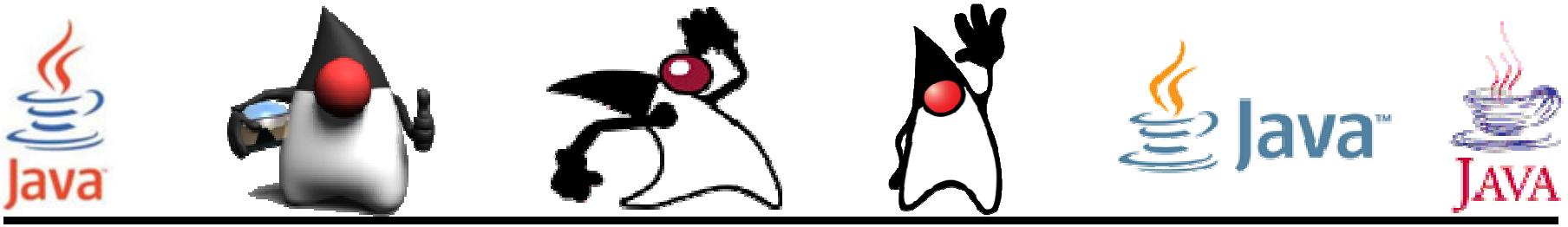


Excepciones

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Aprender el mecanismo de gestión de excepciones
- Aprender a lanzar y capturar excepciones
- Comprender la diferencia entre excepciones verificadas y no verificadas

Índice

- Qué es una excepción?
- Beneficios del sistema de gestión de excepciones
- Captura de excepciones con try-catch-finally
- Lanzamiento de excepciones
- Reglas en la gestión de excepciones
- Jerarquía de la clase Exception
- Excepciones verificadas y no verificadas
- Creación de clases exception propias
- Aserciones

Qué es una excepción?

- Evento excepcional, típicamente un error que ocurre en tiempo de ejecución
- Provoca una interrupción en la ejecución normal de un programa
- Ejemplos:
  - operaciones de división por cero
  - acceder a elementos de una array fuera de rango
  - entrada inválida
  - apertura de un fichero inexistente
  - agotamiento de memoria

Ejemplo de excepción

```
class DivByZero {
  public static void main(String args[]) {
    System.out.println(3/0);
    System.out.println("Por favor imprime esto");
  }
}
```

- Mensaje de error obtenido:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at DivByZero.main(DivByZero.java:3)
```

- Manejador de excepciones por defecto
  - Proporcionado por Java runtime
  - Imprime la descripción de la excepción (método)
  - Imprime la traza de la pila. Causa el fin del programa

Qué pasa cuando ocurre una excepción?

- Cuando una excepción ocurre dentro de un método, el método crea un objeto excepción y deja su manipulación al sistema *runtime*
  - La creación de un objeto excepción y su gestión por el sistema runtime se llama "*throwing an exception*"
  - El objeto excepción contiene información sobre el error, incluyendo su tipo y el estado del programa cuando ocurrió el error
- El sistema *runtime* busca en la pila de llamadas el método que contiene un gestor de excepciones

Qué pasa cuando ocurre una excepción?

- Cuando un manejador apropiado se encuentra, el sistema runtime pasa la excepción al manejador
  - Un manejador de excepciones es considerado apropiado si el tipo de objeto excepción lanzado iguala al tipo que puede manejar
  - El manejador de excepciones seleccionado se dice que captura la excepción
- Si el sistema *runtime* (rs) busca de forma exhaustiva todos los métodos de la pila de llamadas sin hallar un manejador apropiado, el rs termina y usa el manejador de excepciones por defecto

Beneficios del sistema de gestión de excepciones

- Separación del código de gestión de errores del código “regular” de la lógica de negocio
- Propagando errores en de la pila de llamadas
- Agrupando y diferenciando los tipos de error

Separando código de manejo de errores

- En la programación tradicional, la detección, reporte y manejo de errores con frecuencia conduce a código confuso (*spaghetti*)
- Como ejemplo considerar el pseudocódigo del método que lee un fichero entero a memoria

```
readFile {
  abrir el fichero;
  determinar el tamaño;
  reservar memoria;
  leer el fichero en memoria;
  cerrar el fichero;
}
```

Programación tradicional: NO separación del código de manejo de errores

- En programación tradicional, la función readFile tiene más código para detectar, reportar y manejar errores

```
errorCodeTipo readFile {
  inicializar errorCode = 0;
  abrir el fichero;
  if (theFileIsOpen) {
    determinar tamaño del fichero;
    if (gotTheFileLength) {
      reservar esa memoria;
      if (gotEnoughMemory) {
         leer el fichero en memoria;
         if (readFailed) { errorCode = -1; }
      } else { errorCode = -2;
      }
```

Programación tradicional: NO separación del código de manejo de errores

```
    } else {
        errorCode = -3;
    }
    cerrar el fichero;
    if (theFileDidntClose && errorCode == 0) {
       errorCode = -4;
    } else {
        errorCode = errorCode and -4;
    }
  } else {
     errorCode = -5;
  }
  return errorCode;
}
```

Separación del código de manejo de errores en Java

- Excepciones permiten escribir el flujo principal del código y tratar los casos excepcionales aparte

```
readFile {
  try {
    abrir el fichero;
    determinar el tamaño;
    reservar memoria;
    leer el fichero en memoria;
    cerrar el fichero; }
  catch (fileOpenFailed) { doSomething; }
  catch (sizeDeterminationFailed) { doSomething;}
  catch (memoryAllocationFailed) { doSomething; }
  catch (readFailed) { doSomething; }
  catch (fileCloseFailed) { doSomething; } }
```

Las excepciones no evitan el esfuerzo de detectar, reportar y gestionar los errores, pero ayuda a organizar el código de forma más efectiva

Propagando errores en la pila de llamadas

- Suponiendo que el método *readFile* es llamado por varios métodos anidados: metodo1 llama al metodo2, que llama al metodo3, que llama a *readFile*
- Se supone que metodo1 es el único interesado en los errores que pueden ocurrir en *readFile*

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

Forma tradicional de propagar errores

- La técnica tradicional de notificación de errores fuerza al metodo2 y metodo3 propagar los códigos de error retornados por readFile en la pila de llamadas hasta que los códigos de error finalmente alcanzan el metodo1, el único interesado en capturarlos

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

Uso del manejador de excepciones Java

- Un método puede pasar cualquier lanzamiento de excepciones dentro de él. De este modo permite que un método más arriba en la pila de llamadas los capture. Por tanto, sólo los métodos que atienden los errores tienen que preocuparse en su detección
- Toda excepción verificada que se puede lanzar dentro de un método, debe de ser especificada en la claúsula throws

```
method1 {
  try {
    call method2;
  } catch (exception e) {
    doErrorProcessing;
  }
}

method2 throws exception {
   call method3;
}

method3 throws exception {
   call readFile;
}
```

Agrupando y diferenciando tipos de error

- Debido a que todas las excepciones lanzadas dentro de un programa son objetos, el agrupamiento o categorización de excepciones es un resultado natural de la jerarquía de clases
- Un ejemplo de grupo de clases de excepciones en Java son las definidas en java.io.IOException y sus descendientes
  - IOException representa cualquier error I/O
  - Los descendientes representan errores más específicos: FileNotFoundException (fichero no encontrado en disco)

Agrupando y diferenciando tipos de error

- Un método puede tener manejadores específicos que pueden gestionar una excepción específica

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

- Otra alternativa es capturar una excepción general

```
// Captura todas las excepciones I/O, incluyendo
// FileNotFoundException, EOFException, etc.
catch (IOException e) {
...
}
```

Captura de excepciones: sentencia try-catch

- Sintaxis para capturar excepciones con try-catch:

```
try {
  <codigo a monitorizar por excepciones>
} catch (<ExceptionType1> <ObjName>) {
  <manejador si ocurre ExceptionType1 >
}
...
} catch (<ExceptionTypeN> <ObjName>) {
  <manejador si ocurre ExceptionTypeN >
}
```

Ejemplos: captura de excepciones con try-catch

```
class DivByZero {
  public static void main(String args[]) {
    try {
      System.out.println(3/0);
      System.out.println("Imprime esto pf.");
    } catch (ArithmeticException exc) {
      //Division por cero es una ArithmeticException
      System.out.println(exc);
    }
    System.out.println("Despues excepcion.");
  }
}
```

Ejemplos: captura de excepciones con try-catch

```
class MultipleCatch {
  public static void main(String args[]) {
    try {
      int den = Integer.parseInt(args[0]);
      System.out.println(3/den);
    } catch (ArithmeticException exc) {
      System.out.println("Divisor 0");
    } catch (ArrayIndexOutOfBoundsException exc2) {
      System.out.println("Fuera de rango");
    }
    System.out.println("Despues excepcion");
  }
}
```

Ejemplos: captura de excepciones con try anidados

```
class NestedTryDemo {
  public static void main(String args[]){
    try {
      int a = Integer.parseInt(args[0]);
        try {
          int b = Integer.parseInt(args[1]);
          System.out.println(a/b);
        } catch (ArithmeticException e) {
          System.out.println("Division por cero!");
        }
    } catch (ArrayIndexOutOfBoundsException) {
      System.out.println("Requiere 2 parametros");
    }
  }
}
```

Ejemplos: captura de excepciones con try anidados con métodos

```
class NestedTryDemo2 {
  static void nestedTry(String args[]) {
    try {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      System.out.println(a/b);
    } catch (ArithmeticException e) {
      System.out.println("Division por cero!");
    }
  }
  public static void main(String args[]){
    try {
      nestedTry(args);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Requiere dos parametros");
    }
  }
}
```

Captura de excepciones: palabra reservada *finally*

- Contiene código para “limpieza” después de un try-catch. Se ejecuta independientemente de errores
- Sintaxis try-catch-finally:

```
try {
  <codigo a monitorizar por excepciones>
} catch (<ExceptionType1> <ObjName>) {
  <manejador si ocurre ExceptionType1 >
} ...
} finally {
  <codigo a ejecutar antes termine bloque try>
}
```

Captura de excepciones: palabra reservada *finally*

- El código del bloque `finally` se ejecuta siempre una vez que se entra en un bloque try, incluso en:
  - Salidas forzadas que usan return, continue o break
  - Terminación normal
  - Captura de excepción lanzada
  - Excepción no capturada

Ejemplo: captura de excepciones y uso de finally

```
class FinallyDemo {
  static void myMethod(int n) throws Exception {
    try {
      switch(n) {
        case 1: System.out.println("1er case");  return;
        case 3: System.out.println("3er case");
                throw new RuntimeException("3!");
        case 4: System.out.println("4to case");
                throw new Exception("4!");
        case 2: System.out.println("2do case");
      }
    } catch (RuntimeException e) {
      System.out.print("RuntimeException: ");
      System.out.println(e.getMessage());
    } finally {
      System.out.println("try-block ejecutado");
    }
  }
```

Ejemplo: captura de excepciones y uso de finally

```
  public static void main(String args[]){
    for (int i=1; i<=4; i++) {
      try {
        FinallyDemo.myMethod(i);
      } catch (Exception e){
        System.out.print("Exception capturada: ");
        System.out.println(e.getMessage());
      }
      System.out.println();
    }
  }
}
```

Lanzamiento de excepciones: palabra reservada *throw*

- Java permite el lanzamiento o generación de excepciones

```
throw <objeto excepcion>;
```

- La excepción que se genera es un objeto, por lo que hay que crearlo como cualquier otro objeto
- Ejemplo:

```
throw new ArithmeticException("probando...");
```

Ejemplo: lanzamiento de excepciones

```
class ThrowDemo {
  public static void main(String args[]){
    String input = "input invalido";
      try {
        if (input.equals("input invalido")) {
          throw new RuntimeException("throw demo");
        } else {
          System.out.println(input);
        }
        System.out.println("Despues throwing");
      } catch (RuntimeException e) {
        System.out.println("Excepcion capturada:" + e);
      }
  }
}
```

Reglas sobre excepciones

- Se requiere un método para capturar o listar todas las excepciones que puede lanzar, excepto *Error* o *RuntimeException* o sus subclases
- Si un método puede causar una exepción (checked) pero que no lo captura, debe indicarlo mediante `throws`
- Sintaxis:

```
<tipo> <nombreMetodo> (<listaParametros>)
  throws <listaExcepciones> {
    <cuerpoMetodo>
}
```

Ejemplo: método lanza excepción

```
class ThrowingClass {
  static void meth() throws ClassNotFoundException {
    throw new ClassNotFoundException ("demo");
  }
}

-------------------------------------------------------
class ThrowsDemo {
  public static void main(String args[]) {
    try {
      ThrowingClass.meth();
    } catch (ClassNotFoundException e) {
      System.out.println(e);
    }
  }
}
```

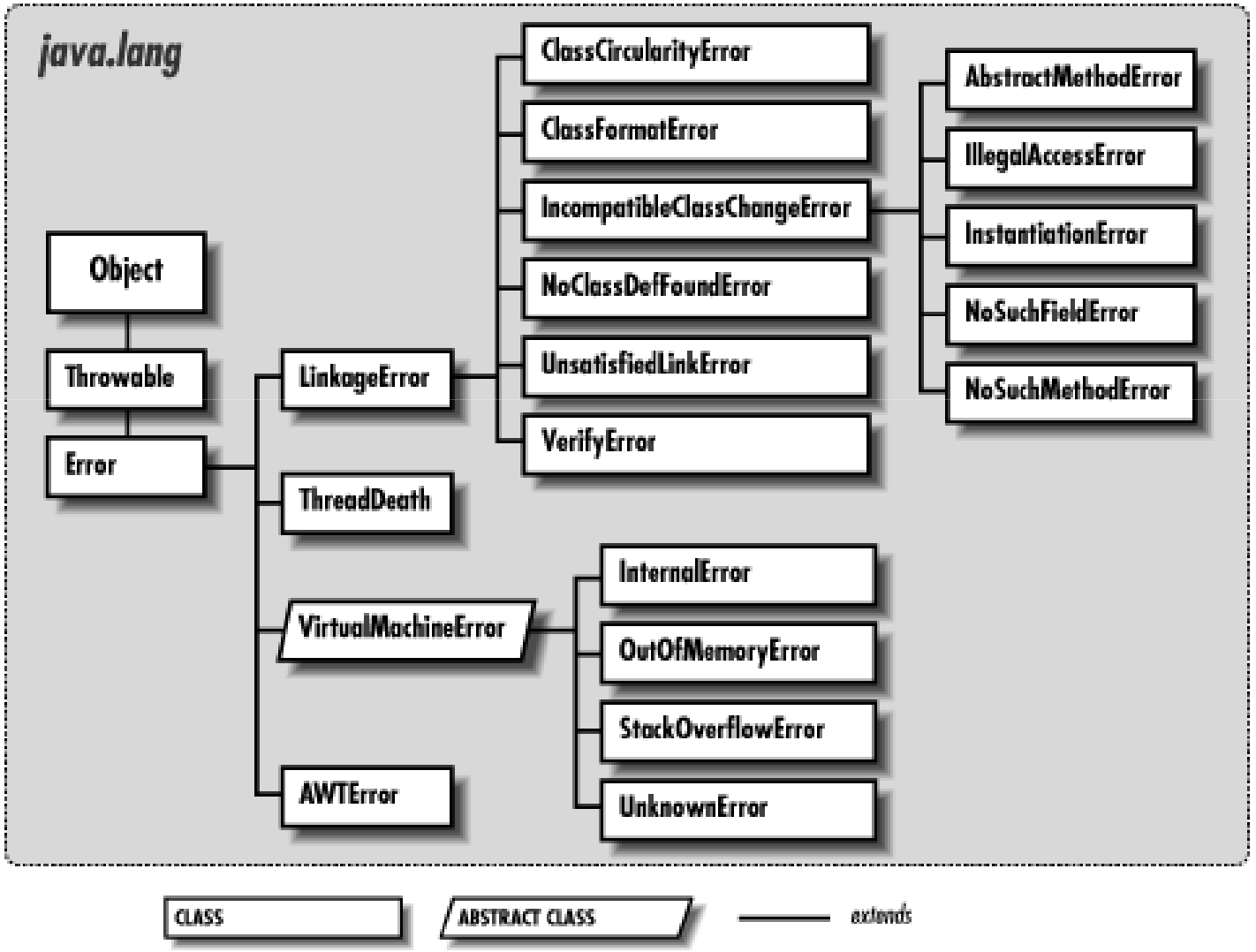
Jerarquía de la clase Exception

- Clase Throwable
  - Clase raíz de las clases exception
  - Subclases inmediatas
    - Error
    - Exception
- Clase Exception
  - Condiciones con las que los usuarios pueden tratar
  - Usualmente el resultado de defectos en el código. Ejms:
    - Error por división por cero
    - Error por fuera de rango

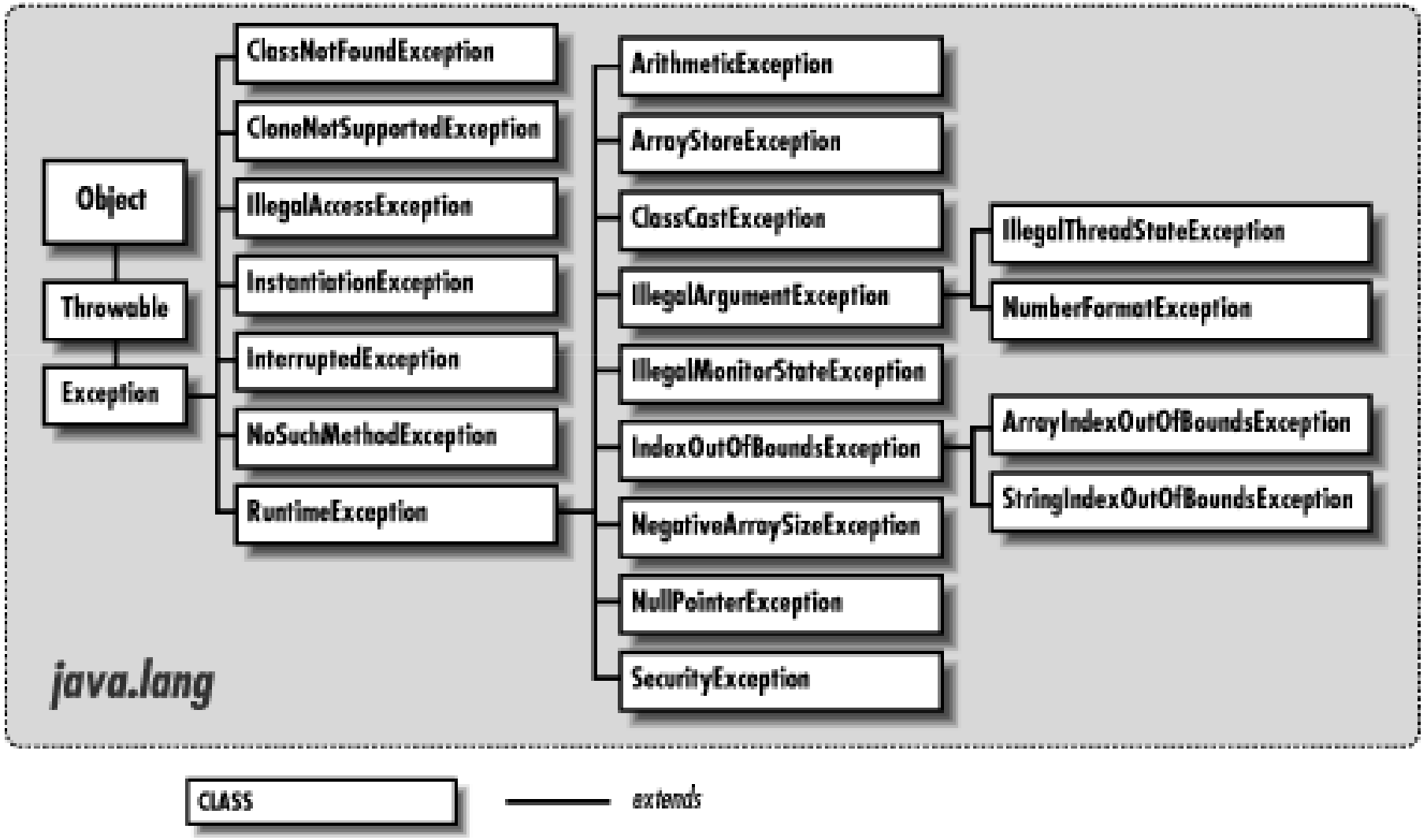
Clase Error

- Clase Error
  - Usada por el runtime Java para gestionar errores que ocurren el el entorno de tiempo real
  - Errores que están fuera de control de los usuarios
  - Ejemplos
    - Errores de memoria
    - Disco duro roto

Clases Error



Clases Exception



Clases Exception y jerarquía

- La captura múltiple debe ordenarse de subclases a superclases

```
class MultipleCatchError {
  public static void main(String args[]){
    try {
      int a = Integer.parseInt(args [0]);
      int b = Integer.parseInt(args [1]);
      System.out.println(a/b);
    } catch (ArrayIndexOutOfBoundsException e) {
      ...
    } catch (Exception ex) {
      ex.printStackTrace(); //para imprimir pila llamadas
    }
  }
}
```

Errores Checked y Unchecked

- Excepciones Checked
  - El compilador Java comprueba si el programa captura o lista las excepciones checked
  - Si no, ocurrirá un error del compilador
- Excepciones unchecked
  - No están sujetos a comprobación en tiempo de compilación para la gestión de las excepciones
  - Clases exception unchecked empotradas
    - Error
    - RuntimeException

Creando una clase excepción propia

- Pasos a seguir:
  - Crear una clase que extiende la clase RuntimeException o Exception
  - Configurar la clase
    - Los miembros y constructores debe ser añadida a la clase
- Ejemplo:

```
public class MyCheckedException extends Exception {
  //Se puede extender RuntimeException si es unchecked
}
```

Ejemplo: Creación de excepción propia

```
class NumberRangeException extends Exception {
  String msg;
  NumberRangeException() {
    msg = new String(“Ingresar numero entre 20 y 100");
  }
  public String toString() { return msg; }
}
                    -----------------
public class My_Exception {
  public static void main (String args [ ]) {
     try {
       int x = 10;  if (x < 20 || x >100)
          throw new NumberRangeException( ); }
       catch (NumberRangeException e) {
          System.out.println (e);  } }
}
```

Asertos (assert)

- Sentencias que permiten comprobar si el programa se comporta como se espera
- Informa a quien lee el código que una condición particular siempre debe satisfacerse
  - La ejecución del programa informa si los asertos realizados son verdaderos o falsos
  - Si un aserto no es verdadero, se lanza AssertionError
- Los usuarios tienen la opción de activar o desactivar cuando se ejecuta la aplicación

Activando o desactivando asertos

- Los programas con asertos puede que no trabajen de forma apropiada si el usuario no está prevenido que se ha usado asertos en el código
- Compilación con asertos
  - `> javac -source 1.4 MyProgram.java`
  - Para especificar compatibilidad de versiones
- Permitiendo asertos en ejecución
  - `> java -enableassertions MyProgram`
  - `> java -ea MyProgram`

Sintaxis de assert

- Dos formas:
  - Forma simple:

```
assert <expresion1>;
```

donde

<expresion1> es la expresión a comprobar su veracidad

  - Otra forma:

```
assert <expresion1>:<expresion2>;
```

donde

<expresion1> es la expresión a comprobar su veracidad

<expresion2> es alguna información útil en caso falla

Ejemplo: sintaxis assert

```
class AgeAssert {
  public static void main(String args[]) {
    int age = Integer.parseInt(args[0]);
       assert (age > 0):"ingresa numero positivo";
      /* si edad es valida (p.e. age>0) */
      if (age >= 18) {
        System.out.println("Eres adulto!");
      }
  }
}
```