capítulo 16

Excepciones

objetivos

En este capítulo aprenderá a:

- Detectar procesos anormales que ocurren en la realización de operaciones.
- Definir bloques de código en los que expresamente se detectarán errores.
- Conocer las excepciones más habituales que se dan en las operaciones matemáticas.
- Propagar las excepciones que ocurren en un método cuando se le llama.
- Especificar manejadores para el tratamiento de posibles excepciones.
- Incorporar la cláusula `finally` para liberar recursos del sistema.

introducción

Un problema importante en el desarrollo de *software* es la gestión de condiciones de error; no importa lo eficiente que éste sea ni su calidad, siempre aparecerán errores por múltiples razones; surgirán, por ejemplo, imprevistos de programación de los sistemas operativos, agotamiento de recursos, etcétera; las *excepciones* son, normalmente, condiciones de errores súbitos que suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema, algunos ejemplos típicos son: agotamiento de memoria, erratas de rango en intervalos, división entre 0, archivos no existentes, etcétera. El manejo de excepciones es el mecanismo previsto por Java para el tratamiento de estas equivocaciones; normalmente el sistema aborta la ejecución del programa cuando se produce una excepción; Java permite al programador intentar la recuperación de estas condiciones y decidir si continuar o no la ejecución del programa.

16.1 Condiciones de error en programas

La escritura de código fuente y el diseño correcto de clases y métodos es una tarea difícil y delicada, por ello es necesario manejar con eficiencia las erratas que se produzcan; la mayoría de los diseñadores y programadores se enfrentan a dos preguntas importantes en el manejo de errores:

1. ¿Qué tipo de problemas se pueden esperar cuando los clientes hacen mal uso de clases, métodos y programas en general?
2. ¿Qué acciones hay que tomar una vez que se detectan estos problemas?

El manejo de errores es una etapa importante en el diseño de programas ya que no siempre se puede asegurar que las aplicaciones utilizarán objetos o llamadas a métodos correctamente; en lugar de añadir conceptos aislados de manejo de errores al código del programa, se prefiere construir un mecanismo de dicho manejo como una parte integral del proceso de diseño.

Para ayudar a la portabilidad y diseño de bibliotecas, Java incluye un mecanismo de manejo de excepciones que está soportado por el lenguaje; en este capítulo se explora el manejo de excepciones y su uso en el diseño; para ello se examinará lo siguiente: detección y manejo de errores, gestión de recursos y especificaciones de excepciones.

16.1.1 ¿Por qué considerar las condiciones de error?

La captura (*catch*) o localización de errores es siempre un problema en programación; en la fase de desarrollo se debe pensar cómo localizar los errores; también hay otras cuestiones a tener en cuenta como: ¿dónde deben capturarse los errores?, o ¿cómo pueden manejarse?

Al encontrar un error, un programa tiene varias opciones: terminar inmediatamente; ignorar el error con la esperanza de que no suceda algo *desastroso*; o establecer un indicador o señal de error, el cual, presumiblemente se comprobará por otras sentencias del programa; en esta última opción, cada vez que se llama a un método específico, el llamador debe comprobar el valor de retorno del método, y si se detecta un error, se debe determinar un medio de recuperar el error o de finalizar el programa.

En la práctica, los programadores no son consistentes con estas tareas debido en parte a que exige una gran cantidad de tiempo y también porque las sentencias de verificación de errores a veces oscurecen la comprensión del resto del código; también es difícil recordar cómo capturar o *atrapar* cada condición posible de error, cada vez que se llama a un método determinado; con frecuencia, el programador debe hacer esfuerzos excepcionales para liberar memoria, cerrar archivos de datos y en general liberar recursos antes de detener un programa; por ello, Java incorpora la cláusula `finally` para definir un bloque cuyo cometido es liberar recursos del sistema utilizados en las sentencias que se controlan.

La solución a tales problemas en Java es llamar mecanismos del lenguaje que soportan manejo de errores para evitar hacer códigos de manejo de errores complejos y artificiales en cada programa; cuando se genera una excepción en Java, el error no se puede ignorar porque el programa terminará; si el código de tratamiento del error encuentra el tipo de error específico, el programa tiene la opción de recuperación del error y continuar la ejecución; este enfoque ayuda a asegurar que no se deslice ninguna equivocación con consecuencias fatales y origine que un programa se ejecute erráticamente.

Un programa *lanza* (*throws*) una excepción en el momento en que se detecta el error; cuando esto sucede, Java busca automáticamente en un bloque de código llamado manejador de excepciones para responder de un modo apropiado; esta respuesta se llama *capturar* o *atrapar* una excepción (*catching an exception*). Si no encuentra un manejador de excepciones, ésta se propaga en el programa hasta ser captada por la rutina que llamó a la que lanzó la excepción, y así sucesivamente hasta alcanzar la rutina que devuelve control al sistema.

16.2 Tratamiento de los códigos de error

Los desarrolladores de *software* han utilizado durante mucho tiempo códigos para indicar condiciones de error; normalmente los procedimientos devuelven un código para seña-

lar los resultados; por ejemplo: un procedimiento `abrirArchivo` puede devolver `0` para indicar un fallo; otros procedimientos pueden devolver `-1` para mostrar un error y `0` para señalar éxito; los sistemas operativos y las bibliotecas de *software* documentan todos los códigos de error posibles que pueden regresarse por un procedimiento; los programadores que utilizan tales procedimientos deben comprobar cuidadosamente el código devuelto y las acciones a realizar en función de los resultados, lo cual puede producir cambios más serios en el código posterior. Cuando una aplicación termina anormalmente pueden suceder anomalías; por ejemplo, archivos abiertos o conexiones de redes que no se cierran u omisiones en la escritura de datos en disco.

Una aplicación bien diseñada e implementada no debe permitir que esto suceda; los errores no se deben propagar innecesariamente; si éstos se detectan y se manejan de inmediato, se evitan daños mayores en el código posterior; al contrario, cuando se propagan a diferentes partes del código, será mucho más difícil trazar la causa real del problema debido a que los síntomas pueden no indicar la causa real.

Cuando se identifica una equivocación en una aplicación, se debe fijar la causa del problema y volver a intentar procesar la operación que lo produjo; por ejemplo: si se produce una errata porque el índice de un arreglo se fijó más alto que el final de dicho arreglo, es fácil localizar la causa del desacierto; pero en otras situaciones, la aplicación puede no ser capaz de fijar la condición de error y el programador debe tener la libertad de tomar la decisión correcta.

Cuando un método encuentra una condición de error y vuelve al llamador, se debe esperar que ejecute las operaciones de limpieza necesarias antes de retornar al llamador; el código que detecta y maneja la condición de error debe incorporar un código extra para esta operación de limpieza; en otras palabras, no es una tarea fácil y se complica cuando en el método hay múltiples puntos de salida; si existen diversas posiciones en el código en las que se deben verificar las condiciones de error, el método se volverá complicado y enredado.

En ciertos casos, una rutina puede no tener información suficiente para manejar una condición de error, entonces es más seguro propagar las condiciones del error a la rutina exterior para poderlo manejar; las rutinas que devuelven códigos de error sencillos pueden no ser capaces de cumplir estos retos.

Los códigos de error devueltos por los métodos no transmiten mucha información al llamador; normalmente se trata de un número que indica la causa del fallo; sin embargo, en muchas situaciones es útil si existe más información disponible sobre la causa del fallo en el llamador; esto ayudará a fijar la condición de error pero un código de error sencillo no puede cumplir este objetivo.

En resumen, las alternativas para el manejo de errores en programas son:

1. Terminar el programa.
2. Devolver un valor que represente el error.
3. Devolver un valor legal pero establecer un indicador o una señal de error global.
4. Llamar a una rutina de error proporcionada por el usuario.

Cualquiera de los métodos tiene inconvenientes y deficiencias, en ocasiones graves; por esta causa es necesario buscar nuevos métodos o esquemas, uno de ellos fue comprobado, es popular y está soportado en muchos lenguajes como Java, Ada, o C++, se trata del principio de levantar o alzar (*raising*) una excepción, en el cual se fundamenta el mecanismo de manejo de excepciones de Java citado anteriormente y que trataremos ahora.

16.3 Manejo de excepciones en Java

La palabra excepción indica una irregularidad en el *software* que se inicia en alguna sentencia del código al encontrar una condición anormal; no se debe confundir con una excepción *hardware*.

Una excepción es un error de programa que sucede durante la ejecución; si al ocurrir está activo un segmento de código denominado manejador de excepción, entonces el flujo de control se transfiere al manejador; si no existe un manejador para la excepción, ésta se propaga al método que invoca, si en este tampoco se capta, la excepción se propaga al que a su vez le llamó; si llega al método por el que empieza la ejecución, es decir, `main()` y tampoco es captada, la ejecución termina.

Una excepción se levanta cuando el *contrato* entre el llamador y el llamado se violan; por ejemplo: si se intenta acceder a un elemento fuera del rango válido de un arreglo se produce una violación del contrato entre el método que controla los índices (operador `[]` en Java) y el llamador que utiliza el arreglo; la función de índices garantiza que devuelve el elemento correspondiente si el índice que se le pasó es válido; pero si éste no lo es, la función de índice debe indicar la condición de error; siempre que se produzca tal violación al contrato se debe levantar o alzar una excepción.

Una vez que se levanta una excepción, ésta no desaparece aunque el programador lo ignore, lo cual no se debe hacer; una condición de excepción se debe reconocer y manejar porque, en caso contrario, se propagará dinámicamente hasta alcanzar el nivel más alto de la función; si también falla el nivel de función más alto, la aplicación termina sin opción posible.

PRECAUCIÓN

El manejo de errores usando excepciones no evita errores, sólo permite su detección y su posible reparación.

En general, el mecanismo de excepciones en Java permite:

a) detectar errores con posibilidad amplia de recuperación;
b) limpiar errores no manejados, y
c) evitar la propagación sistemática de errores en una cadena de llamadas dinámicas.

16.4 Mecanismo del manejo de excepciones en Java

El modelo de un mecanismo de excepciones consta fundamentalmente de cinco nuevas palabras reservadas: `try`, `throw`, `throws`, `catch` y `finally`.

- `try` es un bloque para detectar excepciones,
- `catch` es un manejador para capturar excepciones de los bloques `try`,
- `throw` es una expresión para levantar (*raise*) excepciones,
- `throws` indica las excepciones que puede elevar un método,
- `finally` es un bloque opcional situado después de los `catch` de un `try`.

Los pasos del modelo son:

1. Establecer un conjunto de operaciones para anticipar errores; esto se realiza en un bloque `try`.
2. Cuando una rutina encuentra un error, lanzar una excepción; el lanzamiento (*throwing*) es el acto de levantar una excepción.
3. Para propósitos de limpieza o recuperación, anticipar el error y capturar (*catch*) la excepción lanzada.

El mecanismo de excepciones se completa con:

- Un bloque `finally` que, si se especifica, siempre se ejecuta al final de un `try`;
- Especificaciones de excepciones que dictamina cuáles, si existen, puede lanzar un método.

16.4.1 Modelo de manejo de excepciones

La filosofía que subyace en el modelo de manejo de excepciones es simple; el código que trata con un problema no es el mismo que lo detecta; cuando una excepción se encuentra en un programa, la parte que detecta la excepción puede comunicar que la expresión ocurrió levantando o lanzando una excepción.

De hecho, cuando el código de usuario llama a un método incorrectamente o utiliza un objeto de una clase de forma inadecuada, la biblioteca de la clase crea un objeto excepción con información sobre lo que era incorrecto; la biblioteca levanta una excepción, acción que hace el objeto excepción disponible al código de usuario a través de un manejador de excepciones; el código de usuario que maneja la excepción puede decidir qué hacer con el objeto excepción.

El cortafuegos que actúa de puente entre una biblioteca de clases y una aplicación debe realizar varias tareas para gestionar debidamente el flujo de excepciones, reservando y liberando memoria de modo dinámico (véase figura 16.1).

Una de las razones más significativas para utilizar excepciones es que las aplicaciones no pueden ignorarlas; cuando el mecanismo levanta una excepción, alguien debe tratarla, en caso contrario la excepción es *no capturada* (*uncaught*) y el programa termina por omisión; este poderoso concepto es el corazón del manejo de excepciones y obliga a las aplicaciones a manejar excepciones en lugar de ignorarlas.

El mecanismo de excepciones de Java sigue un modelo de terminación; esto implica que nunca vuelve al punto en que se levanta una excepción; estas últimas no son como manejadores de interrupciones que bifurcan una rutina de servicio antes de volver al punto de interrupción. Esta técnica, llamada *resumption*, tiene un alto tiempo suplementario, es propensa a bucles infinitos y es más complicada de implementar que el modelo de terminación. Diseñado para manejar únicamente excepciones síncronas con un solo hilo de control, el mecanismo de excepciones implementa un camino alternativo de una sola vía en diferentes sitios de su programa.

EJEMPLO 16.1

Se escribe un fragmento de código Java en el que en el método `main()` se define un bloque `try` que invoca al método `f()` que lanza una excepción.

```
void escucha() throws Exception
{
  // código que produce una excepción que se lanza
  // ...
  throw new FException();
```

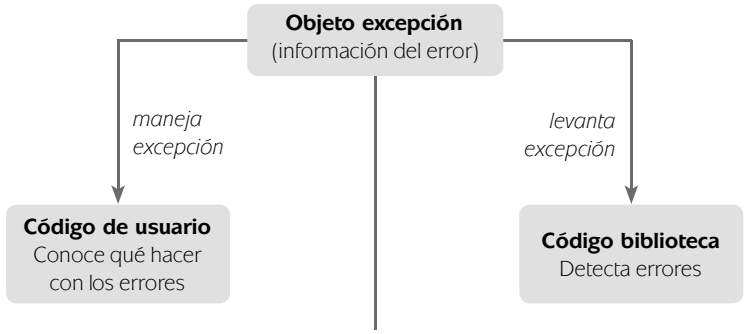


Figura 16.1 El manejo de excepciones construye un *cortafuego*.

```
  //  ...
}
static public void main(String[] a)
{
  try
  {
    escucha(); // método que puede lanzar cualquier excepción
    // Código normal aquí
  }
  catch(FException t)
  {
    // Captura una excepción de tipo FException
    // Hacer algo
  }
  catch(Exception e)
  {
    // Captura cualquier excepción
    // Hacer algo
  }

  // Resto código
}
```

`escucha()` es un simple procedimiento en el que se declara que puede lanzar cualquier excepción; esto se hace en la cabecera del método, con

```
throws Exception
```

`Exception` es la clase base de las excepciones que son tratadas; debido a la conversión automática entre clase derivada y clase base, se afirma que puede lanzar cualquier excepción; cuando el método encuentra una condición de error, lanza una excepción mediante la sentencia

```
throw new FException ();
```

El operando de la expresión `throw` es un objeto que normalmente almacena información sobre el error ocurrido; como Java proporciona una jerarquía de clases para el manejo de excepciones, el programador puede declarar clases para el tratamiento de errores en la aplicación.

En el método `main()`, la llamada a `escucha()` se encierra en un bloque `try`; éste es un bloque de código encerrado dentro de una sentencia `try`:

```
try
{
  // ...
}
```

Un bloque `catch()` captura una excepción del tipo indicado; en el ejemplo anterior se han especificado dos:

```
catch(FException t)
catch(Exception e)
```

Una expresión `catch` es comparable a un procedimiento con un único argumento.

16.4.2 Diseño de excepciones

La palabra reservada `try` designa un bloque con el mismo nombre que es un área del programa que detecta excepciones; en el interior de dichos bloques, por lo general se llaman a métodos que pueden levantar o lanzar excepciones. La palabra reservada `catch` designa un manejador de capturas con una signatura que representa un tipo de excepción; dichos manejadores siguen inmediatamente a bloques `try` o a otro manejador `catch` con un argumento diferente.

Los bloques `try` son importantes, ya que sus manejadores de captura asociados determinan cuál es la parte del programa que maneja una excepción específica; el código del manejador de capturas (`catch`) decide qué se hace con la excepción lanzada.

Java proporciona un manejador especial denominado `finally`; es opcional y, de utilizarse, debe escribirse después del último `catch()`; comúnmente, su finalidad es liberar recursos asignados al bloque `try`; por ejemplo: cerrar archivos abiertos en alguna sentencia del bloque `try`; este manejador tiene la propiedad de que siempre se ejecuta una vez terminado el bloque o cuando una excepción es capturada por el correspondiente `catch()`.

16.4.3 Bloques `try`

Ya se mencionó que un bloque `try` encierra las sentencias que pueden lanzar excepciones y que comienza con la palabra reservada `try` seguida por una secuencia de sentencias de programa encerradas entre llaves; a continuación del bloque hay una lista de manejadores llamados cláusulas *catch*. Al menos un manejador *catch* debe aparecer inmediatamente después de un bloque `try`, si no hay tal manejador, debe especificarse el manejador opcional `finally`. Cuando un tipo de excepción lanzada coincide con el argumento de un `catch`, el control se reanuda dentro del bloque de su manejador; si ninguna excepción se lanza desde un bloque `try`, una vez que terminan las sentencias del bloque, prosigue la ejecución a continuación del último `catch`. Si hay manejador `finally`, la ejecución sigue por sus sentencias; una vez terminadas, continúa la ejecución en la sentencia siguiente.

La sintaxis del bloque `try` es:

1.
```
try
  {
    código del bloque try
  }
  catch (signatura)
  {
    código del bloque catch
  }
```

2.
```
try

   sentencia compuesta
   lista de manejadores
```

También se pueden anidar bloques `try`.

```
void sub(int n) throws Exception
{
  try {
```

```
    ...   // bloque try externo
    try { // bloque try interno
      ...
      if (n == 1)
        return ;
    }
    catch (signatura1) {...} // manejador catch interno
  }
  catch (signatura2) {...}   // manejador catch externo
  ...
}
```

Una excepción lanzada en el bloque interior `try` ejecuta el manejador `catch` con `signatura1` si coincide el tipo de excepción; el manejador con `signatura2` maneja excepciones lanzadas desde el bloque `try` exterior si el tipo de la excepción coincide. El manejador externo de `catch` también captura excepciones lanzadas desde el bloque interior si el tipo de excepción coincide con `signatura2` pero no con `signatura1`. Si los tipos de excepción no coinciden con alguna, la excepción se propaga al llamador de `sub()`.

Normas

```
try
{
  sentencias
}
catch (parámetro)
{
  sentencias
}
catch (parámetro)
{
  sentencias
}
etc.
```

1. Cuando una excepción se produce en sentencias del bloque `try`, hay un salto al primer manejador `catch` cuyo parámetro coincida con el tipo de excepción.
2. Cuando las sentencias en el manejador ya se ejecutaron, se termina el bloque `try` y la ejecución prosigue en la sentencia siguiente; nunca se produce un salto hacia atrás, donde ocurrió la interrupción.
3. Si no hay manejadores para tratar con una excepción, se aborta el bloque `try` y la excepción se relanza.
4. Si utiliza el manejador opcional `finally`, debe escribirse después del último `catch`; la ejecución del bloque `try`, se lance o no una excepción, siempre termina con la sentencia `finally`.

PRECAUCIÓN

Se puede transferir el control fuera de bloques `try` con sentencias `goto`, `return`, `break` o `continue`. Si se especificó el manejador `finally`, primero se ejecuta éste y después transfiere el control.

EJEMPLO 16.2

El método `calcuMedia()` calcula una media de arreglos de tipo `double`, para ello invoca al método `avg()`. En caso de ser llamado de manera incorrecta, `avg()` lanza excepciones del tipo `MediaException`.

La excepción que se va a lanzar en caso de error, `MediaException`, debe ser declarada como una clase derivada de `Exception`:

```
class MediaException extends Exception
{
  //
}
```

El método `calcuMedia()` define un bloque `try` para tratar excepciones:

```
double calcuMedia(int num)
{
  double b[] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
  double media;

  try
  {
    media = avg(b, num); // calculo media
    return media;
  }
  catch (MediaException msg)
  {
    System.out.println("Excepcion captada: " +  msg);
    System.out.println("Calcula Media: uso de longitud del arreglo "
                       + b.length);
    num = b.length;
    return avg(b, num);
  }
}
```

El método `avg()` debe tener en la cabecera la excepción que puede lanzar:

```
double avg(double[]p, int n) throws MediaException
```

El método `calcuMedia()` define un arreglo de seis de tipo `double` y llama a `avg()` con el nombre del arreglo (`b`) y un argumento de longitud `num`; un bloque `try` contiene a `avg()` para capturar excepciones de tipo `MediaException`.

Si `avg` lanza la excepción `MediaException`, el manejador `catch` la capta, escribe un mensaje y vuelve a llamar a `avg()` con un valor por defecto, en este caso, la longitud del arreglo `b`.

16.4.4 Lanzamiento de excepciones

La sentencia `throw` levanta una excepción; cuando la encuentra, la parte del programa que la detecta puede comunicar que ésta ocurrió por levantamiento, o lanzamiento; su formato es:

```
throw objeto
```

El operando `throw` debe ser un objeto de una clase derivada de la clase `Exception` porque una excepción lanzada hace que termine el bloque `try` y que las sentencias que siguen no se ejecuten; el objeto que se lanza puede contener información relativa al problema surgido.

El manejador `catch` que captura una excepción realiza un proceso con ella y puede decidir devolver control, `return`, o continuar la ejecución en el mismo método a continuación del último `catch`; incluso la excepción actual se puede relanzar con la misma sentencia: `throw` objeto, la cual normalmente se utiliza cuando se desea llamar a un segundo manejador desde el primero para procesar posteriormente la excepción.

EJEMPLO 16.3

Se suponen tres métodos: `main()` tiene un bloque `try` en el que se llama a `deolit()`; éste tiene su bloque `try` en el cual se llama a `lopart()`; este último lanza una excepción que es atrapada por el `try-catch` correspondiente a `deolit()`, que a su vez relanza la excepción.

```
void lopart() throws NuevaExcepcion
{
  //
  throw new NuevaExcepcion();
}
void deolit() throws NuevaExcepcion
{
  int i;
  try
  {
    i = -16;
    lopart();
  }
  catch (NuevaExcepcion n)
  {
    System.out.println("Excepción captada, se relanza");
    throw n;
  }
}
public static void main(String []a)
{
  try
  {
    deolit()
  }
  catch(NuevaExcepcion er)
  {
    System.out.println("Excepción capturada en main()");

  }
}
```

16.4.5 Captura de una excepción: `catch`

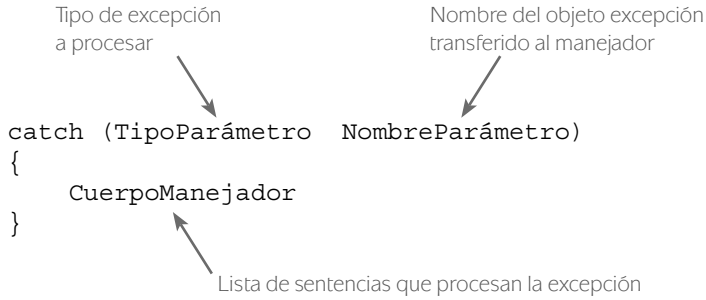
La cláusula de captura constituye el manejador de excepciones; cuando una excepción se lanza desde alguna sentencia de un bloque `try`, se pone en marcha el mecanismo de captura; la excepción será capturada por un `catch` de la lista de cláusulas que siguen al bloque `try`.

Una cláusula `catch` consta de tres elementos: la palabra reservada `catch`, la declaración de un único argumento que será un objeto para manejo de excepciones (declaración de excepciones) y un bloque de sentencias. Si la cláusula `catch` se selecciona para manejar una excepción, se ejecuta el bloque de sentencias.

Formato:

```
catch (argumento)  //captura excepción del tipo del argumento
{
  código del bloque catch
}
```

La especificación del manejador `catch` se asemeja a la definición de un método.



Al contrario que el bloque `try`, `catch` sólo se ejecuta en circunstancias especiales; el argumento es la excepción que puede ser capturada, una referencia a un objeto de una clase ya definida o bien derivada de la clase base `Exception`.

Al igual que con los métodos sobrecargados, el manejador de excepciones se activa sólo si el argumento que se lanza concuerda con la declaración del argumento; se debe tener en cuenta que hay una correspondencia automática entre clase derivada y clase base.

```
catch(Excepcion1 e1) {...}
catch(Excepcion2 e2) {...}
...
catch(Exception e) {...}
```

El último `catch` significa *cualquier excepción* ya que `Exception` es la superclase base de la jerarquía de clases que tratan las anomalías; se puede utilizar como manejador de excepciones predeterminado ya que captura todas las excepciones restantes.

La sintaxis completa de `try` y `catch` permite escribir cualquier cantidad de manejadores de excepciones al mismo nivel.

sintaxis

```
try
{
  sentencias
}
catch (parámetro1)
{
  sentencias
}
catch (parámetro2)
{
  sentencias
}
...
```

Los puntos suspensivos (`...`) indican que puede tener cualquier número de bloques `catch` a continuación del bloque `try`.

Cuando ocurre una excepción en una sentencia durante la ejecución del bloque `try`, el programa comprueba, por orden, cada bloque `catch` hasta que encuentra un manejador cuyo argumento coincide con el tipo de excepción; tan pronto como se encuentra una

PRECAUCIÓN

La clase base de la cual derivan las excepciones es `Exception`; por ello el manejador `catch(Exception e)` captura cualquier excepción lanzada, de utilizarse debe situarse el último bloque `try-catch`.

coincidencia se ejecutan las sentencias del bloque `catch`; cuando se ejecutan las sentencias del manejador, termina el bloque `try` y prosigue la ejecución con la siguiente sentencia; es decir, si el método no ha terminado, la ejecución se reanuda posterior al final de todos los bloques `catch`.

Si no existen manejadores para tratar con una excepción, ésta se relanza para ser tratada en el bloque `try` del método llamador anterior; si no ocurre excepción, las sentencias se ejecutan de modo normal y ninguno de los manejadores es invocado.

EJEMPLO 16.4

Se escriben diversos `catch` para un bloque `try`; las clases que aparecen están declaradas en los diversos paquetes de Java, o se definen por el usuario.

```
try
{
  // sentencias, llamadas a métodos
}
catch(IOException ent)
{
   System.out.println("Entrada incorrecta desde el teclado");
   throw ent;
}
catch(SuperaException at)
{
   System.out.println("Valor supera máximo permitido ");
   return ;
}
catch(ArithmeticException a)
{
   System.out.println("Error aritmético, división por cero... ");
}
catch(Exception et)
{
   System.out.println("Cualquier otra excepción ");
   throw et;
}
```

16.4.6 Cláusula `finally`

En un bloque `try` se ejecutan sentencias de todo tipo, llamadas a métodos, creación de objetos, etcétera; además, ciertas aplicaciones piden recursos al sistema para ser utilizadas; si procesan información, abren archivos para lectura, los cuales se asignarán con uso exclusivo. Es obvio que todos estos recursos tienen que ser liberados cuando el bloque `try`, al que se asignaron, termina; de igual forma, si no se ejecutan normalmente todas las sentencias del bloque por alguna excepción, el `catch` que la captura debe liberar los recursos.

Java proporciona la posibilidad de definir un bloque de sentencias que se ejecutarán siempre, ya sea que termine el bloque `try` normalmente o se produzca una excepción; la cláusula `finally` define un bloque de sentencias con esas características; dicha cláusula es opcional, si está presente debe situarse después del último `catch` del bloque `try`; incluso, se permite que un bloque `try` no tenga `catch` asociados si tiene el bloque definido por la cláusula `finally`.

El esquema siguiente indica cómo se especifica un bloque `try-catch-finally`:

```
try
{
  // sentencias
  // acceso a archivos(uso exclusivo ...)
  // peticiones de recursos
}
catch(Excepcion1 e1) {...}
catch(Excepcion2 e2) {...}
finally
{
  //  sentencias
  //  desbloqueo de archivos
  //  liberación de recursos
}
```

EJEMPLO 16.5

Se declara la excepción `RangoException` que será lanzada en caso de que un valor entero esté comprendido en un intervalo determinado; el bloque `try-catch-finally` ejecuta las sentencias del bloque que define `finally`, esto se pone de manifiesto con una salida por pantalla.

```
import java.io.*;
class ConFinally
{
   static void meteo() throws RangoException
   {
    for (int i= 1; i<9; i++)
    {
        int r;
        r = (int)(Math.random()*77);
        if (r< 3 || r>71)
           throw new RangoException ("con el valor " + r);
     }
     System.out.println("fin método meteo");
   }
   static void gereal()
   {
      for  (int h = 1; h < 9; h++)
       try
       {
          meteo();
       }
       catch(RangoException r)
       {
          System.out.println("Excepción " + r + " capturada");
          System.out.println("Iteración: " + h);
          break;
       }
       finally
       {
         System.out.println("Bloque final de try en gereal()");
       }
    }
```

```
    public static void main(String [] arg)
    {
       try
       {
          System.out.println("Ejecuta main ");
          gereal();
       }
       finally
       {
          System.out.println("Bloque final de try en main()");
       }
       System.out.println("Termina el programa ");
    }
}

class RangoException extends Exception
{
   public RangoException(String msg)
   {
      super(msg);
   }
}
```

La ejecución de este programa da lugar a esta salida:

```
Ejecuta main
Excepción Rango: con el valor 2 capturada
Iteración 1
Bloque final de try en gereal()
Bloque final de try en main()
Termina el programa
```

Observe que al generarse el número aleatorio 2, el método `meteo()` lanza la excepción `RangoException` captada por el `catch` que tiene el argumento `RangoException`, ejecuta las sentencias definidas en su bloque y se sale del bucle; a continuación ejecuta las sentencias del bloque `finally` y devuelve control al método `main()`; acaba el bloque `try` de éste y de nuevo ejecuta el bloque `finally` correspondiente.

16.5 Clases de excepciones definidas en Java

En su empeño por estandarizar el manejo de excepciones Java declara un amplio conjunto de clases de excepciones, las cuales forman una jerarquía en la que la base es `Throwable`, que deriva directamente de la superclase base `Object` (véase figura 16.2).

De `Throwable` derivan dos clases: `Error` y `Exception`; las excepciones del tipo `Error` son generadas por el sistema, se trata de errores irrecuperables y es extraño que se produzcan; por ejemplo: salir de la memoria de la máquina virtual; por tanto, de producirse una excepción `Error` se propagará hasta salir por el método `main()`; los nombres de las subclases que derivan de dicha excepción acaban con el sufijo `Error`, como `InternalError` o `NoClassDefFoundError`.

De `Exception` derivan clases de las que se instancian objetos (excepciones) para ser lanzados; pueden ser capturados por los correspondientes `catch`; las clases derivadas de `Exception` se encuentran en los diversos paquetes de Java, todas tienen como nombre un identificador que indica su finalidad, terminado en `Exception`; la jerarquía de excepciones a partir de esta base es la que se muestra en la figura 16.3.

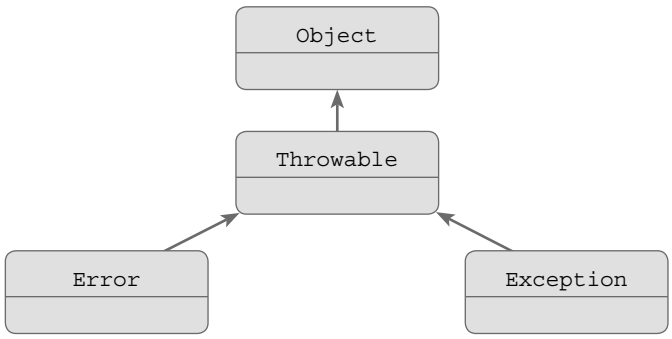


Figura 16.2 Jerarquía de la clase `Object`.

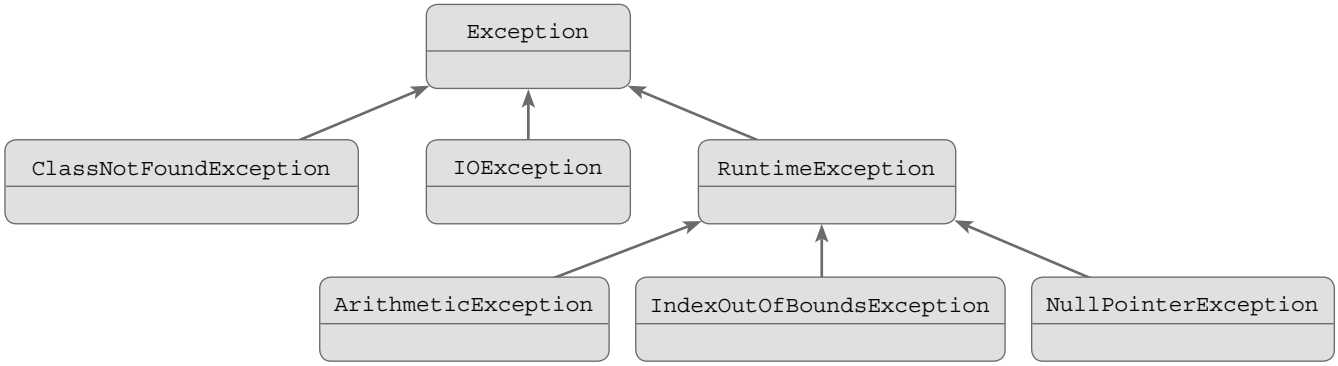


Figura 16.3 Clases derivadas de `Exception`.

16.5.1 `RuntimeException`

Hay excepciones que se propagan automáticamente sin necesidad de especificarlas en la cabecera de los métodos `throw`; esto incluye todas las excepciones del tipo `Runtime Exception`, como división entre 0 o índice fuera de rango; sería tedioso especificar en todos los métodos que se puede propagar una excepción de, por ejemplo, división entre 0; por ello, Java las propaga sin tener que especificar ese hecho.

El método `histog()` de la clase `ConHisto` tiene una expresión aritmética en la que puede producirse una excepción debido a una división entre 0; la excepción se propaga y se capta en `main()`.

```
class ConHisto
{
   static int histog()
   {
      int k, r, z;
      z = 0;  k = 9;
      while (k > 1)
      {
        r = (int)(Math.random()*13);
        System.out.print("r = " + r);
        z += r + (2*r)/(r-k);
        System.out.println("  z = " + z);
        k--;
      }
      return z;
  }
  static public void main(String[] ar)
```

```
  {
     try
     {
        System.out.println("Bloque try. Llamada a histog()");
        histog();
     }
     catch(ArithmeticException a)
     {
        System.out.println("\tCaptura de excepción, " + a);
     }
     catch(RuntimeException r)
     {
        System.out.println("\Captura de excepción, " + r);
     }
  }
}
```

Al depender de los números aleatorios, no siempre se genera excepción; una ejecución da lugar a esta salida:

```
Bloque try. Llamada a histog()
r = 6  z = 2
r = 11 z = 20
r = 4  z = 22
r = 6  Captura de excepción, java.lang.ArithmeticException: /by zero
```

Al tomar `r` y `k` el valor `6` el programa detecta una división entre 0 y lanza la excepción; `histog()` no la captura, por lo que se propaga y alcanza `main()` que sí la captura.

16.5.2 Excepciones comprobadas

Los métodos en los que se lanzan excepciones, directamente con `throw` o porque llaman a otro método que propaga una excepción, deben comprobar la excepción con el apropiado `try-catch`; en caso contrario, la excepción se propaga; esto último exige que en la cabecera del método, con la cláusula `throws` se especifiquen los tipos de excepción que permite propagar, por ejemplo, el método `readLine()` de la clase `BufferedReader` puede lanzar la excepción del tipo `IOException`, en caso de tratarla hay que especificar su propagación; considere este ejemplo:

```
BufferedReader entrada = new BufferedReader(
                     new InputStreamReader(System.in));

int entradaRango()  !! error ¡¡
{
   int d;
   do {
      d = Integer.parseInt(entrada.readLine())
   }while (d<=0 || d>=10);
   return d;
}
```

El método `entradaRango()` tiene un error, error en tiempo de compilación debido a que `readLine()` se implementó con esta cabecera:

```
String readLine() throws IOException
```

Sin embargo, `entradaRango()` no tiene un bloque `try-catch` para tratar la posible excepción, tampoco especifica que se puede propagar lo cual es un error en Java; la forma de evitar el error es especificando la propagación de la excepción:

```
int entradaRango()  throws IOException
```

o capturarla; la propagación puede llegar al método `main()`, ahí se capturaría; también está la opción de no hacerlo y teclear `main() throws IOException`, aunque esto no es una práctica recomendada.

En el ejemplo aparece una llamada al método `parseInt()`, la cual puede generar la excepción `NumberFormatException` del tipo `RuntimeException` que no necesita especificar su propagación.

PRECUCIÓN

Es un error de compilación no capturar las excepciones que un método propaga, o no especificar que se propagará una excepción.

Java permite que no se especifiquen excepciones de la jerarquía que tiene como base `RuntimeException`.

16.5.3 Métodos que informan de la excepción

La clase `Exception` tiene dos constructores: uno sin argumentos (predeterminado) y otro con un argumento que corresponde a una cadena de caracteres.

```
Exception();
Exception(String m);
```

Por lo que se puede lanzar la excepción con una cadena informativa:

```
throw new Exception("Error de lectura de archivo");
```

El método `getMessage()` devuelve una cadena que contiene la descripción de la excepción, es la cadena pasada como argumento al constructor; su prototipo es:

```
String getMessage();
```

A continuación se escribe un ejemplo en el que el método `primero()` llama a `lotes()`, lanza una excepción, se captura y se imprime el mensaje con el que se construyó.

```
void lotes() throws Exception
{
  //...
  throw new Exception("Defecto en el producto.");
}
void primero()
{
  try
  {
    lotes();
  }
  catch (Exception msg)
  {
    System.out.println(msg.getMessage());
  }
}
```

El método `printStackTrace()` resulta útil para conocer la secuencia de llamadas a métodos realizadas hasta llegar al método donde se produce el problema, donde se lanza la excepción; el propio nombre, `printStackTrace()`, indica su finalidad: escribir la cadena con que se inicializa el objeto excepción si no se ha utilizado el constructor predeterminado y a continuación identificar la excepción junto al nombre de los métodos por donde se propagó.

EJEMPLO 16.6

Se va a generar una excepción de división entre cero; ésta no es capturada por lo que se pierde en el método `main()` y termina la ejecución. En pantalla se muestra el trace de llamadas a los métodos que acabó con una excepción `RuntimeException`.

```
class ExcpArt
{
   static void atime()
   {
      int k, r;
      r = (int)Math.random();
      System.out.println("Método atime");
      k = 2/r;
   }
   static void batime()
   {
      System.out.println("Método batime");
      atime();
   }
   static void zatime()
   {
      System.out.println("Método zatime");
      batime();
   }

   static public void main(String[] ar)
   {
      System.out.println("Entrada al programa, main()");
      zatime();
   }
}
```

La ejecución muestra por pantalla:

```
Entrada al programa, main()
Método zatime
Método batime
Método atime
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExcpArt.atime(ExcpArt.java: 8)
at ExcpArt.batime(ExcpArt.java: 13)
at ExcpArt.zatime(ExcpArt.java: 18)
at ExcpArt.main(ExcpArt.java: 24)
```

NOTA

Una excepción que no es capturada por ninguno de los métodos donde se propaga, llegando a la entrada al programa, `main()`, termina la ejecución y hace internamente una llamada a `printStackTrace()` que visualiza la ruta de llamadas a métodos y los números de línea.

16.6 Nuevas clases de excepciones

Las clases de excepciones que define Java pueden ampliarse en las aplicaciones; se pueden definir nuevas para que las aplicaciones tengan control específico de errores; aquellas que se definan tienen que derivar de la clase base `Exception`, o bien directa o indirectamente; por ejemplo:

```
public class MiExcepcion extends Exception { ...}
```

```
public class ArrayExcepcion extends NegativeArraySizeException {...}
```

Es habitual que la clase que se define tenga un constructor con un argumento cadena en el que se puede dar información sobre la anomalía generada; en ese caso, a través de `super` se llama al constructor de la clase base `Exception`.

```
public class MiExcepcion extends Exception
{
  public MiExcepcion(String info)
  {
    super(info);
    System.out.println("Constructor de la clase.");
  }
}
```

Las clases que definen las excepciones pueden tener cualquier tipo de atributo o método que ayude al manejo de la anomalía que representan; la siguiente clase guarda el tamaño que se intenta dar a un arreglo.

```
public class ArrayExcepcion extends NegativeArraySizeException
{
  int n;
  public ArrayExcepcion (String msg, int t)
  {
    super(msg);
    n = t;
  }
  public String informa()
  {
    return getMessage()+ n;
  }
}
```

Se pueden lanzar excepciones de tipo definido en las aplicaciones, por ejemplo:

```
int tam;
int [] v;
tam = entrada.nextInt();
if (tam <= 0)
  throw new ArrayExcepcion ("Tamaño incorrecto.",tam);
```

En aplicaciones complejas que lo requieran se puede definir una jerarquía propia de clases de excepciones, donde la clase base siempre será `Exception`. El siguiente esquema muestra una jerarquía para la aplicación `Cajero`:

```
public class CajeroException extends Exception {...}
public class CuentaException extends CajeroException{...}
public class LibretaException extends CajeroException{...}
public class CuentaCorrienteException extends CuentaException{...}
```

16.7 Especificación de excepciones

La técnica de manejo de excepciones se basa, como se vio anteriormente, en la captura que ocurre al ejecutar las sentencias; una pregunta que surge es: ¿cómo saber cuál es el

tipo de excepción que puede generar un método? Una forma de conocer esto es, naturalmente, leer el código de los métodos, pero en la práctica esto no es posible en métodos que son parte de programas y que además llaman a otros métodos; otra forma es leer la documentación disponible sobre la clase y los métodos, esperando que contenga información sobre los diferentes tipos de excepciones que se pueden generar; desgraciadamente, tal información no siempre se encuentra.

Java ofrece una tercera posibilidad que además, para evitar errores de sintaxis, obliga a utilizar; consiste en declarar en la cabecera de los métodos las excepciones que pueden generar, esto se conoce como especificación de excepciones y hace una lista de las que un método puede lanzar; también garantiza que el método no lance ninguna otra excepción, a no ser las de tipo `RuntimeException` que no es necesario especificar. Los sistemas bien diseñados con manejo de excepciones necesitan definir cuáles métodos lanzan excepciones y cuáles no. Con especificaciones de excepciones, como ya se dijo, se describen exactamente cuáles excepciones, si existen, lanza el método.

El formato de especificación de excepciones es:

```
tipo nombreMetodo(signatura) throws e1, e2, en
{
  cuerpo del metodo
}
```

`e1, e2, en`	Lista separada por comas de nombres de excepciones; especifica que el método puede lanzar directa o indirectamente, incluyendo excepciones derivadas públicamente de estos nombres

En este aspecto, Java distingue dos tipos de excepciones: las que se deben comprobar y las que no; estas últimas se producen en expresiones aritméticas, índices de arreglo fuera de rango o formato de conversión incorrecto.

Java obliga a capturar las excepciones que se deben comprobar con un bloque `try-catch`; o bien, a su propagación al método llamador y para ello es necesario especificar en la cabecera del método que puede propagar la excepción con la cláusula `throws` seguida del nombre de la excepción, en caso contrario se produce un error de sintaxis o de compilación; todas las excepciones que el programador define se deben comprobar; en general, todas las excepciones, descartando `RuntimeException` y las de tipo `Error`, son las que se deben comprobar.

En cuanto a sintaxis, una especificación de excepciones es parte de una definición de un método cuyo formato es:

```
tipo  nombreMetodo(lista arg,s) throws lista excepciones
```

Por ejemplo:

```
void metodof(argumentos) throws T1, T2
{
  ...
}
```

Especificación de excepciones

REGLA

Una especificación de excepciones sigue a la lista de argumentos del método; se especifica con la palabra reservada `throws` seguida por una lista de tipos de excepciones, separadas por comas.

REGLA

En la especificación de excepciones se sigue la regla en cuanto a conversión automática de tipo clase derivada a clase base de forma que pueda expresar que se puede propagar o lanzar cualquier excepción:

```
tipo nombreMetodo(arg,s) throws Exception
```

Al ser `Exception` la clase base de todas las excepciones a comprobar.

Por ejemplo:

```
// Clase pila con especificaciones de excepciones
class PilaTest
{
  // ...
  public void quitar (int valor) throws EnPilaVaciaException{...};
  public void meter (int valor) throws EnPilaLlenaException {...}
  // ...
}
```

ejercicio 16.1

Cálculo de las raíces o soluciones de una ecuación de segundo grado.

Una ecuación de segundo grado $ax^2 + bx + c = 0$ tiene dos raíces:

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Los casos de indefinición son: 1) $a = 0$; 2) $b^2 - 4ac < 0$, que no producen raíces reales, sino imaginarias; en consecuencia, se consideran dos excepciones:

`NoRaizRealException` y `CoefAceroException`.

La clase `EcuacionSegGrado` tiene como atributos `a`, `b` y `c`, que son coeficientes de la ecuación; además, `r1`, `r2` son las raíces; el método `raices()` las calcula, su declaración es:

```
void raices() throws NoRaizRealException, CoefAceroException
```

El cuerpo del método es:

```
void raices() throws NoRaizRealException, CoefAceroException
{
  double discr;
  if(b*b < 4*a*c)
    throw new NoRaizRealException("Discriminante negativo",a,b,c);
  if(a == 0)
    throw new CoefAceroException("No ecuaciones De segundo grado
                                 ",a,b,c);
  discr = Math.sqrt(b*b - 4*a*c);
  r1 = (-b - discr) / (2*a);
  r2 = (-b + discr) / (2*a);
}
```

`raices()` lanza excepciones si no tiene raíces reales o si el primer coeficiente es cero; el método no captura excepciones sino que se propagan; es necesaria la cláusula `throws` con los tipos `NoRaizRealException`, `CoefAceroException`.

El método desde el que se llame a `raices()` debe tener un bloque `try-catch` para capturar las excepciones lanzadas o se propagarán, lo que exige de nuevo la cláusula `throws`.

La siguiente aplicación define las clases con las excepciones antes mencionadas; en el método `main()` se piden los coeficientes de la ecuación, se crea el objeto, se llama al método de cálculo y se escribe en pantalla.

```
import java.util.*;
// representa la excepción: ecuación no tiene solución real
class NoRaizRealException extends Exception
{
  private double a,b,c;
  public NoRaizRealException(String m, double a, double b, double c)
  {
    super(m);
    this.a = a;
    this.b = b;
    this.c = a;
  }
  public String getMessage()
  {
    return  "Para los coeficientes "+(float)a +", " +
        (float)b + ", " +(float)c +super.getMessage();
  }
}
// representa la excepción: no es ecuación de segundo grado
class CoefAceroException extends Exception
{

  public CoefAceroException(String m)
  {
    super(m);
  }
}
// clase para representar cualquier ecuación de segundo grado
class RaiceSegGrado
{
  private double a,b,c;
  private double r1,r2;
  public RaiceSegGrado(double a, double b, double c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
  }
  public  void raices() throws NoRaizRealException, CoefAceroException
  {
    ...
  }
  public  void escribir()
  {
    System.out.println("Raices de la ecuaci¢n; r1 = "
                                + (float)r1 + "  r2 = " + (float)r2);
  }
}
// clase principal
```

```
public class Raices
{
  public static void main(String [] ar)
  {
    RaiceSegGrado rc;
    double a,b,c;
    Scanner entrada = new Scanner(System.in);
    // entrada de coeficientes de la ecuación
    System.out.println("Coeficientes de ecuación de segundo grado");
    System.out.println(" a = ");
    a = entrada.nextDouble();
    System.out.println(" b = ");
    b = entrada.nextDouble ();
    System.out.print(" c = ");
    c = entrada.nextDouble();
    // crea objeto ecuación y bloque para captura de excepciones
    try
    {
      rc = new RaiceSegGrado(a,b,c);
      rc.raices();
      rc.escribir();
    }
    catch(NoRaizRealException er)
    {
      System.out.println(er.getMessage());
    }
    catch(CoeficienteAcero er)
    {
      System.out.println(er.getMessage());
    }
  }
}
```

Se puede observar que en el constructor de la clase `NoRaizRealException` se hace una llamada al constructor de la clase base, `Exception`, para pasar la cadena; también se redefinió el método `getMessage()`, de tal forma que devuelve la cadena con que se inicializa al objeto excepción concatenada con los coeficientes de la ecuación de segundo grado.

resumen

- Las excepciones son, normalmente, condiciones o situaciones de error imprevistas; por lo general terminan el programa del usuario con un mensaje de error proporcionado por el sistema; por ejemplo: división entre cero, índices fuera de límites en un arreglo, etcétera.
- Java posee un mecanismo para manejar excepciones, las cuales son objetos de clases de una jerarquía proporcionada por el lenguaje; el programador puede definir sus clases de excepciones y tienen que derivar, directa o indirectamente de `Exception`.
- El código Java puede levantar (*raise*) una excepción utilizando la expresión `throw`; ésta se maneja invocando un manejador de excepciones seleccionado de una lista que se encuentra al final del bloque `try` del manejador.

- El formato de sintaxis de `throw` es:

```
throw objetoExcepcion ;
```

  El cual lanza una excepción que es un objeto de una clase de manejo de excepciones.
- El formato de sintaxis de un bloque `try` es:

```
try
sentencia compuesta
lista de manejadores
```

  El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada; el orden en el que son definidos los manejadores determina la secuencia en la que un manejador de una excepción levantada va a ser invocado.
- La sintaxis de un manejador es:

```
catch(argumento formal)
sentencia compuesta
```

- El manejador `finally` es opcional; de utilizarse se escribe después del último `catch`; su característica principal es que siempre se ejecuta la sentencia compuesta especificada a continuación de `finally`, una vez que termina la última sentencia del bloque `try` o a continuación del `catch` que captura una excepción.
- La especificación de excepciones es parte de la declaración de un método y tiene el formato:

```
cabecera_método throws lista_excepciones
```

- Java define una jerarquía de clases de excepciones; `Throwable` es la superclase base, aunque `Exception` que deriva directamente de la anterior, es la clase base de las excepciones manejadas.
- Las clases definidas por el programador para el tratamiento de anomalías tienen que derivar directa o indirectamente de la clase `Exception`.

```
class MiException extends Exception {...}
class OtraException extends Miexception {...}
```

- Una excepción lanzada en un método debe ser capturada en el método o propagarse al llamador, en cuyo caso es necesario especificar la excepción en la cabecera del método (`throws`).

conceptos clave

- Captura de excepciones.
- `catch`.
- Especificación de excepciones.
- Excepción
- Excepciones estándar.
- `finally()`.
- Lanzamiento de excepciones.
- Levantar una excepción.
- Manejador de excepciones.
- Manejo de excepciones.
- `throw`.
- `try`.

ejercicios

16.1 El siguiente programa maneja un algoritmo de ordenación básico pero no funciona bien. Situar declaraciones en el código del programa de modo que se compruebe si funciona correctamente; corregir el programa.

```
void intercambio (int x, int y)
{

  int aux;
  aux=x;
  x=y;
  y=aux;
}
void ordenar (int []v, int n)
{
  int i, j;
    for (i=0; i< n; ++i)
      for (j=i; j< n; ++j)
          if (v[j] < v[j+1]])
            intercambio (v[j], v[j+1]);
}
static public void main(String[]ar)
{
  int z[]={14,13,8,7,6,12,11,10,9,-5,1,5};

  ordenar (z, 12);
  for (int i=0 ; i<12; ++i)
    System.out.print(z[i] + " ");

}
```

16.2 Escribir el código de una clase Java que lance excepciones para cuantas condiciones estime convenientes; utilizar una cláusula `catch` que emplee una sentencia `switch` para seleccionar un mensaje apropiado y terminar el cálculo; utilizar una jerarquía de clase para listar las condiciones de error.

16.3 Escribir el código de un método en el cual se defina un bloque `try` y dos manejadores `catch`, en uno de los cuales se relanza la excepción; incluir un manejador `finally` para lanzar una excepción; determinar qué ocurre con la excepción que relanza el `catch`, mediante un programa en el que se genere la excepción que es captada por el `catch` descrito.

16.4 Escribir el código de una clase para tratar el error que se produce cuando el argumento de un logaritmo neperiano es negativo; el constructor de la clase debe tener como argumento una cadena y el valor que ha generado el error.

16.5 Escribir un programa Java en el que se genere la excepción del ejercicio anterior y se capture.

16.6 Definir una clase para tratar los errores en el manejo de cadenas de caracteres; definir una subclase para tratar el error supuesto de cadenas de longitud mayor de 30 caracteres y otra que maneje los errores de cadenas que tienen caracteres no alfabéticos.

16.7 Escribir un programa en el que se dé entrada a cadenas de caracteres y se capturen excepciones del tipo mencionado en el ejercicio anterior.