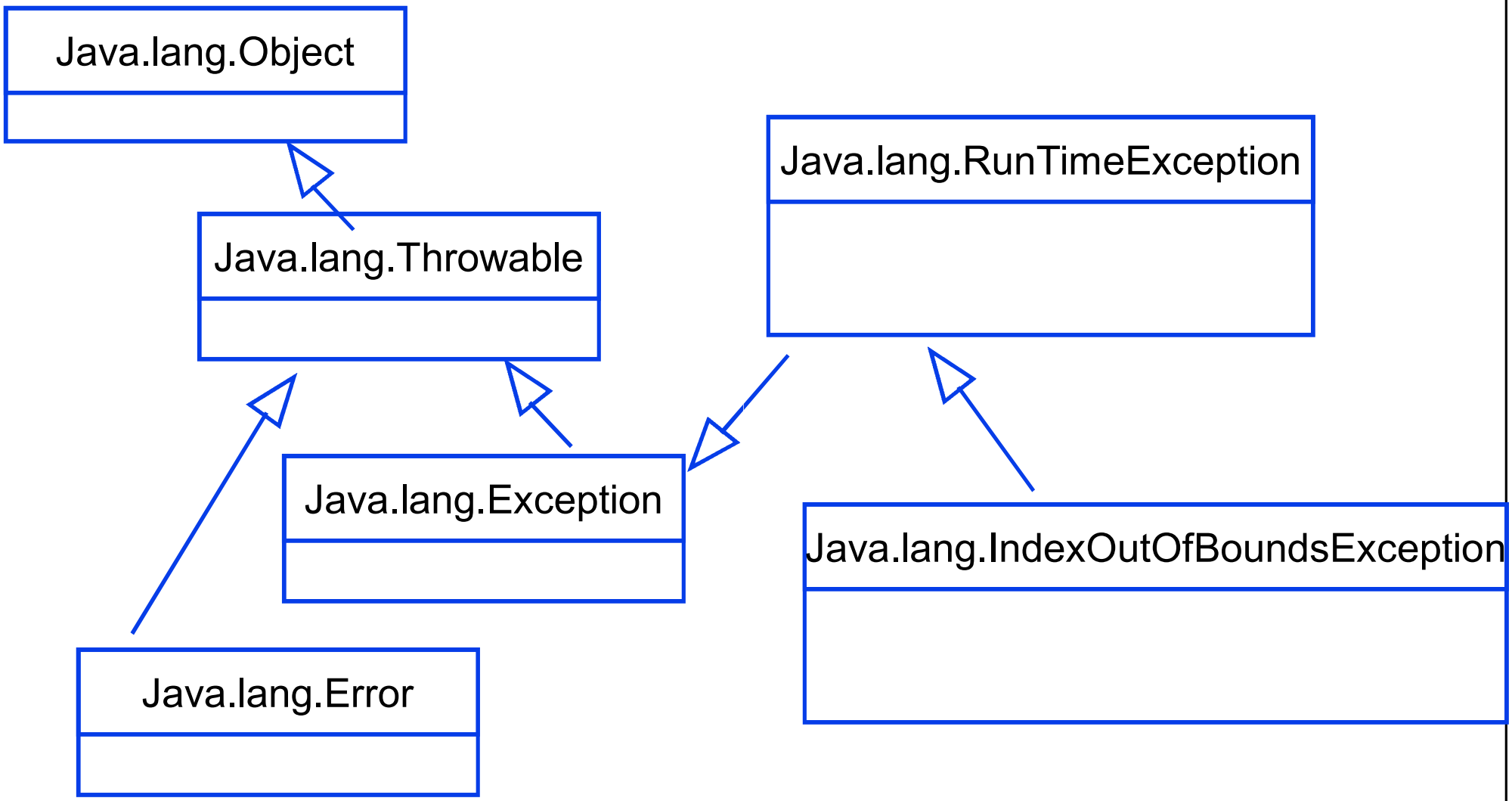
Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvocationTargetException, IOException, JMException, LastOwnerException, LineUnavailableException, MidiUnavailableException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ServerNotActiveException, SQLException, TimeoutException, TooManyListenersException, TransformerException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URISyntaxException, UserException, XAException, XMLParseException, XPathException

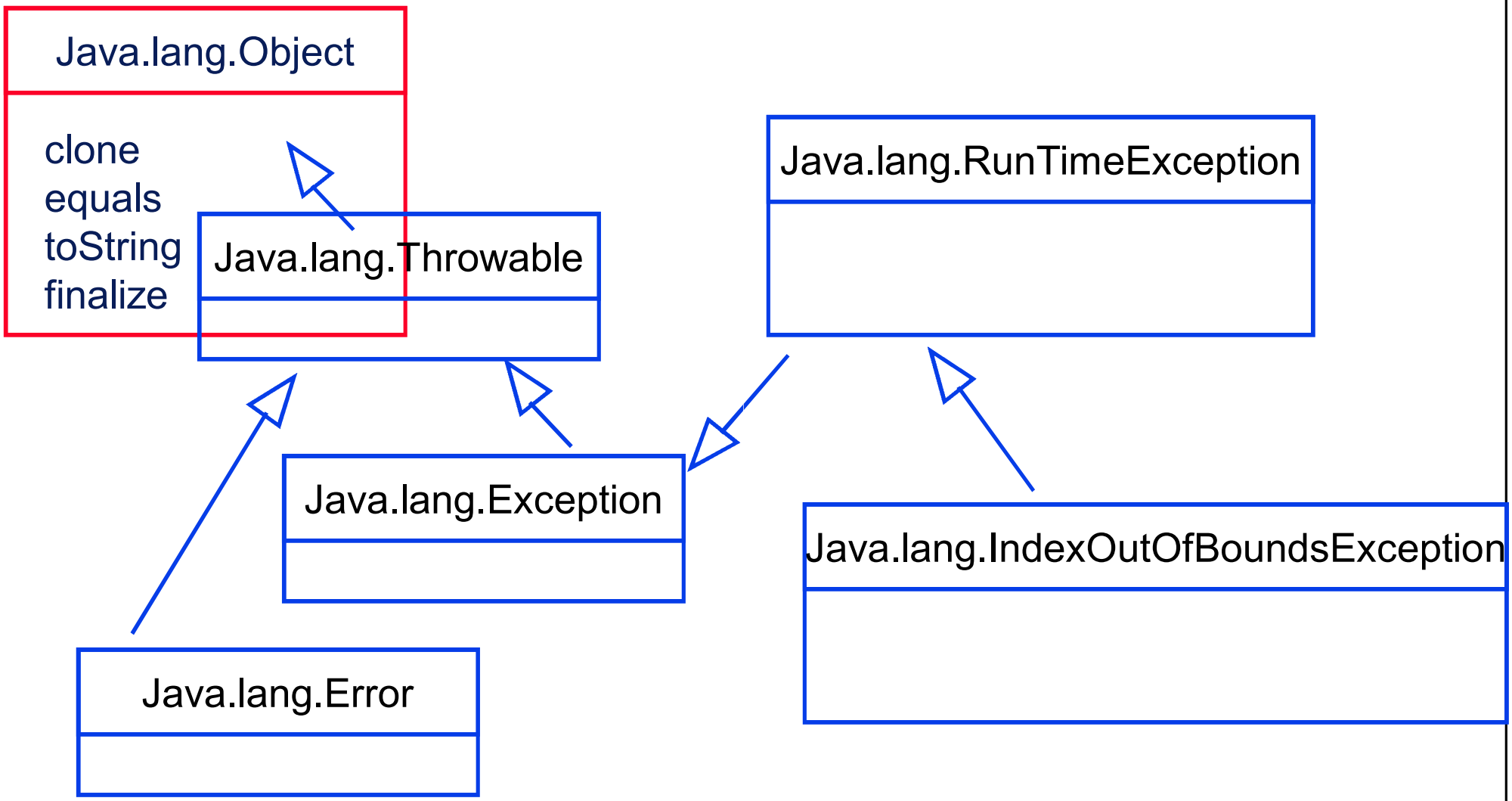
The Exception Class Error

- The **`Error`** hierarchy describes internal errors and resource exhaustion inside the java run-time system
  - You should NOT throw an object of this type
  - If such an error occurs there is little you can do beyond notifying the user and trying to terminate the program gracefully
  - Errors are quite rare
- As programmers, we focus on the **`Exception`** hierarchy, which splits into two branches: exception that derive from **`RunTimeException`**, and those that don't
  - **`RuntimeException`** is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine

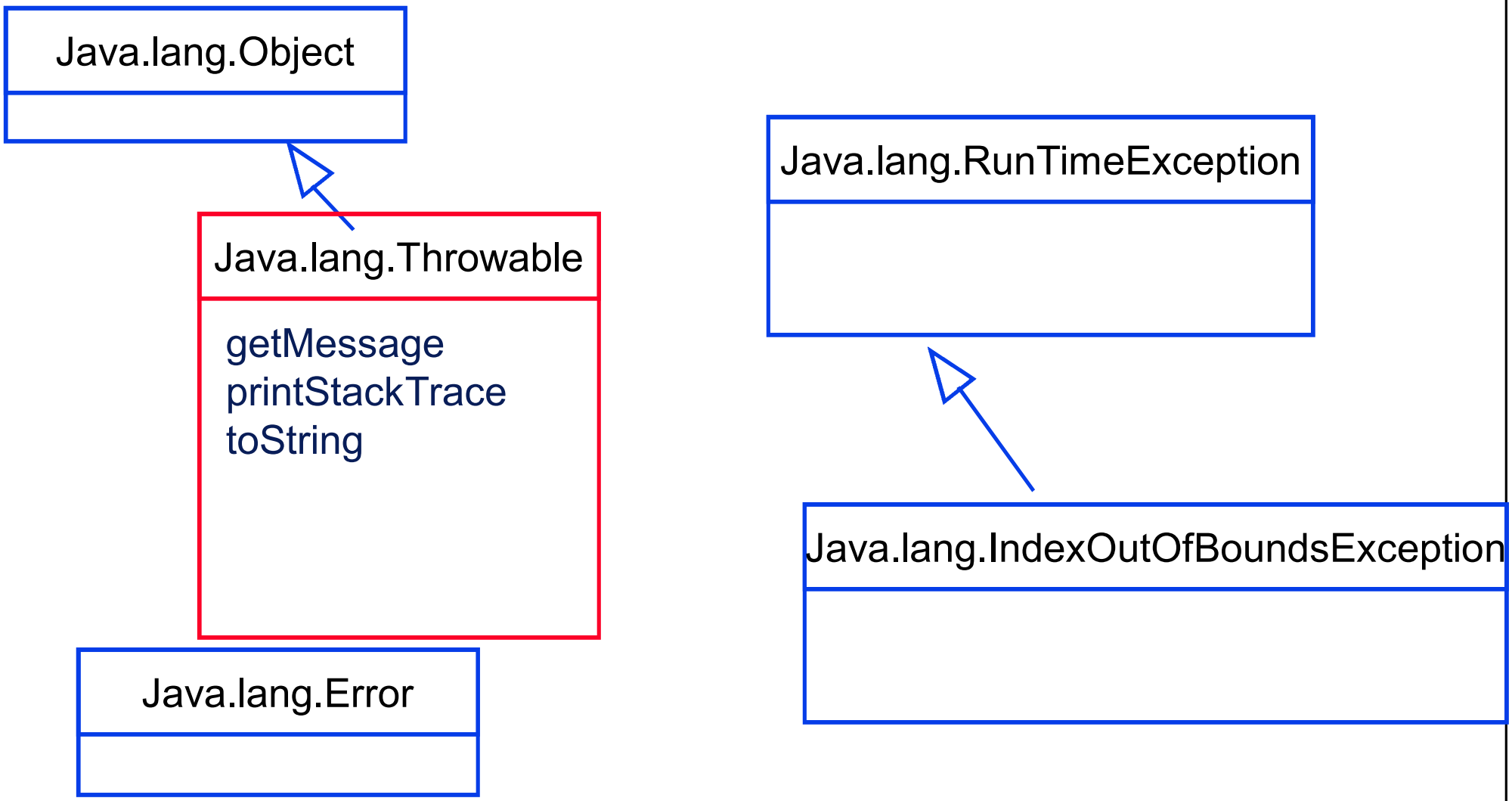
Exception Class Hierarchy



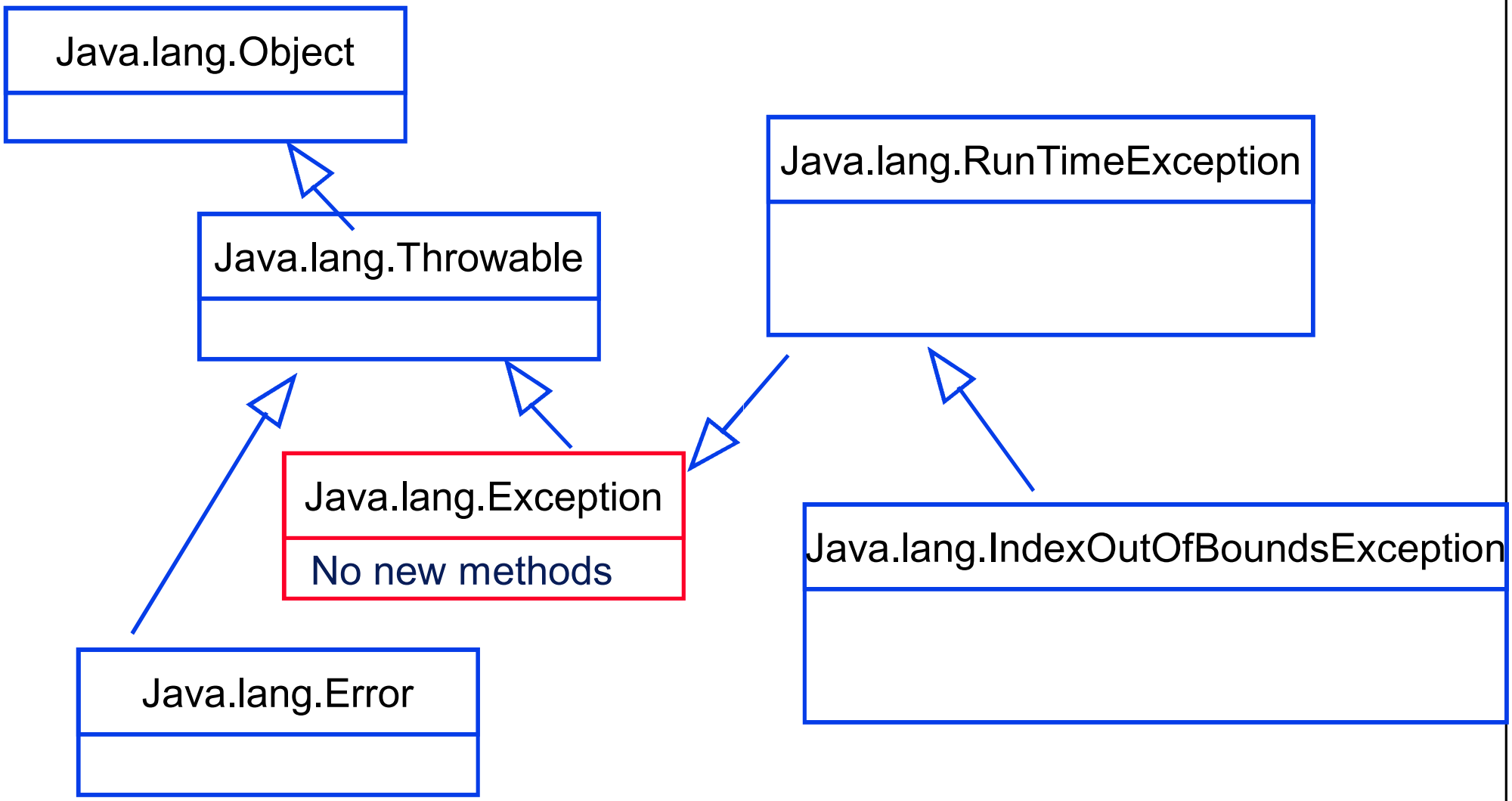
Exception Class Hierarchy



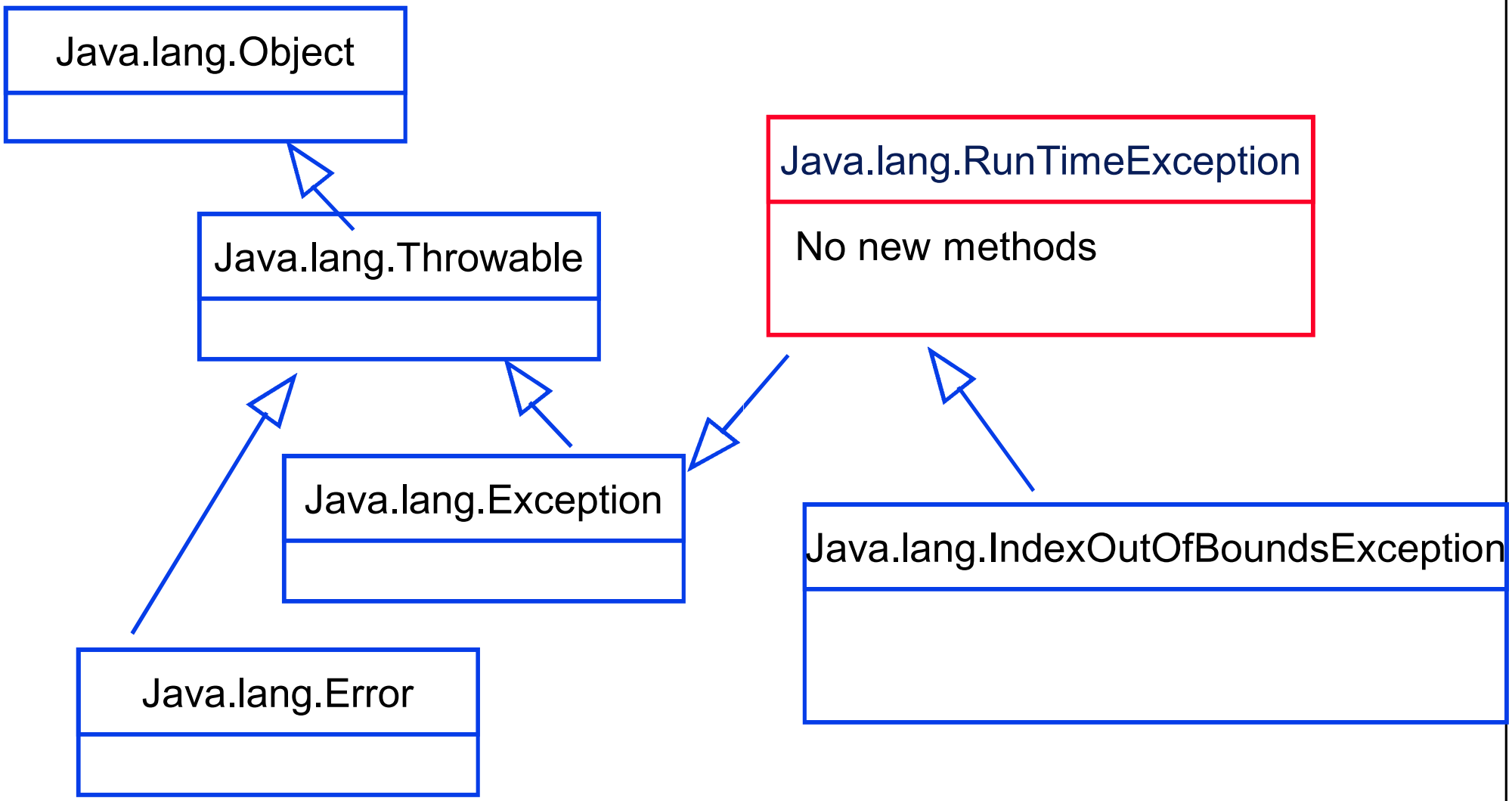
Exception Class Hierarchy



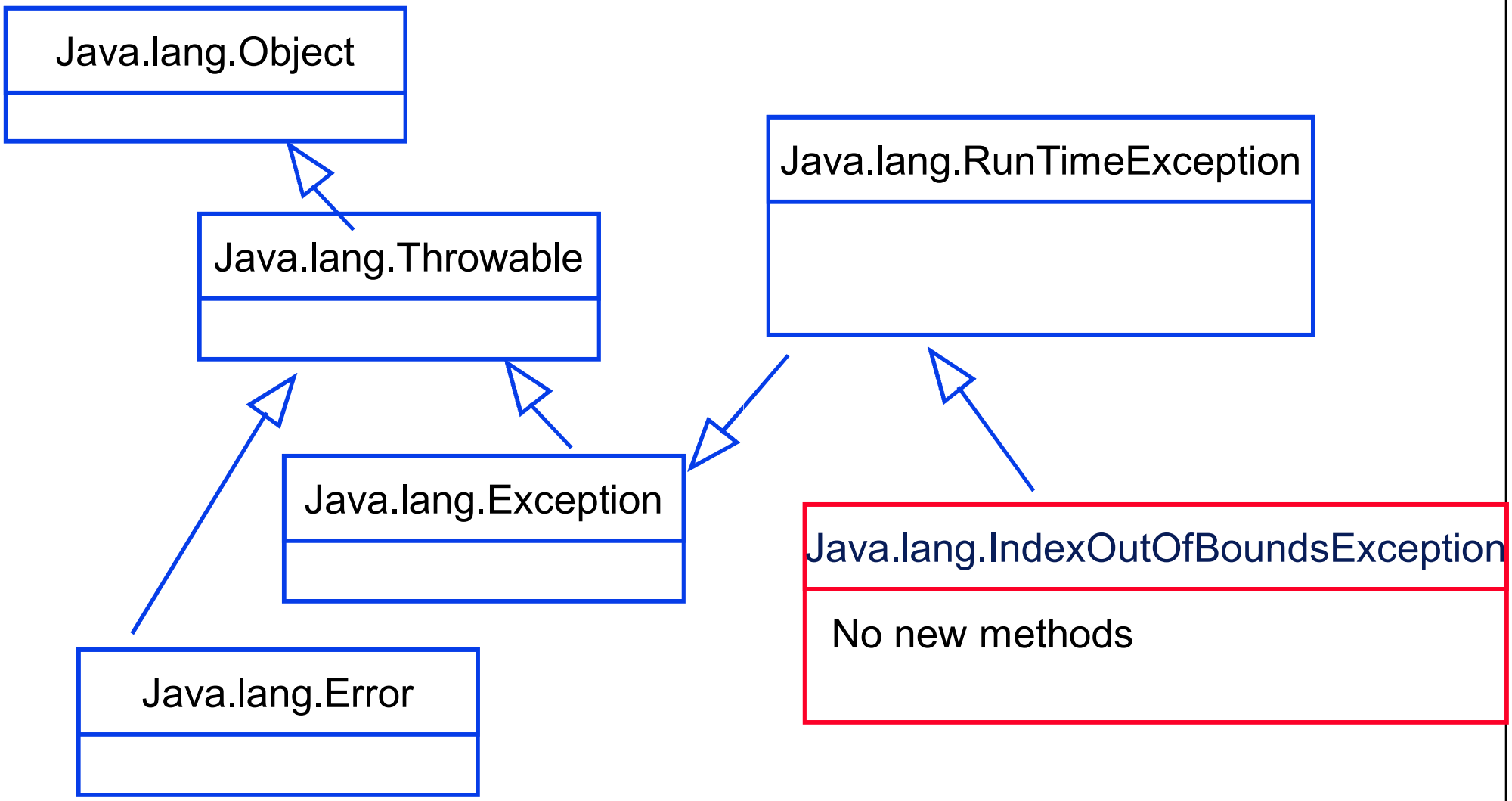
Exception Class Hierarchy



Exception Class Hierarchy



Exception Class Hierarchy



Occurrence of an Exception

- When a program performs an illegal operation the following happens:
  - An exception object is created and thrown
  - The regular flow of the program stops
  - The program may try to handle the exceptional situation
  - If the program ignores the exception the program execution ceases
    - We sometimes say that the program crashes
  - Some exceptions have to be handled by the programmer while others are handled by the default exception handler

Occurrence of an Exception Example

```
class ClockProblem {
    public static void main(String[] args) {
        int hours,minutes,seconds;
        // ...
        Clock myClock = new Clock();
        myClock.setTime(hours,minutes,seconds);
        // ...
    }
}
```

- What happens if hours<0 or seconds>60?

The Root of the Exception

We don't need to declare this exception type

```
public void setTime(int hour, int minute, int second)
  {
  if (hour<0 || hour>59 || minute<0 || minute>59 ||
      second<0 || second>59) {
   throw new IllegalArgumentException("Invalid time");
    }
      this.hour = hour;
      this.minute = minute;
      this.second = second;
}
```

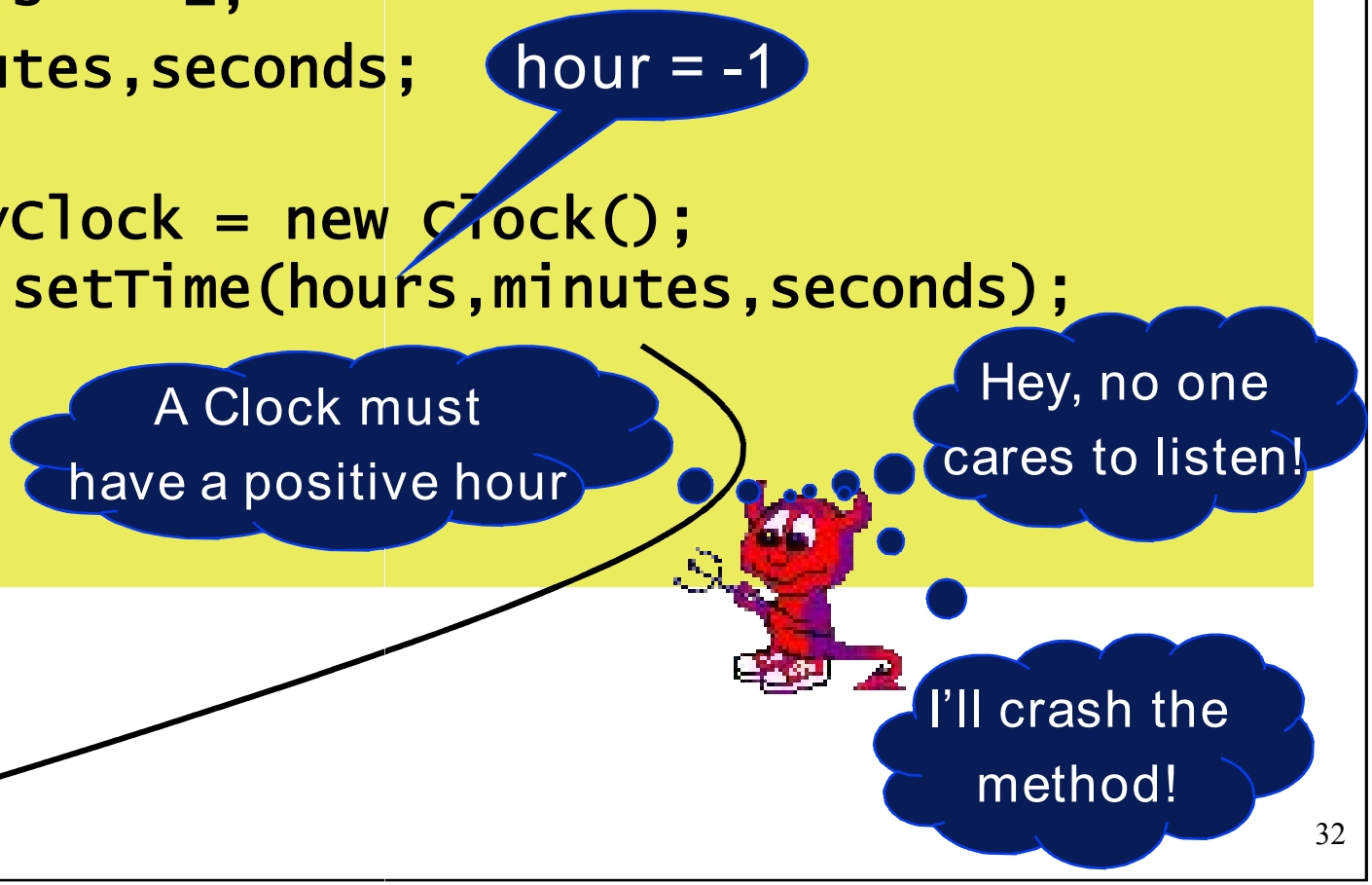
If it was

`new Exception("Invalid time")`

then we would have to

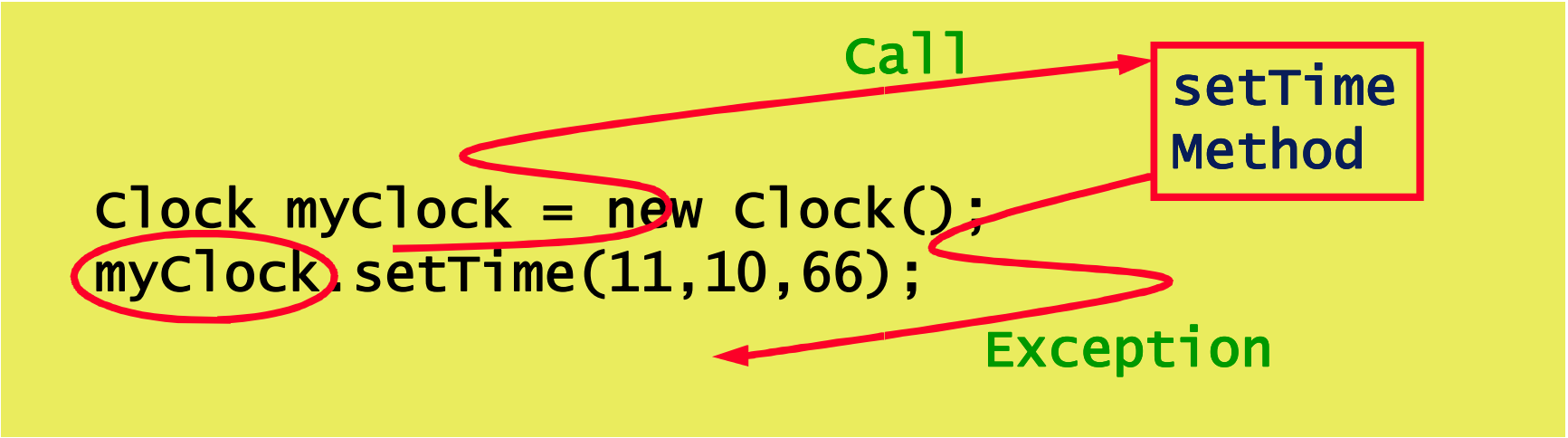
Exception Outcome

```
class ClockProblem {
    public static void main(String[] args) {
        int hours = -1;
        int minutes,seconds;
        // ...
        Clock myClock = new Clock();
        myClock.setTime(hours,minutes,seconds);
        // ...
    }
}
```



Who Receives the Exception?

- The code that invoked the method will receive the exception object



- It can then examine the information it carries and act accordingly

Separating Exception Cases in Your Code

- In the former example we could have checked if the parameters are legal
- Sometimes we cannot prevent an exceptional state: For example when reading from a diskette we cannot tell if the diskette is readable or not without trying to read from it
- Even if we can check in advance if there is a possibility of running into an exceptional case we may want to handle this case outside the main block so as not to complicate the readability of the handling of the regular case

Handling Exceptions

- Java permits an exception to be a non-fatal error by allowing us to anticipate when an exception might be thrown and arrange to catch it
  - An exceptional condition can be intercepted and controlled by using a **`try/catch`** pair of statement blocks
- E.g. an exception occurs when the program executes the **`throw`** keyword
  - The throw statement passes control to a **`catch`** block
  - If there is no **`catch`** block, control (and the exception) is passed back to the caller

```
try{
  statements that might cause an exception
}
catch (ExceptionType e) {
  statements handling the exceptions
}
```

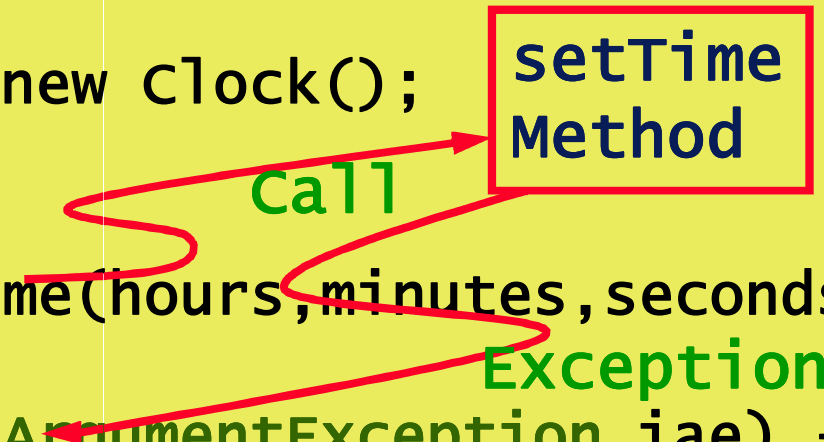
The exception object(s) to be handled

Try.. Catch Block

- The idea is to **`try`** and execute some statements
  - If whatever you tried succeeds, the **`catch`** statements are ignored
  - If an exception is thrown by any of the statements within the **`try`** block, the rest of the statements are skipped and the corresponding **`catch`** block is executed
- Each **`catch`** clause has an associated exception type
- An exception can be caught and handled at one of three places:
  - the method in which the exception was thrown
  - the method that called the method that threw the exception (or a method that called the method that called the method that…)
  - the default exception handler

Handling Exceptions Example

```
class ClockProblem {
    public static void main(String[] args) {
        int hour,minutes,seconds;
        // ...
        Clock myClock = new Clock();
        try {

            myClock.setTime(hours,minutes,seconds);

        } catch (IllegalArgumentException iae) {
          // act accordingly...
        }
        // ...
    }
}
```

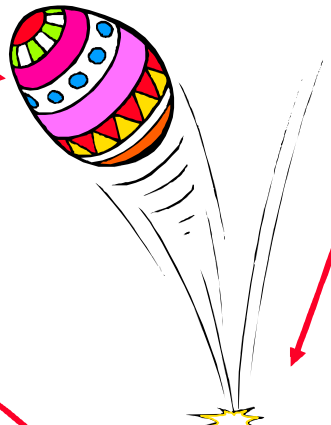


Using Exceptions

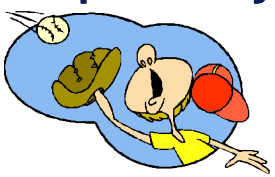
- To use exceptions you have to do two things
  - Put the code that may cause an exceptional situation into a `try` block
  - Provide one or more `catch` blocks immediately following the `try` block
    - You should provide `catch` blocks for all errors you want to handle. Do this by choosing to catch particular exception types
    - If you don't provide a `catch` block for a particular exception, then the exception will be propagated
- An Alternative:
  - You don't have to use `try...catch`
  - Instead, you can defer to the caller of your method
  - Add `throws` Exception clause to method definition
  - Means that caller must use try catch or throw the exception as well

Throwing in one method, and catching in another

ExceptionType1 object

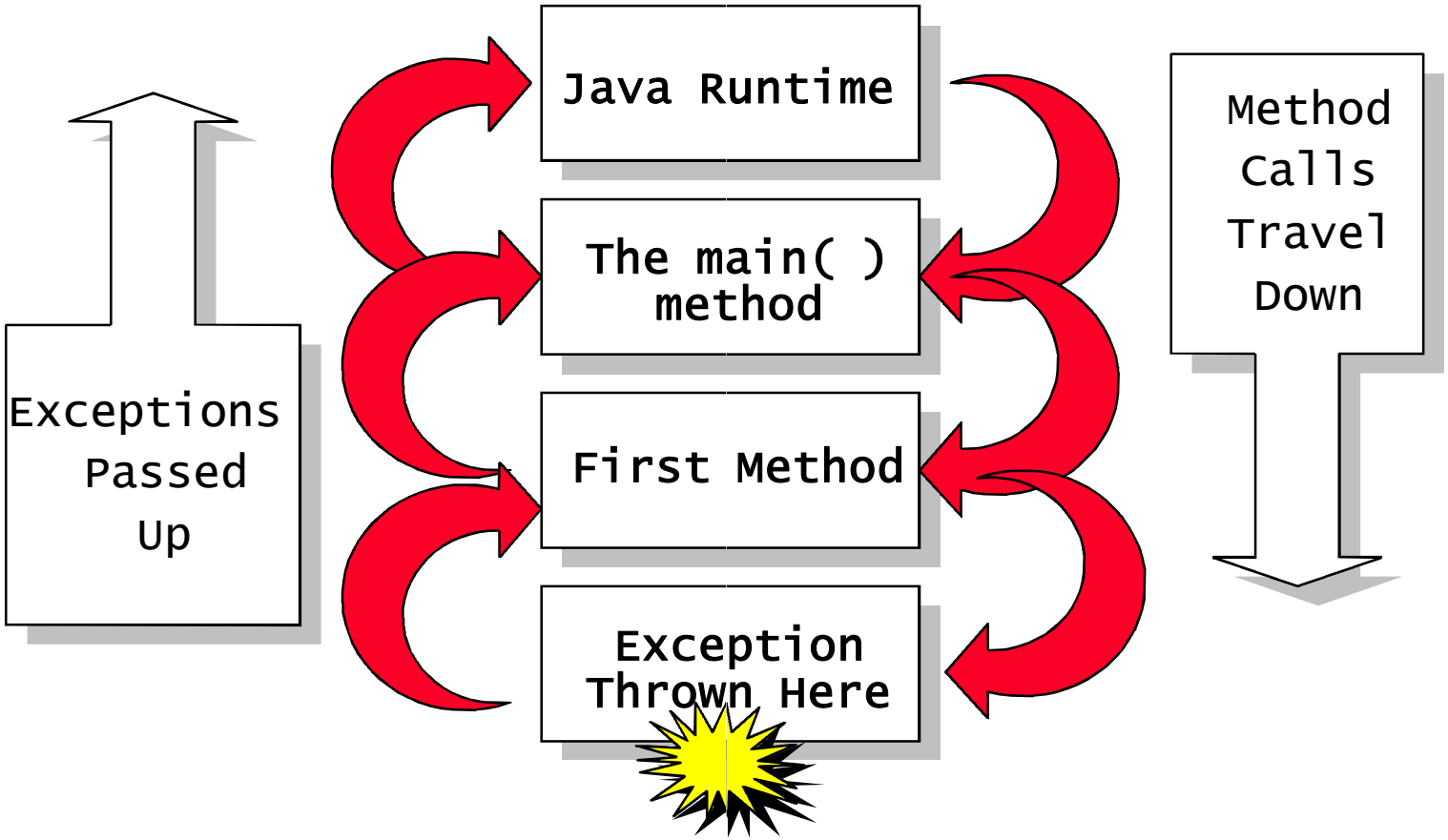


```
… method1() throws ExceptionType1{
    …
}

… method2() throws ExceptionType1{
    …
    calls method1();
    …
}

… method3() throws ExceptionType1{
    …
    calls method2();
    …
}

… methodn(){
    try{
      calls methodn-1();
    }catch(ExceptionType1 se){
        …
    }
    …
}
```



first method in the call chain that catches ExceptionType1 objects

Exception Propagation



Exception Propagation Example

```
public class C1{
 public void method1() throws Exception1, Exception2{
  try{
   System.out.println("m1_1");
   //some code here that will randomly throw Exception1,2,or3
   System.out.println("m1_2");
  }catch(Exception3 e3){
   System.out.println("m1_3");
  }
  System.out.println("m1_4");
}
public void method2() throws Exception1{
 try{
   method1();
   System.out.println("m2_1");
 }catch(Exception2 e2){
   System.out.println("m2_2");
 }
 System.out.println("m2_3");
}
```

Exception Propagation Example

```
public void method3(){
 try{
   method2();
   System.out.println("m3_1");
 }catch(Exception1 e1){
   System.out.println("m3_2");
 }
 System.out.prinltn("m3_3");
}

public class Tester{
  public static void main(…){
    C1 c = new C1();
    c.method3();
  }
}
```

If an exception is thrown by the method that you are calling, you HAVE TO deal with this exception by catching it
OR
throwing it

Exception Propagation Example

- Scenario 1: What happens if `Exception3` is thrown?

```
m1_1
m1_3
m1_4
m2_1
m2_3
m3_1
m3_3
```

- Scenario 2: What happens if `Exception2` is thrown?

```
m1_1
m2_2
m2_3
m3_1
m3_3
```

- Scenario 3: What happens if `Exception1` is thrown?

```
m1_1
m3_2
m3_3
```

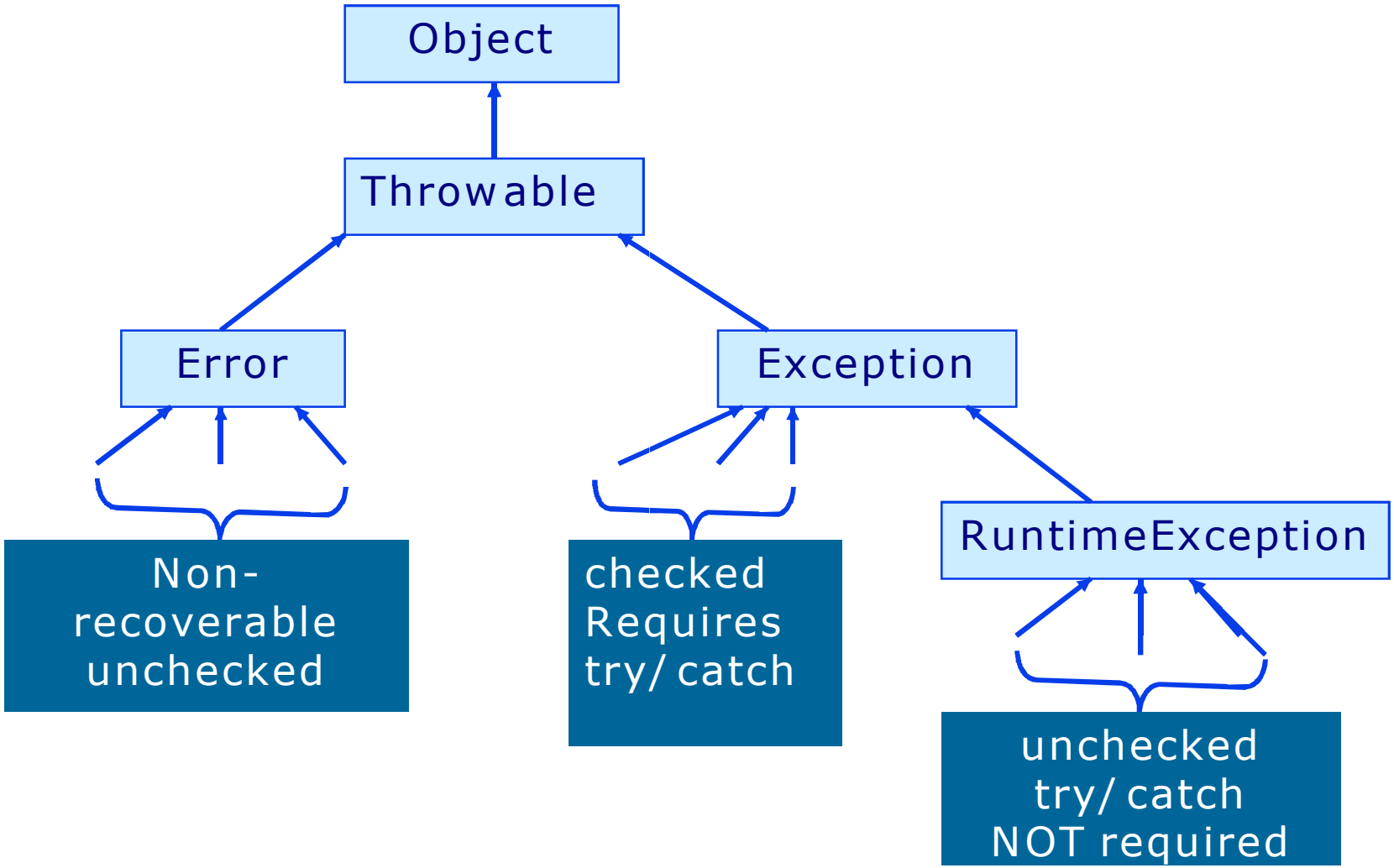
When to Advertise an Exception?

- An exception is thrown when :
  - You call a method that throws an exception
  - you detect an error and throw an exception with the **`throw`** statement
  - You make a programming error, such as a[-1] = 0
  - An internal error occurs
- You need to advertise only in the first two scenarios
  - A method indicates that it might throw one or several exceptions by including them as part of its header using a **`throws`** clause
- Let's say method A is declared to throw exception E. If method B calls A, then B must either:
  - Catch exception E
  - Be declared to throw E as well

Checked and Unchecked Exceptions

- Java divides exceptions into two categories: checked and unchecked
  - **`RuntimeException, Error`** (and their subclasses) are unchecked exceptions
  - **`Exception`** (and its subclasses except **`RuntimeException`**) are checked exceptions
- Main differences:
  - If a method might throw a checked (unchecked) exception, Java (do not) requires that the exception be listed in the methods' header; otherwise, there will be a compile-time error
  - If a client code calls a procedure that might throw a checked (unchecked) exception, Java (do not) requires that the client code handle the exception; otherwise, there will be a compile-time error
- Including a throws descriptor for both kinds of exceptions, especially if you throw it yourself in your code, is a good idea!
  - You can't completely understand how to use a method without this information about exceptions

Exception Object Hierarchy



Dealing with Your Own Exceptions

- Your code may run into problem that is not adequately described by any of the standard exceptions
  - To recognize the exceptional state, you may use standard if-else logic
  - To respond to them, you can simply create your own exception class

```
class FileFormatException extends IOException {
   public FileFormatException() {}
   public FileFormatException(String description){
     super(description);
   }
}
```

  - Just throw your own exception when needed

```
FileFormatException f = new FileFormatException();
throw f;
```

- New checked exceptions will be subclasses of **`Exception`** while new unchecked exceptions will be subclasses of the **`RuntimeException`**

Creating & Throwing Your Own Exceptions

```
public class EmployeeNotFoundException extends Exception {
  public
      EmployeeNotFoundException(String name) {
      super("Employee " + name + " is not found in company");
   }
}
public class CompanyE {
  private Vector names; // more...
  public void pay(String employeeName)
                        throws EmployeeNotFoundException {
      int index = names.indexOf(employeeName);
      if (index == -1)
         throw new EmployeeNotFoundException(employeeName);
      Employee e = getEmployee(index);
      e.pay();
     } // more...
   }
```

Coping with Unchecked Exceptions

- Any call can potentially throw any unchecked exception
  - unchecked exceptions will be implicitly propagated to the caller even if they aren't listed in the header; this means that methods can raise unchecked exceptions even when this isn't mentioned in their header
- We may have a problem in catching unchecked exceptions because it's hard to know where they come from
  - The only way to be certain about the origin of an unchecked exception is to narrow the scope of the try block
- Choosing between unchecked and checked exceptions should be based on expectations about how the exception will be used
  - If you except using code to avoid calls that raise the exception should be unchecked; otherwise, exceptions should be checked
- No need to remember which exceptions are checked and which are unchecked – the compiler will tell you if a checked exception has not been handled

Exceptions and Contract-based Programming

- Make both client and supplier responsibilities explicit as part of documentation: Like a business contract (house-building)
  - The module's client is the caller of the module
  - The supplier is the called module
- Both client and supplier know what is expected of them and what to expect of the other
  - The pre-condition states the responsibilities of the client
  - The post-condition states the guarantees made by the supplier
- Contract-based Programming:
  - Pre-Post conditions: Rights and Obligations
  - Contract Violations: Exceptions

The ADT Queue

- Characteristics: A queue Q stores items of some type, in First-In, First-Out order (FIFO)
- Operations:
  - `Create()` returns an empty Queue
  - `Enqueue(x)` item x is added to the rear of the queue (also called insert)
  - `Dequeue()` the item at the front of the queue is removed (also called remove)
  - `Front()` returns the least-recently enqueued item without removing it
  - `Size()` returns the number of queue items
  - `IsEmpty()` returns true if and only if the queue contains no items
- Possible Implementations:
  - Circular Array
  - Linked List

The ADT Queue Algebraic Specification

- Syntax:

```
create()                          ->Queue
enqueue(q,o)  Queue x Object      ->Queue
front(q)      Queue               ->Object
dequeue(q)    Queue               ->Queue
size(q)       Queue               ->int
isEmpty(q)    Queue               ->boolean
```

- Axioms

```
isEmpty(q)           = (size(q) == 0)
size(create())       = 0
size(enqueue(q,o)) = size(q) + 1
size(dequeue(q)) = if isEmpty(q) then 0 else size(q)-1
front(create())      = error
front(enqueue(q,o)) = if isEmpty(q) then o else front(q)
dequeue(create())    = error
dequeue(enqueue(q, o))= if isEmpty(q) then create()
                        else enqueue(dequeue(q), o)
```

The ADT Queue Contract

```
ADT Queue {
  public create() { ...
     // require:  always valid
     // ensure:   isEmpty()
     // ensure:   size() = 0
     }
  public Queue enqueue(Queue q, Object o) { ...
     // require:  always valid
     // ensure:   not isEmpty()
     // ensure:   size() = size(q) + 1
     }
  public Object front(Queue q) {...
     // require:  not isEmpty()
     // ensure:   front(enqueue(q,o)) = if isEmpty(q) then o
     //                                 else front(q)
     }
```

The ADT Queue Contract

```
public Queue dequeue(Queue q) {...
    // require:  not isEmpty()
    // ensure:  size() = size(q) - 1
    // ensure:  dequeue(enqueue(q,o)) = if isEmpty(q) then
    //                    create() else enqueue(dequeue(q),o)
 }
 public boolean isempty(Queue q) {...
    // require: always valid
    // ensure:  no state change
 }
 public boolean size(Queue q) {...
    // require: always valid
    // ensure:  no state change
 }
// class invariants:
//  size() >= 0
//  isEmpty() = (size() == 0)
}
```

From ADT Specs to Pre-Post Conditions

Constructors	(…x … -> *T*)
Accessors	(…x *T* x … -> …)
Transformers	(…x *T* x … -> *T*)

- A precondition for a specs' operation reappears as a precondition for the corresponding method
- Axioms involving a transformer (possibly with selectors) reappear as postconditions of the corresponding method
- Axioms involving only accessors reappear as postconditions of the corresponding methods or as clauses of the class invariant
- Axioms involving a constructor reappear in the postcondition of the corresponding constructor method

Exceptions of the ADT Queue Contract

- Given an empty queue, where a client tries to dequeue an item . . .
  - What should you have the Queue do?
  - How do you know what it should do?
  - Should it print out a message?
  - Should it try to decrease the Queue's size?
  - Should it not dequeue the item?
- What should you do?
  - Take benefit of the semantics of ADT Queue Algebraic Specification: if the precondition does not hold, the ADT is not required to provide anything!
  - Use exceptions to inform about run-time assertion violations (i.e., pre, post and invariant conditions)
- A run-time assertion violation is a manifestation of a bug in the software
  - Precondition violation : Bug in the client
  - Postcondition violation : Bug in the supplier

Failure and Exceptions

- A method call fails if it terminates its execution in a state that does not satisfy the method's contract
  - Violates postcondition or class invariant
  - Calls a method whose precondition is violated
  - Causes an abnormal OS signal such as memory exhaustion etc.
- An exception is a run-time event that may cause a method call to fail
  - Every failure results from an exception, but not every exception results in a failure (if the method can recover from it)
- Rescue Clause
  - Retry: Attempt to change the conditions that led to the exception and execute the routine again from the start (possibly, exploring alternatives to favor software fault tolerance)

```
{ true } Retry_Body { Inv and Pre }
```

  - Failure: Clean-up the environment (restore the invariant), terminate the call and report failure to the caller

```
{ true } Rescue_Body { Inv }
```

Propagating Exceptions

- Exceptions are propagated up the call chain (dynamic)
- A supplier code can define an exception and raise it, to enable context-sensitive handling of the exception by the various clients
  - Checked exceptions contribute to robustness by forcing the client to process exception explicitly, or propagate it explicitly

Exceptions: Propagation

- When an exception is thrown, it must be caught immediately or declared to be allowed to propagate
- An example of code that will not compile:

```
class Queue {
  ...
  public boolean isEmpty( ) {
  ...
  } // isEmpty

  public void dequeue(Object o)                {
     if (isEmpty( ) == true)
         throw new QueueEmptyException( );
      ...
  } // dequeue
  ...
} // class Queue
```

Dequeue is not allowed to do this

Exceptions: Propagation

- Results in an error saying that dequeue must catch or declare `QueueEmptyException`
- To resolve this, modify dequeue so that the exception is declared to be thrown:

```
public void dequeue(Object o) throws QueueEmptyException {
    if (isEmpty( ) == true)
        throw new QueueEmptyException( );
        ...
  }
```

- The method header above declares that this method can throw a `QueueEmptyException`
  and that the method calling `dequeue( )` must plan on catching the `QueueEmptyException`

Dealing with Customer Exceptions

- Suppose you want to use this Queue class to simulate a line of Customers, and you do:

```
class Customer {
   ...
} // Customer
class Cashier  {
   Queue customerQueue = new Queue( );
   ...
   public void getNextCustomer( ) {
      Customer cust;
      ...
      cust = (Customer) customerQueue.dequeue( );
      ...
   } // getNextCustomer
   ...
} // Cashier
```

Dealing with Customer Exceptions

```
class QueueEmptyException extends Exception
   {
     public QueueEmptyException( ) { }
     public QueueEmptyException(String strMessage)
      {super(strMessage); }
 }
```

- This will result in a compile-time error because method `getNextCustomer` must:
  - catch exception `QueueEmptyException`, or
  - declare that `QueueEmptyException` propagates upwards
- Thus, we can repair `getNextCustomer` in one of two ways:
  - Option 1: have it catch exception `QueueEmptyException`
  - Option 2: have it declare that this method allows `QueueEmptyException` to propagate

An Exception: Catching It

```
public void getNextCustomer( )
   {
      Customer cust;

      try  {
         ...
         cust = (Customer) customerQueue.dequeue( );
         ...
      }  // try

      catch (QueueEmptyException e)   {
          // handle the QueueEmptyException here
          ...
      } // catch

   } // getNextCustomer
```

An Exception: Declare that it will propagate

```
public void getNextCustomer( )
                              throws QueueEmptyException
   {
      Customer cust;
      ...
      cust = (Customer) customerQueue.dequeue( );
      ...
   } // getNextCustomer
```

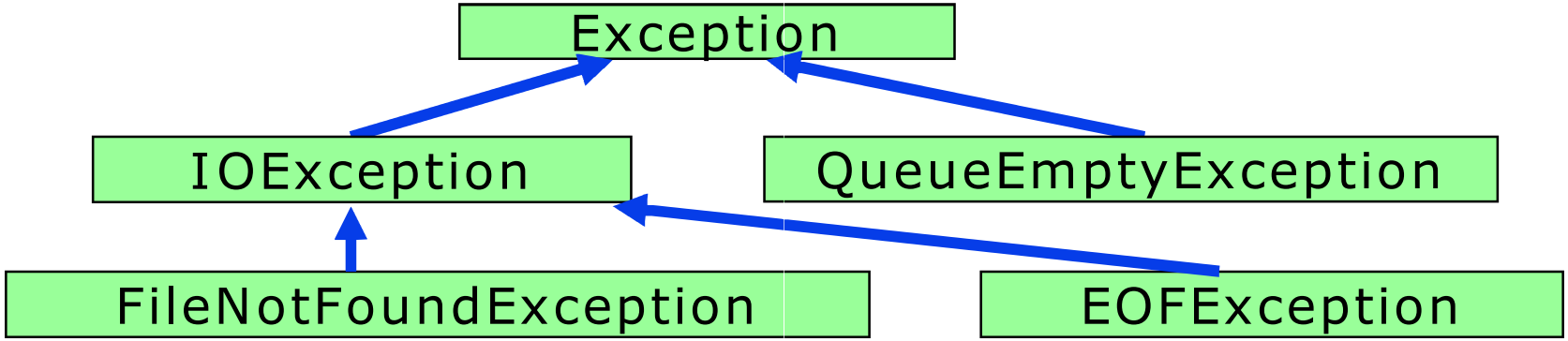
- This option dictates that whoever calls method `getNextCustomer( )` has the responsibility of handling the exception

Disciplined Exception Handling

- An error is the presence in the software of some element not satisfying its specification (wrong design decision)
- An exception is the occurrence of an abnormal condition during the execution of a software element
- A failure is the inability of a software element to satisfy its purpose
- There are only two reasonable ways to react to an exception:
  1. clean up the environment and report failure to the client ("organized panic ")
  2. attempt to change the conditions that led to failure and retry
- It is not acceptable to return control to the client without special notification
- When should an object throw an exception?
  - If and only if an assertion is violated
- If it is not possible to run your program without raising an exception, then you are abusing the exception- handling mechanism!

Exceptions and Inheritance

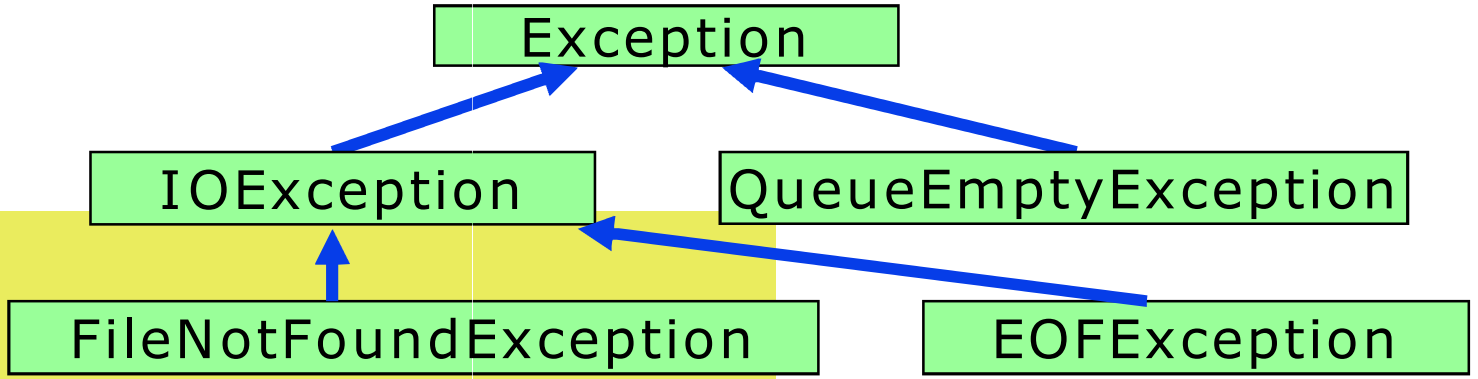
- If a method raises different exceptions it would be nice to be able to differentiate between them
  - We can do this in Java using inheritance
- `Exception` is a class just like other classes in the Java system
- Suppose we had an inheritance hierarchy of exceptions like this:



- You can have more than one catch block for a try block
- A catch block will only catch that exception or a subclass of that exception

Exceptions and Inheritance

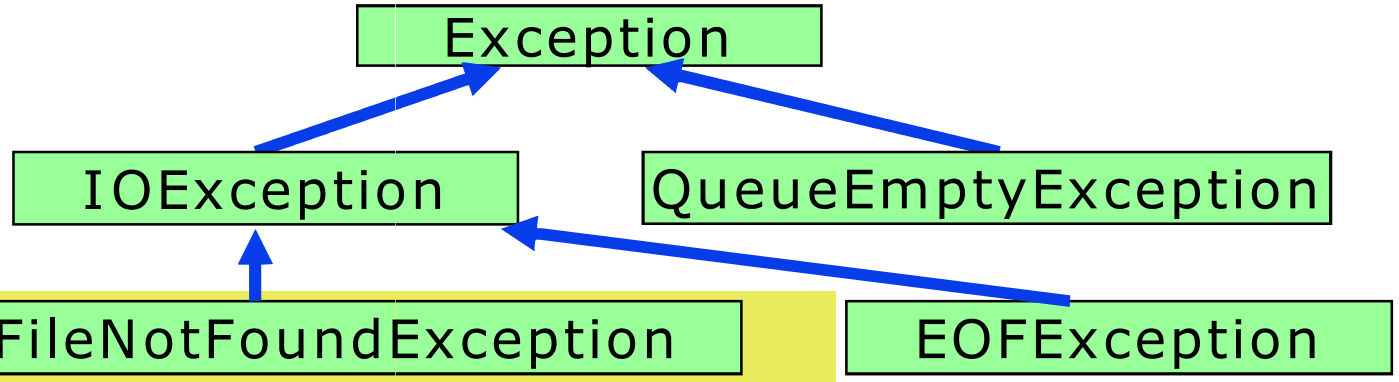
- This sequence of catches works:




```
try    {
  ...
  } // try
catch (QueueEmptyException e) {
  ...
  } // catch QueueEmptyException
catch (FileNotFoundException e) {
  ...
  } // catch FileNotFound Exception
catch (IOException e) {
  ...
  } // catch IO Exception
```

Exceptions and Inheritance

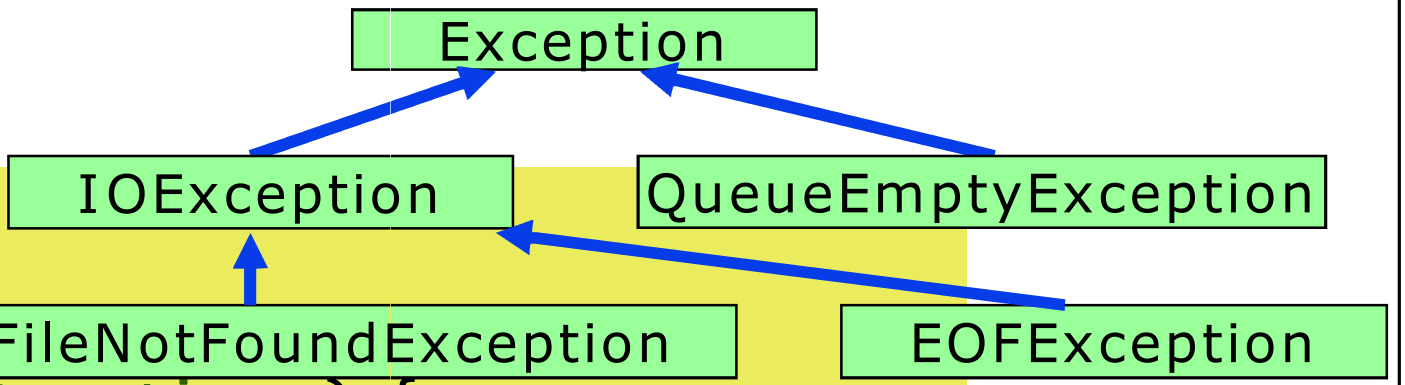
- This sequence of catches doesn't work:




```
try   {
   ...
} // try
catch (IOException e) {
   ...
} // catch IOException
catch (FileNotFoundException e)    {
   ...
// this code can never be reached because
// FileNotFound is subclass of IOException
} // catch FileNotFound Exception
```

Exceptions and Inheritance

- Something you can do with exceptions:




```
try   {
   ...
}  // try
catch (QueueEmptyException e) {
   ...
}  // catch QueueEmptyException
catch (IOException e) {
   ...
} // catch IOException
catch (Exception e) {
   ...
// this will catch any other kind of exception
// since all exceptions extend Exception
} // catch base Exception
```

"catch-all" handler

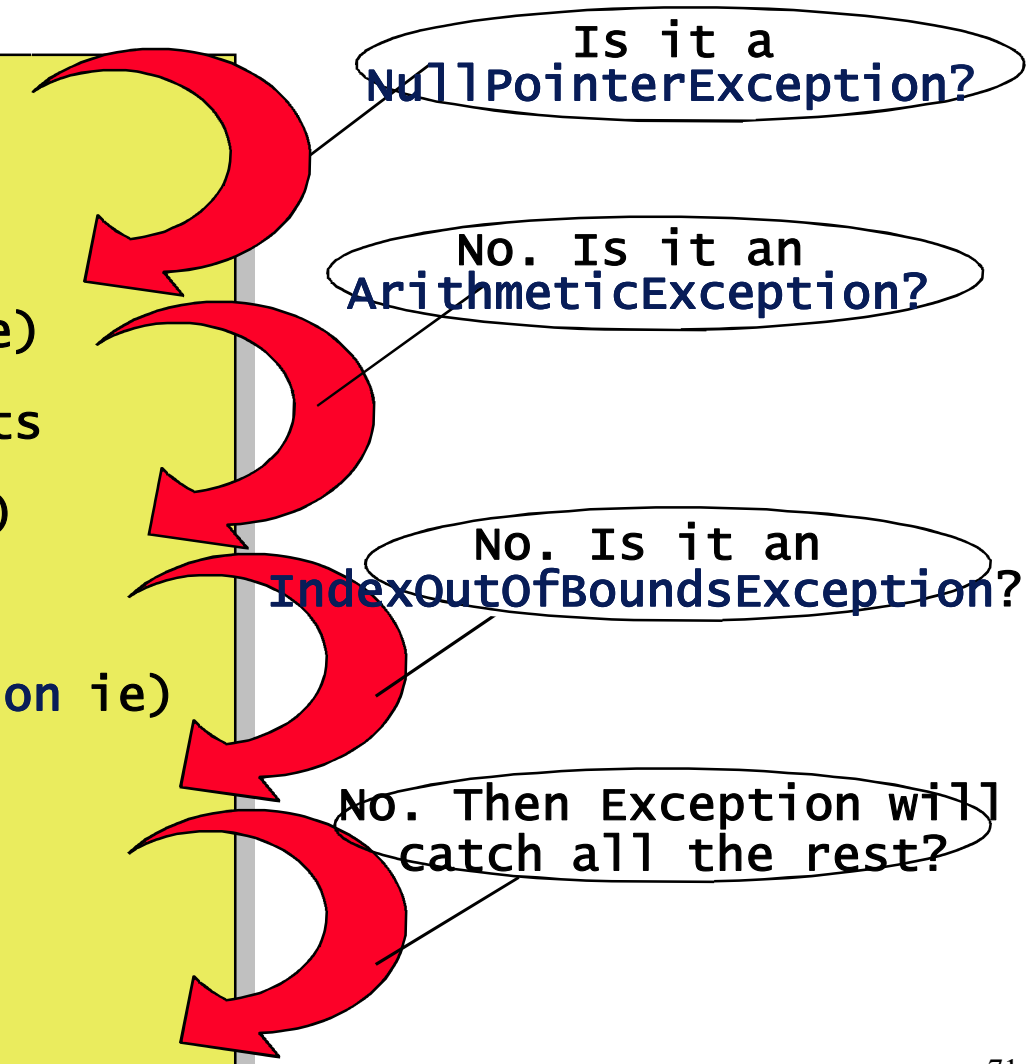
Multiple Exceptions and Inheritance

- You can process multiple exceptions arising from the same method using multiple catch blocks
  - be sure that the super Exception class is caught last
- If an exception of type T is thrown, each catch clause is examined in turn, from first to last, until one is found that matches type T
  - When found its block is executed
  - No other catch clause will be executed
- If you override a method, the subclass method cannot throw more checked exceptions (it can throw fewer)
  - In particular, if the superclass throw no exceptions at all, neither can the subclass

Catching Multiple Exceptions Example

badMethod() throws an Exception

```
try
{
   badMethod();
}
catch (NullPointerException ne)
{
  // fix uninitialized objects
}
catch (ArithmeticException ae)
{
  // fix arithmetic errors
}
catch (IndexOutOfBoundsException ie)
{
  // fix array errors
}
 catch (Exception e)
{
  // fix everything else
}
```



Nested Exception Handling Blocks

- Exception Handling blocks may be nested
- When an exception occurs, Java Runtime searches for the nearest matching handler for it
  - Only the nearest matching handler is executed for an exception
  - All other handlers skipped
- If an exception thrown by the body of the inner try statement is not caught by one of its catch clauses, it can be caught by one of the catch clauses of the outer try statement
  - Nesting is rarely needed or useful
  - Nesting can make code difficult to read

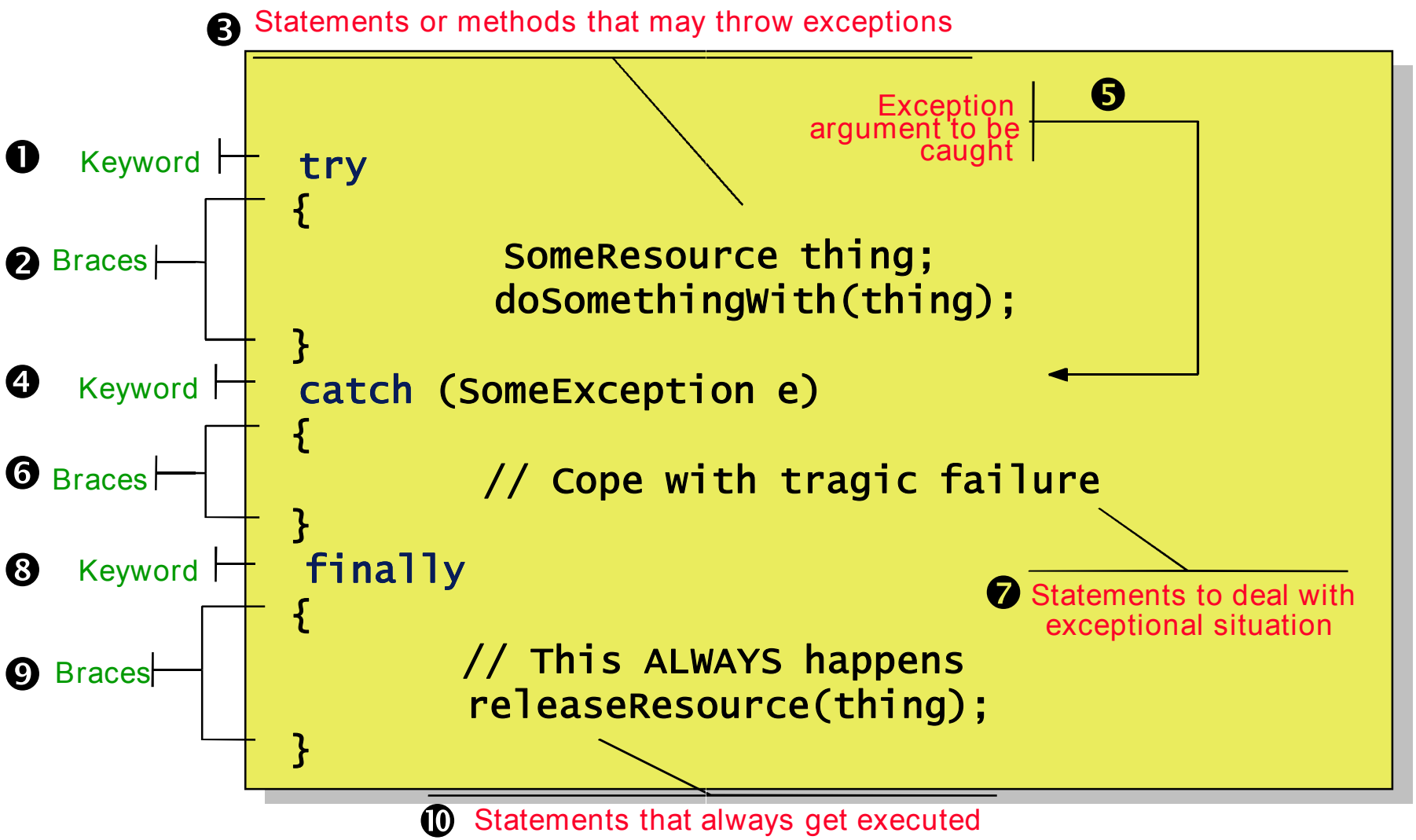
```
try{
    …
    try{
        ...
    }
    catch ( Exception1 ){...}
    …
}
catch( Exception2 ){...}
```

The Finally Block

- A block of code that comes after a try block and is executed before control exits the method either by return or by throwing an exception
  - the finally clause of a try block provides a mechanism for executing a section of code whether or not an exception is thrown

```
public boolean searchFor(String file, String word)
       throws StreamException
  { Stream input = null;
     try {
         input = new Stream(file);
         while ( !input.eof() )
             if ( input.next() == word ) return true;
         return false;
      } finally {
         if ( input != null ) input.close();
      }
  }
```

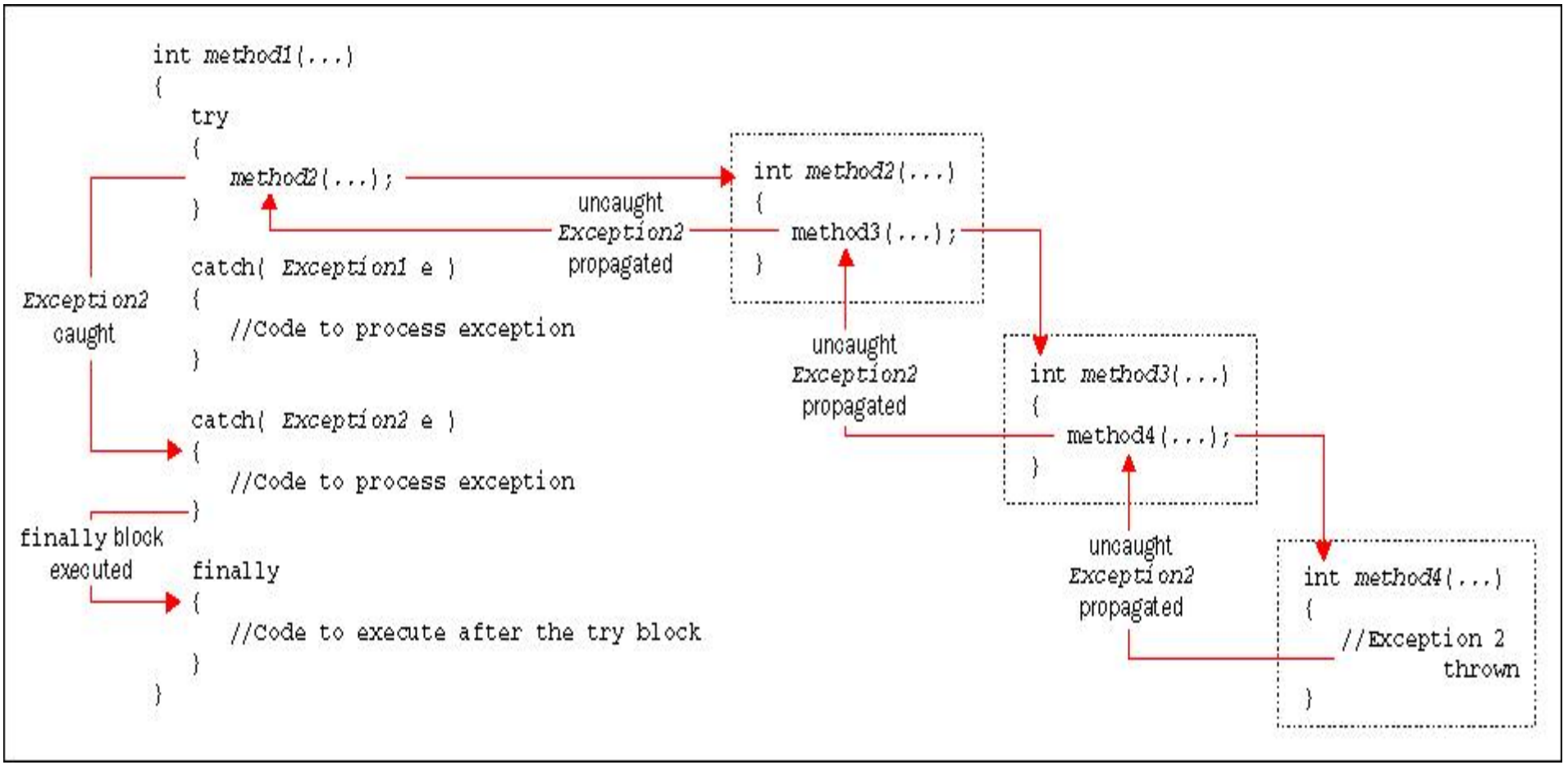
The Java Try/Catch/Finally Block



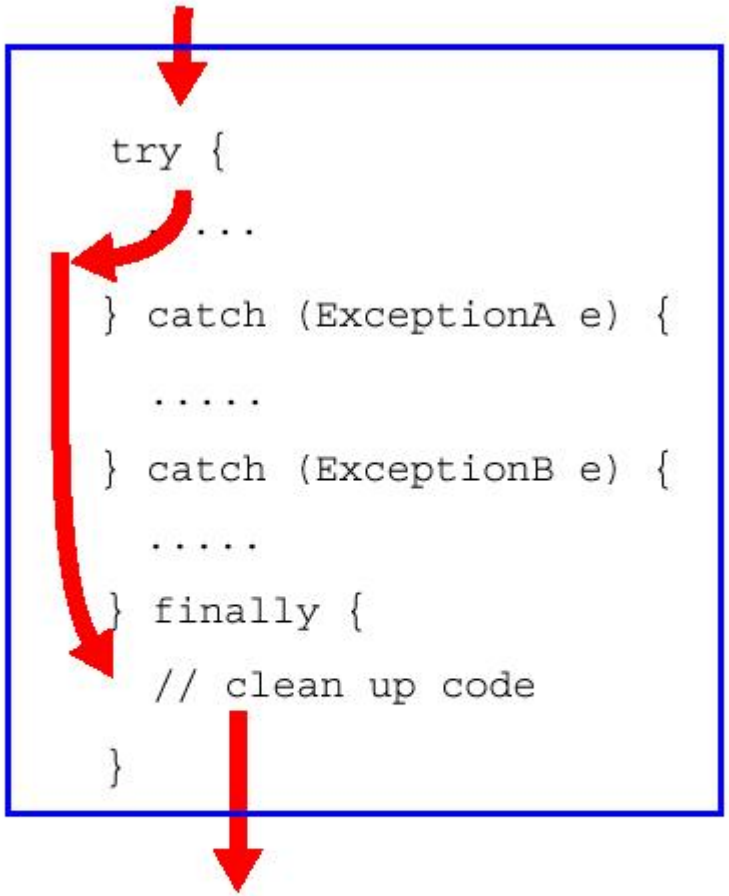
Scoping Rules

- Variables declared within a try block are not accessible outside the block
- Variables declared within one catch block may not be accessed from other catch blocks
- Variables declared at the beginning of a function may be accessed throughout the try and catch blocks

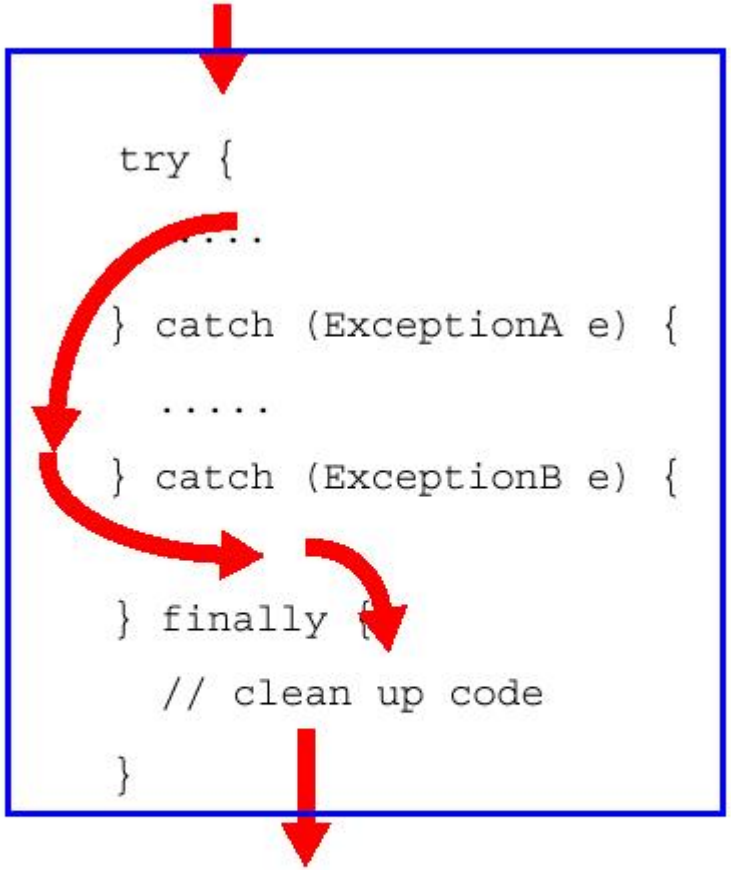
Control Flow when an Exception is not Caught



Possible Control Flows in Exception Handling

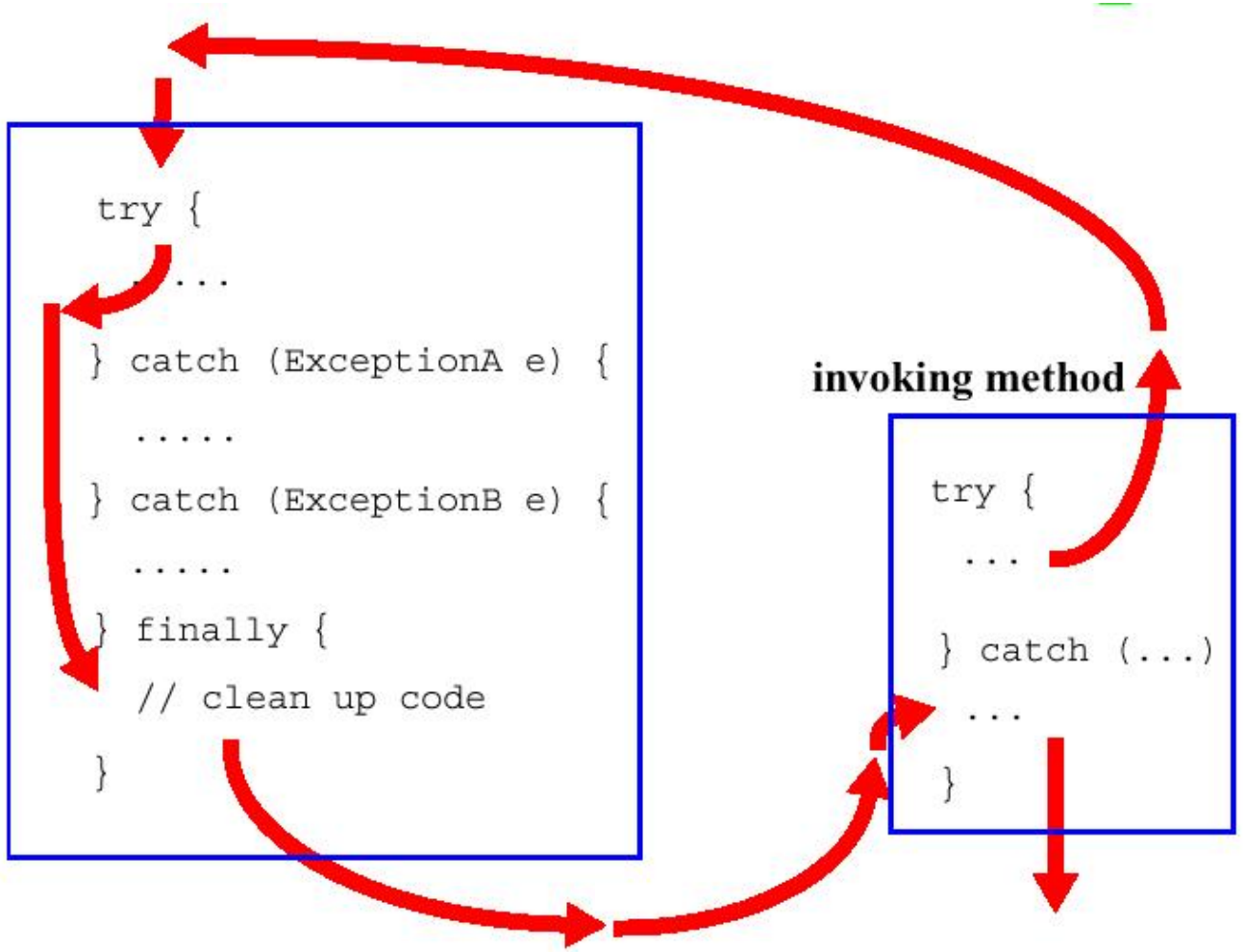


a. no exception thrown



b. exception thrown and caught locally

Possible Control Flows in Exception Handling



c. exception thrown and caught in invoking method

Documenting Exceptions

- The exceptions that might be thrown by a method should also be documented with the **`@exception`** tag:

```
/**
 * Sets the time of the clock
 * @param hour, minute, second The new time
 * @exception QueueEmptyException If we try
 * to dequeue an empty Queue
 */
```

Packaging Exceptions

- Exception types much be defined in some package
  - Either in the same package that contains the class of the method that throw them (longer names to avoid conflicts) or
  - In a separate package that defines exception types

- Note: Java does not require that exception types have the form `EnameException`

Exception strategies

- When to catch and when to throw?
  - If you know how to handle a situation, use **`try-catch`**
  - If you don't know how to handle an error, or,
  - If you think there may be situations where users of your code will want to handle an error differently, then use **`throw`**
- Use exceptions for exceptional situations only
  - exception handling is a less structured mechanism than the other language control structures
  - exception handling are usually less efficient than other control mechanisms
  - when used for dealing with situations other than errors, it could lead to a less understandable code

Dealing with Truly Exceptional Situations

- There is a problem with the internal state of the program
- A contract is violated
- A security risk arises (SecurityException)
- There is an error with an object or the data it manipulates.
- Coping with bad parameters
- Memory, stack problems

When to Use Exceptions: Internal State

```
public int getAge(int iSocialSecurityNumber)
                              throws RecordKeepingException
{
  int index = getHashKey(iSocialSecurityNumber);
   int iAge =
         myArray[index].getAge(iSocialSecurityNumber);

  if (iAge <= 0)
      throw new RecordKeepingException
         ("Exception: Age for " +
                                 iSocialSecurityNumber +
                                 " not in range: " + iAge);
  else
      return iAge
} // getAge
```

When to Use Exceptions: Contract Violated

```
public TreeNode getNodeRecursively (int index, TreeNode
                  currentNode) throws MissingNodeException  {

  if (currentNode == null)  {
        throw new MissingNodeException
            ("Exception: No node with " + index + " found");
     } // if
  else if (currentNode.getNumber() == index) {
        return currentNode;
      }  // else
       else if (currentNode.getNumber() > index) {
                return getNodeRecursively (index,
                                  currentNode.getLeftChild());
            }  // if
            else {
                  return getNodeRecursively (index,
                                  currentNode.getRightChild());
            } // else
}// getNodeRecursively
```

When to Use Exceptions: Errors with Objects

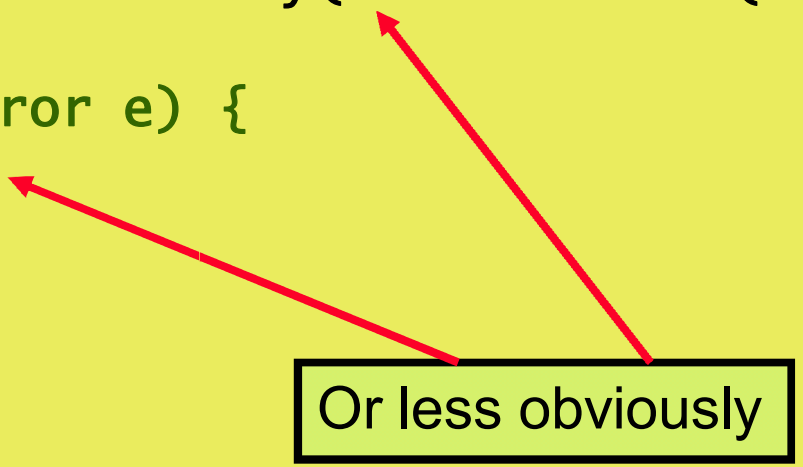
```
public void initializeTreeNode(int iNumberNodes) {

 if (myTree == null) {
     if (DEBUG)
       System.out.println ("Null tree found!");
     throw new NullPointerException ("Null tree found");
     /* NOTE: Runtime exception;
      *       no need to declare propagation */
   }
  else {
        for (int i=0; i < iNumberNodes; i++)  {
             TreeNode newNode = new TreeNode( i );
             tree.insertNode(newNode);
           }
       }

} // initializeTreeNode
```

When to Use Exceptions: Bad Parameters

```
public Integer convertNumber (String strToConvert) {

  for (int i =0; I < strToConvert.length(); i++) {
       char chTemp = strToConvert.charAt(i)
       if (!Character.isDigit(chTemp)) {
           if (DEBUG) System.out.println
                ("Bad input String: " + strToConvert);
           throw new NumberFormatException("Exception: " +
                          strToConvert + " is not numeric");
            }
     }

} // convertNumber
```

When to Use Exceptions: Memory Stack Overflow

```
public TreeNode getNode(int index) {

  TreeNode tempNode;
  try  {
        tempNode =
        myTree.getNodeRecursively(new TreeNode(index));
    }  // try
  catch(StackOverflowError e) {
        System.exit(1);
    }  // catch
  return tempNode;

}  // getNode
```



When Catching Exceptions you can . . .

- Print an error message
- Log the exception
- Retry the method (maybe with default parameters)
- Restore the system to some previously known "good" state
- Set the system to some "safe" state
- Let exception propagate to whoever called the method in which the exception arose
- "Catch it and ignore it" is generally bad:
  - If the error was serious enough to throw an exception, it should be dealt with, not ignored

local?

high-level?

OOA/OOD/OOP "Who"/what knows enough to handle the exception?

- What Should an Exception Handler Do?
  - fix the problem and try again
  - rethrow the exception
  - rethrow another exception
  - return the needed result

Rethrowing Exceptions

- Sometime you have to catch an exception in order to do some local cleanup, but you can't handle the root problem
- Rethrow exception with the statement: **`throw e;`**
  - Detected by next enclosing **`try`** block

```
try
{
      code that might throw exceptions
}
catch ( IOException ioExc )
{
      do some local cleanup (close files etc.)
      throw ioExc;
}
```

Rethrowing an Exception

```
try{
   try{
      …
   }
   catch ( IOException e){...; throw e;}
}
catch( Exception e){...}
```

Both handlers are executed

Changing Exceptions

- If for some reason you need to throw a different exception than the one you just caught – no problem!

```
try {
    mati.util.Widget a = new mati.util.Widget();
    a.load(s);
    a.paint(g);
    }

catch ( RuntimeException e ) {
    // Not again! Another Mati error
    throw new Exception("Mati Error");
    }
```

Tips on Using Exceptions (1)

1. Exception-handling is not supposed to replace control structures

Consider a simple program that tries 1,000,000 times to pop an empty stack

It first uses a simple test to check whether or not the stack is empty

```
if ( !s.empty() ) s.pop();
```

Now, lets pop the stack no matter what then we catch the **EmptyStackException**

```
try {
   s.pop();
} catch ( EmptyStackException e ) {}
```

Using the simple test took ~50ms
Using the exception took ~15000ms

Tips on Using Exceptions (2)

❷ Do not micromanage exceptions

Don't handle every possible exception separately

```
try{ some code  }
catch ( … ) { handle exception}
try{ some more code  }
catch ( … ) { handle exception}
```

instead use :

```
try{
      // all the code
}
catch ( … ) { }
catch ( … ) { }
```

Tips on Using Exceptions (3)

❸ Do not “shut up” exceptions

```
String loadString()
{
   try
   {
        lots of code
   }
   catch( Exception e ) { }
}
```

Tips on Using Exceptions (4)

❹ Propagating exceptions is not a shame

If you don't know how to handle an exception don't be ashamed to propagate it. Higher-level methods see a broader picture and have more information on how to handle the exception

The former does not mean however that you should always propagate exceptions

The Utility of Backtracking

- Backtracking is the process by which the stack frame is unwound in the presence of an unhandled exception.
- This process has some important properties:
  - Objects created on the stack are discarded to the GC.
  - Methods which cannot handle the exception are cleaned up implicitly.
  - Each stack frame has the ability to subsume the exception.
  - The end of the line is the JVM, which logs unhandled exceptions.
- Under C++, each object discarded is immediately, and fully, destroyed. This is a key problem for Java, and C#:

finally and finalize(), siblings?

- Java's garbage collector (GC) offers each method a type of destructor called 'finalizer()'.
- It is called with neither timing nor order guarantees, by the JVM.
- This is called non-deterministic finalization and gives rise to serious problems with Java objects which encapsulate OS resources: network, graphics, and database objects.
- Enter the finally block:

```
try {
  /* Insert code here. */
} finally {
  myResource.close();
}
```

Bytecode Layout

- Each method has a table of exception information at the start of it:
  - start_pc - where a try block begins
  - end_pc - where the try block ends, just before the first catch
  - catch_pc - the location of a catch block
  - catch_type - the class type caught by this catch block.
- If a finally block is in use, then each statement which can exit the try block has a special bytecode instruction attached to it (Java Specification Request-JSR). This calls the special finally subroutine.
- It is easy to see how a proliferation of try/catch blocks and many finally blocks can quickly grow the code and the exception table on the method.

Bytecode Layout, part 2

```
public void foo() {
   try {/* marks the start of a try-catch block: start_pc */

      int a[] = new int[2];
      a[4] = 1; /* causes a runtime exception due to the index */

   /* end of the code in the try block: end_pc */

   } catch (ArrayIndexOutOfBoundsException e) {/*catch_pc,
                                                 catch_type */
       System.out.println("AIOOBE = " + e.getMessage());

   } catch (NullPointerException npe) {/* catch_pc, catch_type */
       System.out.println("NPE = " + npe.toString();
   }
}
```

Performance Issues

- At a macro level you can make it easier on a JIT compiler by doing the following:
  - Do not use exceptions to manage flow of control: exit a loop with a status variable rather than raising an exception.
  - Place try blocks outside of a while loop, rather than inside it.
  - If you use a finally block, avoid having many exit paths in the content of the block. Remember that each exit point has a JSR attached to it.
  - The single largest exception cost in a non-native runtime is the creation of the exception, not the catching of it.
- In general avoid exceptions in the tight, fast code of your implementation. While the cost is minimal in JVM, the price rises rapidly in the JIT…

Exceptions and JIT Optimization

- Optimizing a Java program at runtime, within the JIT, requires a process called “path analysis”.
- A path analysis graph describes the structure of the method in the form of a graph. Each of the canonical control structures (e.g. if, for, while, …) creates new nodes and edges on the graph.
- In the presence of exceptions, new nodes and edges can be created for each and every Potentially Excepting Instruction (PEI) in the method encountered.
- Considering the number of both checked an unchecked exceptions, these secondary, exceptional, paths can proliferate quickly.
- The larger the path analysis graph, the more difficult it is for the JIT compiler to optimize the Java application.

Bytecode to Machine Code

- When the JIT compiler is generating machine code from the bytecode it is optimized over and over again.
- This optimization takes advantage of the path analysis graph, and potentially reorders the work of a method.
- Java's clear and precise specifications for exception handling deter reorder-based optimization. This is because program state must be correct before each PEI is invoked. (Make your eyes big and say, OOHH!)
- To really understand what all this means I would advise reading the presentation:
  - The Evolution of Optimization and Parallelization technologies for Java, or why Java for High Performance Computing is not an oxymoron! by IBM's Vivek Sarkar.

Why Runtime Exceptions are Not Checked

11.2.2

The runtime exception classes (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of Java, having to declare such exceptions would not aid significantly in establishing the correctness of Java programs. Many of the operations and constructs of the Java language can result in runtime exceptions. The information available to a Java compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such runtime exceptions cannot occur, even though this may be obvious to the Java programmer. Requiring such exception classes to be declared would simply be an irritation to Java programmers.

For example, certain code might implement a circular data structure that, by construction, can never involve null references; the programmer can then be certain that a `NullPointerException` cannot occur, but it would be difficult for a compiler to prove it. The theorem-proving technology that is needed to establish such global properties of data structures is beyond the scope of this Java Language Specification.

Lexicon: Actors and Actions

- **Operation**
  A method which can possibly raise an exception
- **Invoker**
  A method which calls operations and handles resulting exceptions
- **Exception**
  A concise, complete description of an abnormal event

- **Raise**
  Brings an exception from the operation to the invoker, called throw in Java. Another very common word for this is emit
- **Handle**
  Invoker's response to the exception, called catch in Java
- **Backtrack**
  Ability to unwind the stack frames from where the exception was raised to the first matching handler in the call stack

Lexicon: Types of Exceptions

- Hardware
  Generated by the CPU in response to a fault (e.g. divide by zero, overflow, segmentation fault, alignment error, etc)
- Software
  Defined by the developer to represent any other type of failure. These exceptions often carry much semantic information
- Domain Failure
  The inputs, or parameters, to the operation are considered invalid or inappropriate for the requested operation
- Range Failure
  Operation cannot continue, or output is possibly incorrect
- Monitor
  Describes the status of an operation in progress, this is a mechanism for runtime updates which is simpler than subthreads