Lenguaje Java Avanzado

Sesión 3: Tratamiento de errores

Índice

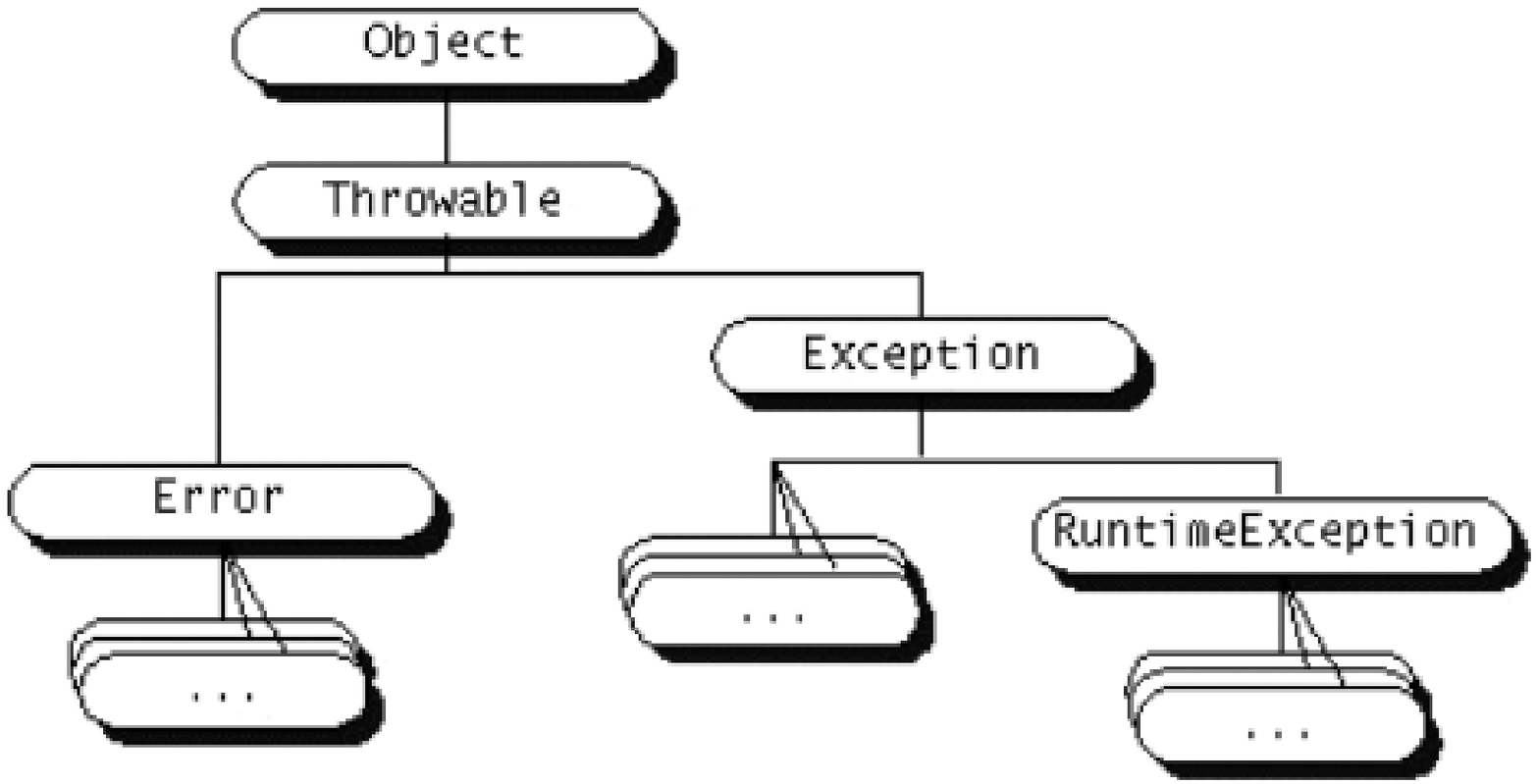
- Excepciones
- Captura de excepciones
- Propagación de excepciones
- Nested exceptions
- Errores en tiempo de compilación
- Tipos genéricos

Tratamiento de errores en tiempo de ejecución

- *Excepción*: Evento que sucede durante la ejecución del programa y que hace que éste salga de su flujo normal de ejecución
  - Se *lanzan* cuando sucede un error
  - Se pueden *capturar* para tratar el error

- Son una forma *elegante* para tratar los errores en Java
  - Separa el código normal del programa del código para tratar errores.

Jerarquía



Tipos de excepciones

- *Checked*: Derivadas de `Exception`
  - Es obligatorio capturarlas o declarar que pueden ser lanzadas
  - Se utilizan normalmente para errores que pueden ocurrir durante la ejecución de un programa, normalmente debidos a factores externos
  - P.ej. Formato de fichero incorrecto, error leyendo disco, etc

- *Unchecked*: Derivadas de `RuntimeException`
  - Excepciones que pueden ocurrir en cualquier fragmento de código
  - No hace falta capturarlas (es opcional)
  - Se utilizan normalmente para errores graves en la lógica de un programa, que no deberían ocurrir
  - P.ej. Puntero a `null`, fuera de los límites de un *array*,etc

Creación de excepciones

- Podemos crear cualquier nueva excepción creando una clase que herede de `Exception` (*checked*), `RuntimeException` (*unchecked*) o de cualquier subclase de las anteriores.

```
public class MiExcepcion extends Exception {
    public MiExcepcion (String mensaje) {
        super(mensaje);
    }
}
```

try-catch-finally

```
try {
    // Código regular del programa
    // Puede producir excepciones
} catch(TipoDeExcepcion1 e1) {
    // Código que trata las excepciones de tipo
    // TipoDeExcepcion1 o subclases de ella.
    // Los datos sobre la excepción los encontraremos
    // en el objeto e1.
    ...
} catch(TipoDeExcepcionN eN) {
    // Código que trata las excepciones de tipo
    // TipoDeExcepcionN o subclases de ella.
} finally {
    // Código de finalización (opcional)
}
```

Ejemplos

Sólo captura `ArrayOutOfBoundsException`

```
int [] hist = leeHistograma();
try {
    for(int i=1;;i++) hist[i] += hist[i-1];
} catch(ArrayOutOfBoundsException e) {
    System.out.println(“Error: “ + e.getMessage());
}
```

Captura cualquier excepción

```
int [] hist = leeHistograma();
try {
    for(int i=1;;i++) hist[i] += hist[i-1];
} catch(Exception e) {
    System.out.println(“Error: “ + e.getMessage());
}
```

Información sobre la excepción

- Mensaje de error

```
String msg = e.getMessage();
```

- Traza

```
e.printStackTrace();
```

- Cada tipo concreto de excepción ofrece información especializada para el error que representa
  - P.ej. `ParseException` ofrece el número de la línea del fichero donde ha encontrado el error

Lanzar una excepción

- Para lanzar una excepción debemos
  - Crear el objeto correspondiente a la excepción

```
Exception e = new ParseException(mensaje,linea);
```

  - Lanzar la excepción con una instrucción `throw`

```
throw e;
```

  - Si la excepción es *checked*, declarar que el método puede lanzarla con `throws`

```
public void leeFichero() throws ParseException {
    ...
    throw new ParseException(mensaje, linea);
    ...
}
```

Capturar o propagar

- Si un método lanza una excepción *checked* deberemos
  - Declarar que puede ser lanzada para propagarla al método llamador

```
public void init() throws ParseException {
    leeFichero();
}
```

  - O capturarla para que deje de propagarse

```
try {
    leeFichero();
} catch(ParseException e) {
    System.out.println(“Error en linea “ + e.getOffset() +
                       “: “ + e.getMessage());
}
```

- Si es *unchecked*
  - Se propaga al método llamante sin declarar que puede ser lanzada
  - Parará de propagarse cuando sea capturada
  - Si ningún método la captura, la aplicación terminará automáticamente mostrándose la traza del error producido

Nested exceptions

- Captura excepción causante
- Lanza excepción propia

```
try {
    ...
} catch(IOException e) {
    throw new MiExcepcion("Mensaje de error", e);
}
```

- Encadena errores producidos. Facilita depuración.
- Información detallada del error concreto.
- Aislar al llamador de la implementación concreta.

Errores en tiempo de compilación

- Son preferibles a los de ejecución
- Errores de sintaxis: ej, falta un punto y coma, referencia a nombre de variable inexistente
- Errores semánticos: de más alto nivel, ej. intentar usar el valor de una variable que nunca se ha inicializado.
- Errores en cascada: confunden al compilador y no se localiza correctamente la causa del error. Ej, cuando falta el cierre de una llave.

Errores en cascada

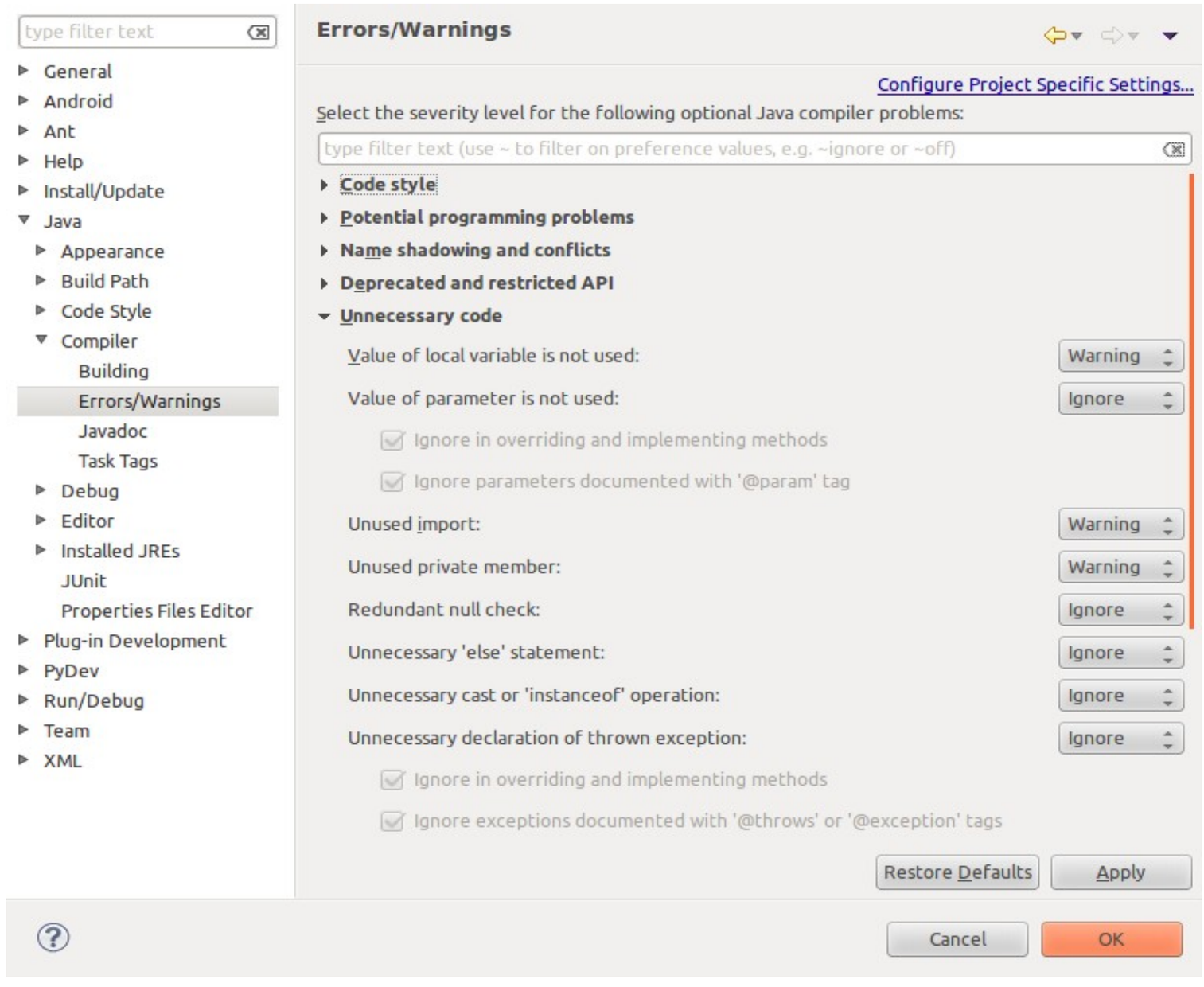
- Ejemplo:

```
fo ( int i = 0; i < 4; i++ ){}

Prueba.java:24: '.class' expected
        fo ( int i = 0; i < 4; i++ )
                   ^
Prueba.java:24: ')' expected
        fo ( int i = 0; i < 4; i++ )
                     ^
Prueba.java:24: not a statement
        fo ( int i = 0; i < 4; i++ )
                       ^
Pueba.java:24: ';' expected
        fo ( int i = 0; i < 4; i++ )
                               ^
Prueba.java:24: unexpected type
required: value
found   : class
        fo ( int i = 0; i < 4; i++ )
             ^
Prueba.java:24: cannot resolve symbol
symbol  : variable i
location: class Prueba
        fo ( int i = 0; i < 4; i++ )
                                 ^
6 errors
```

Warnings

- Ayudan a mejorar el estilo y la corrección del código
- Eclipse:
  permite aumentar
  niveles de warning



Herramientas de análisis

- Herramientas (externas) de análisis de código fuente
- Detecta código duplicado, código inalcanzable, código subóptimo, expresiones complicadas, y otras posibles fuentes de bugs
- Plugin para Eclipse

Comprobación de tipos: genéricos

- Antes de Java 1.5:

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Error en tiempo de ejecución
```

- Con tipos genéricos:

```
List<String> v = new ArrayList<String>();
v.add("test");
String  s = v.get(0); // Correcto (sin necesidad de cast explícito)
Integer i = v.get(0); // Error en tiempo ce compilación
```

Definición de genéricos

- Interfaces / Clases

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

- Uso

```
Entry<String, String>  grade440 =
  new Entry<String, String>("mike", "A");
Entry<String, Integer> marks440 =
  new Entry<String, Integer>("mike", 100);
System.out.println("grade: " + grade440);
System.out.println("marks: " + marks440);
```

```
public class Entry<K, V> {

  private final K key;
  private final V value;

  public Entry(K k,V v) {
    key = k;
    value = v;
  }

  public K getKey() {
    return key;
  }

  public V getValue() {
    return value;
  }

  public String toString() {
    return "(" + key + ",
           " + value + ")";
  }
}
```

Métodos genéricos

- También los métodos se pueden parametrizar con tipos genéricos

```
public static <T> Entry<T,T> twice(T value) {
   return new SimpleImmutableEntry<T,T>(value, value);
}

Entry<String, String> pair = this.<String>twice("Hello"); // Declarado
Entry<String, String> pair = twice("Hello");              // Inferido
```

Subtipos y comodines

- `ArrayList<Hija>` no es subtipo de `ArrayList<Padre>`
- Para flexibilizar el tipo podemos utilizar el comodín: <?>
- Para acotar el comodín en la jerarquía de herencia usamos `super` y `extends`
- Permitir clases derivadas:
  `ArrayList<? extends Padre>`
- Permitir superclases (clases padre):
  `ArrayList<? super Padre>`

Tipos genéricos y excepciones

- Es posible indicar a un método o una clase qué excepción debe lanzar, a través de genéricos

```
public <T extends Throwable> void metodo() throws T {
  throw new T();
}
// O bien

public class Clase<T extends Throwable≥
  public void metodo() throws T {
    throw new T();
  }
}
```

Tipos genéricos y excepciones

- No es posible es crear excepciones con tipos genéricos, debido al “borrado de tipos”
- Ejemplo, no se puede hacer:

```
public class MiExcepcion<T extends Object> extends Exception {
  private T someObject;

  public MiExcepcion(T someObject) {
    this.someObject = someObject;
  }

  public T getSomeObject() {
    return someObject;
  }
}
```

Borrado de tipos

- porque durante la compilación, tras comprobar que los tipos están bien, éstos se eliminan, el siguiente código pasa a ser:

```
try {
  //Código que lanza o bien
  //MiExcepcion<String>, o bien
  //MiExcepcion<Integer>
}
catch(MiExcepcion<String> ex) {
  // A
}
catch(MiExcepcion<Integer> ex) {
  // B
}
```

```
try {
  //Código que lanza o bien
  //MiExcepcion<String>, o bien
  //MiExcepcion<Integer>
}
catch(MiExcepcion ex) {
  // A
}
catch(MiExcepcion ex) {
  // B
}
```

¿Preguntas...?