



Sesión 3: Excepciones e hilos

Índice



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

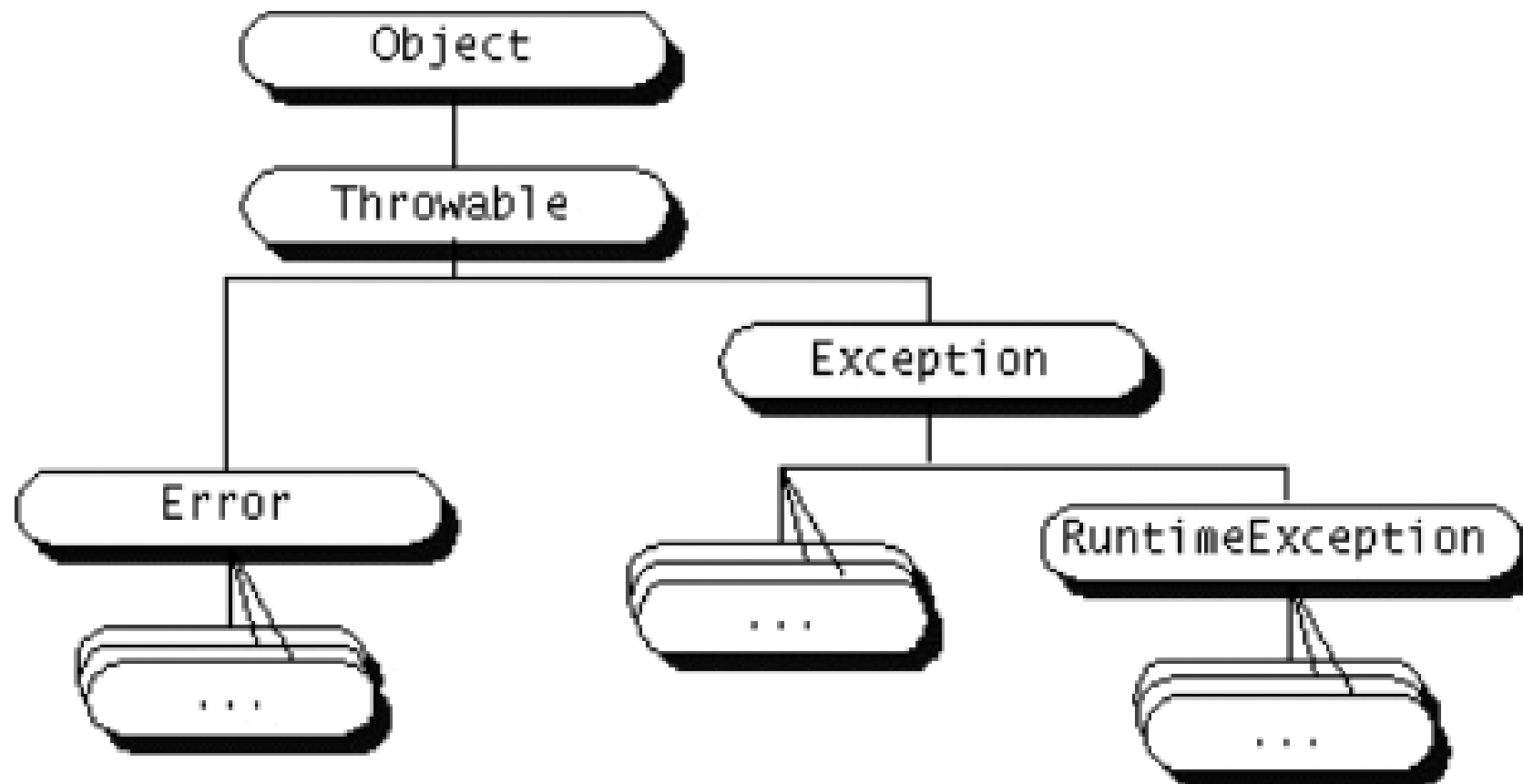
Tratamiento de errores en Java



- ***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

- ***No-checked:* 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`, salirse 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` (*no-checked*) o de cualquier subclase de las anteriores.

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

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

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

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

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 llamante

```
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 *no-checked*
 - 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

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

Hilos



- **Permiten realizar múltiples tareas al mismo tiempo**
- **Cada hilo es un flujo de ejecución independiente**
 - Tiene su propio contador de programa
- **Todos acceden al mismo espacio de memoria**
 - Necesidad de sincronizar cuando se accede concurrentemente a los recursos
- **Se pueden crear de dos formas:**
 - Heredando de `Thread`
 - Problema: No hay herencia múltiple en Java
 - Implementando `Runnable`
- **Debemos crear sólo los hilos necesarios**
 - Dar respuesta a más de un evento simultáneamente
 - Permitir que la aplicación responda mientras está ocupada
 - Aprovechar máquinas con varios procesadores

Heredando de Thread



- Crear una clase que herede de `Thread`
- Sobrescribir el método `run`

```
public class MiHilo extends Thread {  
    public void run() {  
        // Codigo de la tarea a ejecutar en el hilo  
    }  
}
```

- En este método introduciremos el código que será ejecutado por nuestro hilo
- Instanciar el hilo

```
Thread t = new MiHilo();
```

Implementando Runnable



- Crear una clase que implemente `Runnable`

```
public class MiHilo implements Runnable {  
    public void run() {  
        // Código de la tarea a ejecutar en el hilo  
    }  
}
```

- Definir en el método `run` el código de la tarea que ejecutará nuestro hilo
- Crear un hilo a partir de la clase anterior

```
Thread t = new Thread(new MiHilo());
```

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

Estados de los hilos



- **Nuevo hilo**

- El hilo acaba de instanciarse

- **Hilo vivo**

- Llamando al método `start` pasa al estado de hilo vivo

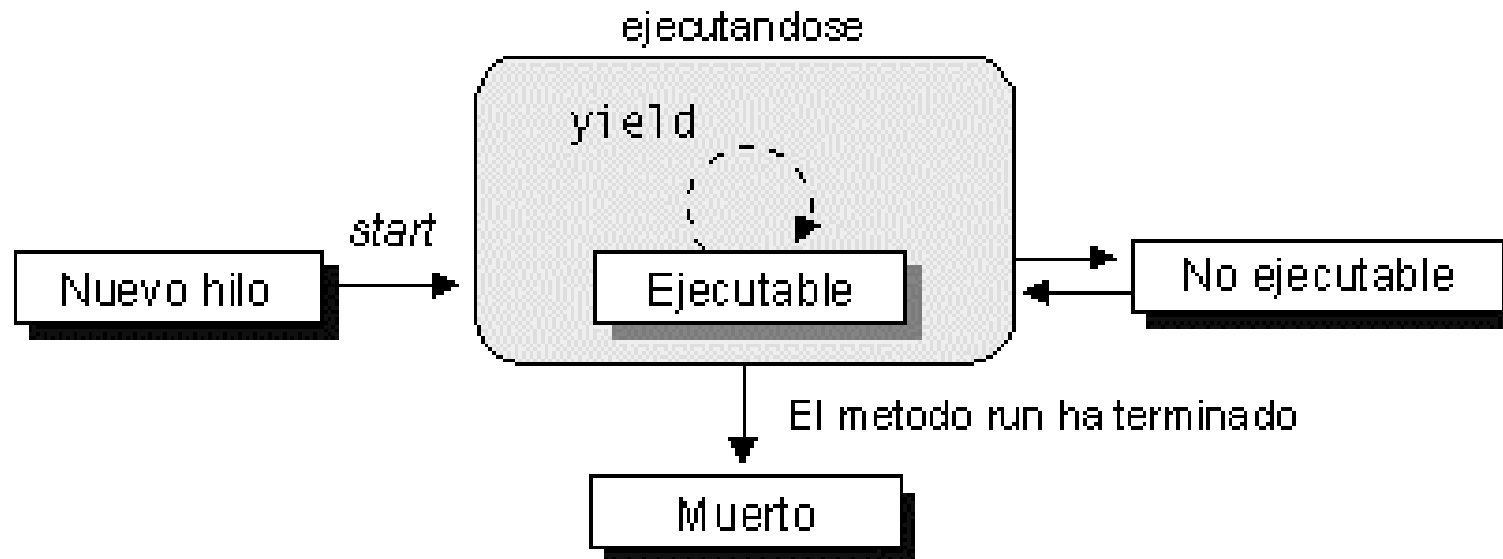
```
t.start();
```

- Durante este estado puede ser Ejecutable o No ejecutable

- **Hilo muerto**

- Se ha terminado de ejecutar el código del método `run`

Ciclo de vida



- **El hilo será no ejecutable cuando:**
 - Se encuentre durmiendo (llamando a `sleep`)
 - Se encuentre bloqueado (con `wait`)
 - Se encuentre bloqueado en una petición de E/S

Scheduler



- El *scheduler* decide qué hilo ejecutable ocupa el procesador en cada instante
- Se sacará un hilo del procesador cuando:
 - Se fuerce la salida (llamando a `yield`)
 - Un hilo de mayor prioridad se haga ejecutable
 - Se agote el *quantum* del hilo

Prioridad de los hilos



- Los hilos tienen asociada una prioridad
- El *scheduler* dará preferencia a los hilos de mayor prioridad
- Establecemos la prioridad con

```
t.setPriority(prioridad);
```

- La prioridad es un valor entero entre
 - Thread.MIN_PRIORITY
 - Thread.MAX_PRIORITY

Excepciones e hilos



- **Excepciones**
- **Captura de excepciones**
- **Propagación de excepciones**
- **Hilos en Java**
- **Estado y propiedades de los hilos**
- **Sincronización de hilos**

Acceso concurrente



- Cuando varios hilos acceden a un mismo recurso pueden producirse problemas de concurrencia
- *Sección crítica*: Trozo del código que puede producir problemas de concurrencia
- Debemos sincronizar el acceso a estos recursos
 - Este código no debe ser ejecutado por más de un hilo simultáneamente
- Todo objeto Java (`Object`) tiene una variable cerrojo que se utiliza para indicar si ya hay un hilo en la sección crítica
 - Los bloques de código `synchronized` utilizarán este cerrojo para evitar que los ejecute más de un hilo

Métodos sincronizados



- Sincronizan los métodos de un objeto

```
public synchronized void seccion_critica() {  
    // Código  
}
```

- Utilizan el cerrojo del objeto en el que se definen
 - Sólo un hilo podrá ejecutar uno de los métodos sincronizados del objeto en un momento dado

Bloques sincronizados



- Sincronizan un bloque de código

```
synchronized(objeto) {  
    // Código  
}
```

- Utilizan el cerrojo del objeto proporcionado
 - Sólo un hilo podrá ejecutar un bloque de código sincronizado con dicho objeto en un momento dado

Uso de la sincronización



- **Deberemos utilizar la sincronización sólo cuando sea necesario, ya que reduce la eficiencia**
- **No sincronizar métodos que contienen un gran número de operaciones que no necesitan sincronización**
 - **Reorganizar en varios métodos**
- **No sincronizar clases que proporcionen datos fundamentales**
 - **Dejar que el usuario decida cuando sincronizarlas en sus propias clases**

Bloqueo de hilos



- **Un hilo puede necesitar esperar a que suceda un determinado evento para poder continuar**
 - **P.ej, esperar a que un productor produzca datos que queremos consumir**
- **Deberemos bloquearlo para evitar que ocupe el procesador durante la espera**

```
wait();
```

- **Este método debe ser invocado desde métodos sincronizados**

Desbloquear hilos



- Cuando suceda el evento, deberemos desbloquearlo desde otro hilo

```
notify();
```

- Podemos utilizar `notifyAll` para desbloquear todos los hilos que haya bloqueados
- Será conveniente utilizar `notify` ya que es más eficiente, excepto en el caso en que varios hilos puedan continuar ejecutándose
- Estos métodos también deben ser invocados desde métodos sincronizados

Dependencia de hilos



- Podemos necesitar esperar a que un hilo haya acabado de ejecutarse para poder continuar
 - P.ej, si necesitamos que se haya completado la tarea que realiza dicho hilo
- Podemos quedarnos bloqueados esperando la finalización de un hilo `t` con:

```
t.join();
```