Exception Handling

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- Unexpected events (End of File)
- Erroneous events (Subscript out of bounds)

When an exception occurs, the method currently executing creates an exception object and passes it to the runtime system, which looks for a special block of code, called an exception handler, that deals with the exception.

- A method “throws” an exception.
- An exception handler “catches” an exception.

Five reserved words: **try**, **catch**, **throw**, **throws**, **finally**

Advantages

- Code that handles errors and unusual events can be separated from the normal code.
- Errors automatically propagate up the calling chain until they are handled.
- Errors and special conditions can be classified and grouped according to common properties.

Classifying Exceptions

Checked Exceptions

- Programmer may not ignore these exceptions.
- Class Exception and all its subclasses except RuntimeException and its subclasses

  Must be handled by an exception handler using **catch**

  or

  Must be specified using a **throws** clause in method header.

Compiler verifies that each method only throws those checked exceptions that it declares it will throw.

Unchecked Exceptions (usually programmer errors)

Need not be specified in a **throws** clause in header of a method.

Need not be caught (although, they may).

Can occur anywhere in a program.

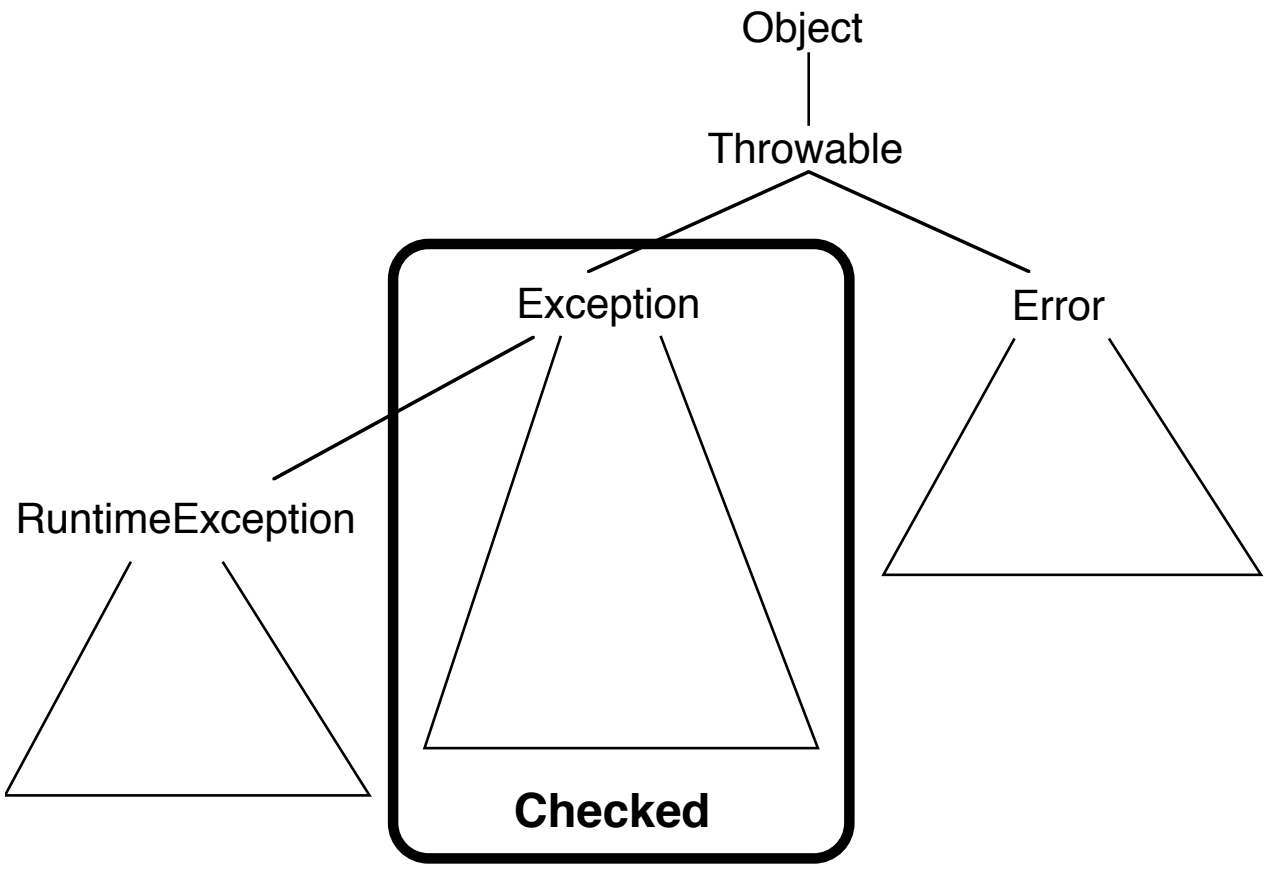
- Class RuntimeException and its subclasses
- Class Error and its subclasses

Note: All exceptions occur at runtime.

Some Exceptions

Object	java.lang
Throwable	java.lang
Exception	java.lang
ClassNotFoundException	java.lang
InterruptedException	java.lang
NoSuchMethodException	java.lang
IOException	java.io
EOFException	java.io
FileNotFoundException	java.io
MalformedURLException	java.net
UnknownHostException	java.net
AWTException	java.awt

RuntimeException	java.lang
ArithmeticException	java.lang
ClassCastException	java.lang
IllegalArgumentException	java.lang
NumberFormatException	java.lang
IndexOutOfBoundsException	java.lang
ArrayIndexOutOfBoundsException	java.lang
StringIndexOutOfBoundsException	java.lang
NegativeArraySizeException	java.lang
NullPointerException	java.lang
NoSuchElementException	java.util



Throwing Exceptions

- Explicitly using the throw command in a method:

  throw new ArithmeticException();

  Creates an exception object and throws it.
- Implicitly by operations being performed by the runtime system:

  Accessing a null pointer
  Subscript out of range
  Out of memory
- Implicitly by (library) methods called from the current program:

  Check the **throws** clauses in the documentation.

  Example (restoring a saved object)

  public final Object readObject()
  throws IOException, ClassNotFoundException

Catching Exceptions

Enclose the code that may raise exceptions in a **try** block followed by **catch** blocks:

```
try
{     // code that expects an exception might be thrown
      // or that calls methods that may throw exceptions
}
catch (ExceptType1 e)
{     // handle exceptions of this type and its subclasses
}
catch (ExceptType2 e)
{     // handle these exceptions
}
finally
{     // always execute this block
}
```

A **try** block must have at least one **catch** block or a **finally** block.

Consequences

- When an exception is raised, the runtime system probes backwards in the calling chain of methods (and blocks) until it finds a handler (catch block) that responds to the thrown exception.
- If none is found, the system reports the exception at the top level and terminates the program if it is an application.
- If and when an exception is caught, control continues with the code immediately following the try-catch-finally block in which the exception was caught.

Possible Execution Sequences

Suppose this method is called, say *getNumber(okay)*, where *okay* has some unknown boolean value.

```
static int getNumber(boolean b)
{
    try
    {   System.out.print("A");
        compute();
        System.out.print("B");
        if (b) return 456;
    }
    catch (FileNotFoundException e)
    {
        System.out.print("C");
        return -123;
    }
    catch (IOException e)
    {
        System.out.println("D");
        return -953;
    }
    finally
    {
        System.out.print("E");
    }
```

```
        System.out.print("F");
        return 789;
    }
```

Suppose that code inside the *compute* method may throw an exception.

Possible strings that can be printed and return values.

ABEF	ABE	ACE	ADE	AE
789	456	-123	-953	none

Variation

Change the finally block as follows

```
finally
{
    System.out.print("E");
    return 999;
}
```

Consequences

1. The compiler complains that the last two commands

   ```
   System.out.print("F");
   return 789;
   ```

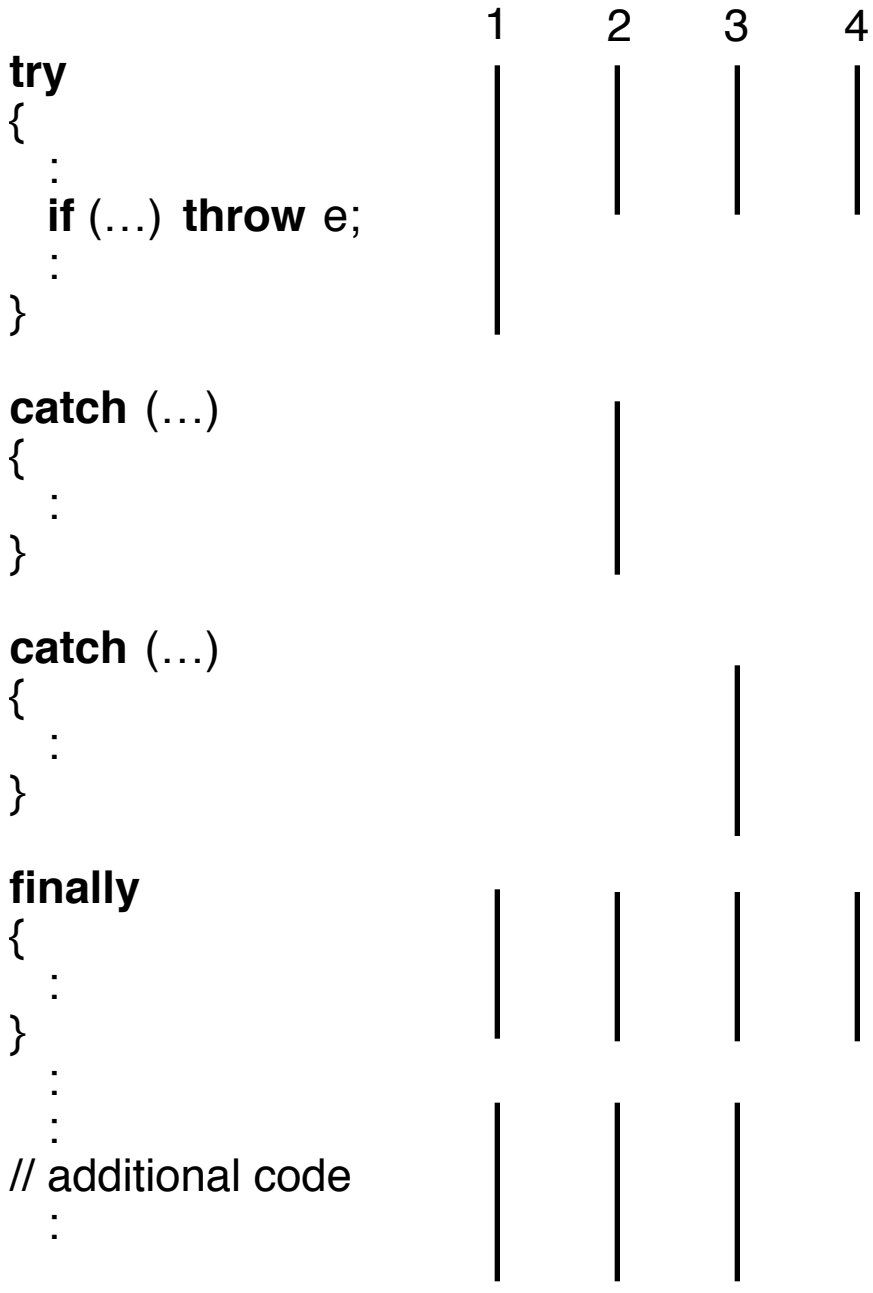
   are unreachable.

   Comment the last two lines of the method to get by compiler.

2. The return in the finally replaces the suspended return, either in the try block or in one of the catch block.

   The function returns 999 whenever no exception occurs or in the case an exception is thrown but also caught in the function.

Possible Execution Sequences



Example: No Catching

```
public class XTrace
{
   private String note = null;

   void printLength()
   {  System.out.println("Length = " + note.length());
      System.out.println("After printing length");
   }

   void caller()
   {  printLength();
      System.out.println("After call");
   }

   public static void main(String [] args)
   {
      System.out.println("Starting main");
      XTrace xt = new XTrace();
      xt.caller();
      System.out.println("Normal termination");
   }
}
```

Calling chain

main ➡ caller ➡ printLength

Output

```
% java XTrace
Starting main
Exception in thread "main" java.lang.NullPointerException
        at XTrace.printLength(XTrace.java:7)
        at XTrace.caller(XTrace.java:13)
        at XTrace.main(XTrace.java:21)
```


Catching the Exception

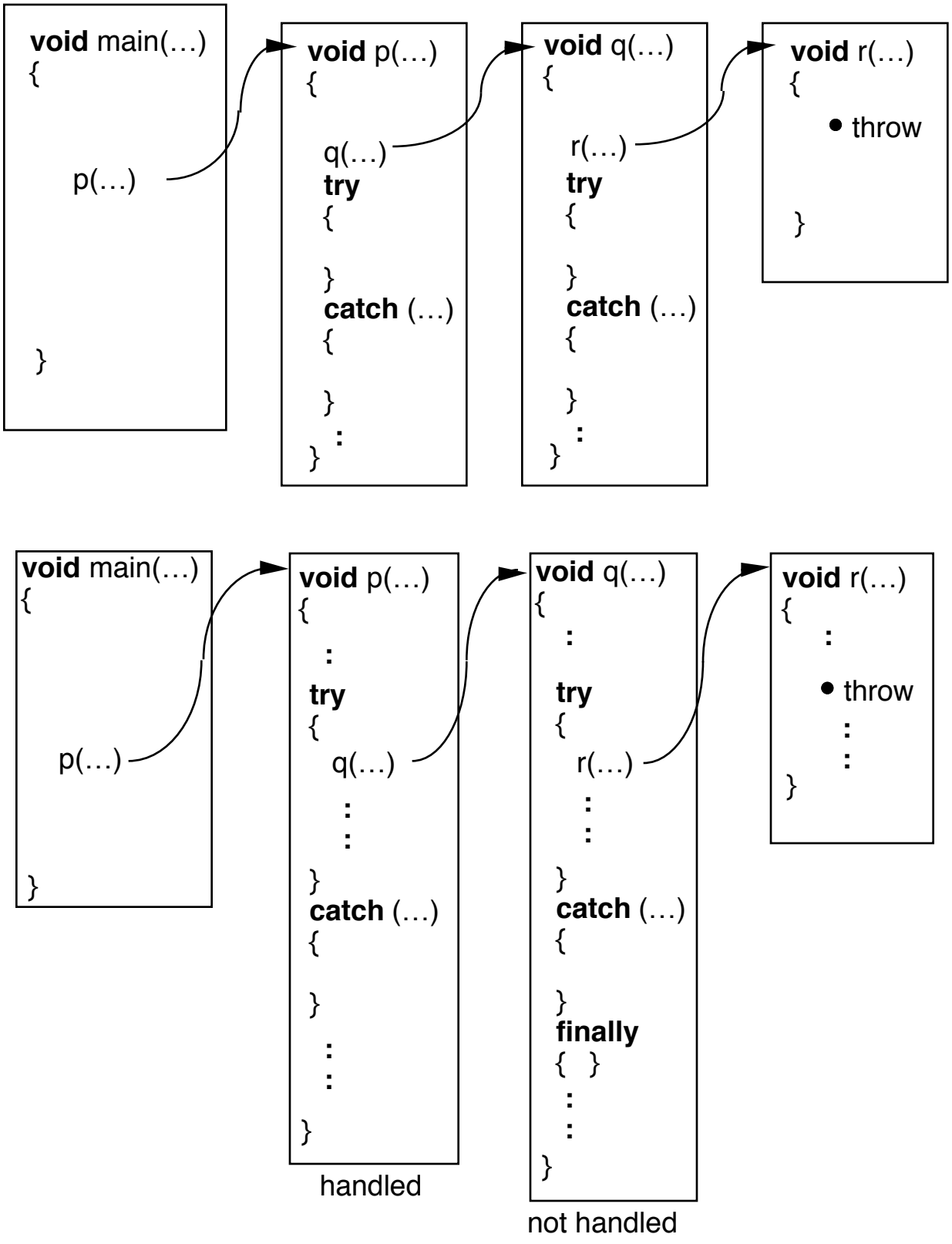
```
public static void main(String [] args)
{
   System.out.println("Starting main");
   XTrace xt = new XTrace();
   try
   {  xt.caller();
   }
   catch (Exception e)
   {  System.out.println("Message = " + e.getMessage());
      System.out.println("Stack trace: ");
      e.printStackTrace();
   }

   System.out.println("Normal termination");
}
```

Output

```
% java XTrace
Starting main
Message = null
Stack trace:
java.lang.NullPointerException
        at XTrace.printLength(XTrace.java:7)
        at XTrace.caller(XTrace.java:13)
        at XTrace.main(XTrace.java:23)
Normal termination
```

Examples: Control



Declaring New Exceptions

Normally user-defined exceptions are created as subclasses of Exception.

- These are checked.
- Subclass RuntimeException only if you have an exception that is an error you cannot recover from.

Example

```
class AutoException extends Exception
{ // … ①
}

class FlatTireException extends AutoException
{ // … ②
}

class OutOfGasException extends AutoException
{ // … ③
}

class TicketException extends AutoException
{ // … ④
}

class SpeedingException extends TicketException
{ // … ⑤
}

class ParkingException extends TicketException
{ // … ⑥
}
```

Catching these exceptions

```
catch (ParkingException pe)
{  // … handles only ⑥
}

catch (TicketException te)
{  // … handles ④, ⑤, and ⑥
}

catch (AutoException ae)
{  // … handles ①, ②, ③, ④, ⑤, and ⑥
}

catch (Exception e)
{  // … handles all exceptions
}
```

Heirarchy

```
Object
    Throwable
        Exception
            AutoException
                FlatTireException
                OutOfGasException
                TicketException
                    SpeedingException
                    ParkingException
```

