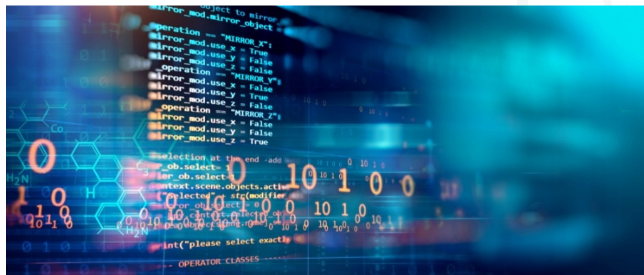
Excepciones

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

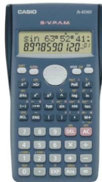
Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-20

Motivación

Problema

Implementa
una función que dada una expresión matemática guardada en una cadena de texto devuelva el valor resultante de evaluar dicha expresión matemática. Las operaciones binarias permitidas son sumas (+), restas (–), multiplicaciones (*), divisiones (/) y potencias (^), entre números reales.
Los operadores
se aplican siempre de izquierda a derecha (no hay reglas de precedencia por operador) pero para cambiarlo se pueden aplicar paréntesis (,).




```
double calcular(const std::string& expresion);
```

Puede que necesites usar estas funciones:

```
double stod (const string& str, size_t* idx = 0);
string string::substr(size_t pos = 0, size_t len = npos) const;
```

Problema

Diseña una función que permita resolver una ecuación de segundo grado.

Problema

Usando lo anterior diseña una función que, dada el área y el perímetro de un rectángulo, calcule sus dimensiones.

Problema

Haz un programa principal que, usando la función anterior le pida al usuario el área y el perímetro de un rectángulo y muestre por pantalla sus lados.

Motivación

Ningún programa es perfecto, por lo cual podemos considerar que en algún momento de su funcionamiento puede fallar.

 Motivación

Los fallos de un programa pueden categorizarse de dos formas:

- fallos debidos a errores de diseño o de implementación (*bugs*).
- fallos debidos al uso normal del programa en situaciones inesperadas o con datos anómalos.

Como por ejemplo:

- accesso a un elemento inexistente de un vector
- intento de acceso en escritura a un fichero de sólo lectura
- cáculos imposibles o indeterminados

 Motivación

El programador debe evitar los bugs y eliminarlos en una fase de depuración, pero no puede blindar todo el código frente a ese tipo de errores (excepto en situaciones críticas).

El resto de errores si que pueden ser tenidos en cuenta en el diseño del programa.

Se dice que un código es **robusto** si tiene en cuenta esos posibles errores de ejecución y está preparado para actuar en consecuencia, más allá de la interrupción incontrolada del programa.

Ejemplo 1

```
int[] v = new int[16];

for (int i=0; i<20; i++)
  total = total + v[i];
```

Ejemplo 2

```
FILE* f = fopen("....","rw");
fprintf(f,"%f",3.14);
```

 Motivación

El control de errores puede realizarse mediante códigos de retorno.

```
1 if (do_something() < 0)
2     ...
```

```
1 FILE* f = fopen("....","rw");
2 if (f != NULL)
3   fprintf(f,"%f",3.14);
```

Pregunta

¿Qué problema tiene esto?

Motivación

```
void h() {
  if (something fails) return ....
}
void g() { h(); }
void f() { g(); }
void run() {
  f();
}
```

Propagar el error hace el código ilegible.

Más difícil todavía (Java):

```
1 T get(int i)
2 {
3   if (i está en los límites correctos)
4     return data[i]
5   else
6     ??????
7 }
```

Podriamos devolver **null**...

Motivación

O incluso imposible (C++):

```
1  T operator[] (int i)
2  {
3    if (i está en los límites correctos)
4      return data[i]
5    else
6      ??????
7  }
```

Las excepciones permiten separar el control de errores del diseño funcional del programa, sin obligar a modificar la estructura de clases y métodos para integrar ese control.

Permite modificar el flujo normal del programa en caso de errores y centralizar la gestión de los mismos.

Objetivos

Objetivos

Los objetivos de este tema son:

- Presentar la Gestión de Excepciones como una herramienta más de programación.
- Presentar la forma habitual de un lenguaje de soportar las excepciones.
- Presentar la implementación C++ y Java de las excepciones
- Destacar las ventajas de las excepciones de C++ y Java frente a otras implementaciones.

Gestión de Excepciones

Excepción

Ocurrencia de una **condición excepcional** durante la ejecución de un programa. El programa puede **señalar** la ocurrencia de esa situación, de forma que se **transfiera el control** a otro punto del programa, saltándose las reglas habituales de las estructuras de control estándar de la Programación Estructurada u Orientada a Objetos.

La señalización de la excepción y la definición del código a ejecutar (en su caso) se definen mediante construcciones especiales del lenguaje.

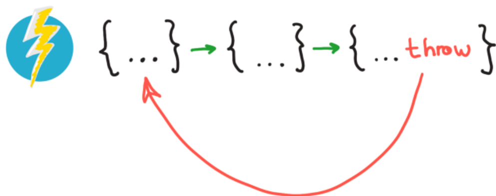
El código que marca la excepción puede pasar información al código que se va a ejecutar mediante algún tipo de valor o estructura de datos.
Por extensión también se denomina a esa estructura de datos una *excepción*.

Señalización

Indicación de que se ha producido una excepción, acompañada de la construcción de la estructura de datos con información y su paso al *runtime* del lenguaje para su gestión.

Propagación

El aviso y la información sobre la excepción se pasan al *bloque* en el que se produce. Si no es gestionada, se propaga al bloque superior.
Normalmente los bloques coinciden con llamadas sucesivas entre funciones.



Captura y **Gestión**
Reconocimiento de la existencia de la excepción y ejecución del código correspondiente, usando la información contenida en la propia excepción.

El tipo de información que se puede incluir en la excepción depende del lenguaje:

- valores simples enteros o enumerados
- valores de un tipo predefinido (ADA)
- objetos de una clase específica (Java)
- cualquiera (C++)

ADA

```
declare
  MyException: exception;
begin
  if ...
    raise MyException with "Something broke !!!";
exception
  when Constraint_Error =>
      ... ;
  when Storage_Error =>
      ... ;
  when xcp: MyException =>
      Put(Exception_Message(xcp));
  when others =>
      ... ;
end
```

C++/Java

- **Señalización**

```
1   if (...) throw ...;
```

- **Propagación** automática siguiendo ciertas reglas.
- **Captura**: bloque *try–catch*

```
1   try {
2       // Código susceptible de fallo
3   }
4   catch (...) {
5       // Código de gestión de la excepción
6   }
```

C++

```cpp
void this_can_fail()
{
  if (...)
    throw 20;
  ...
}
```

```cpp
try {
  this_can_fail();
}
catch (int xcp) {
  switch (xcp) {
    case 20:
           ...
    default:
           ...
  }
}
```

El paso de un valor escalar permite identificar la excepción, pero muchas veces es insuficiente.

Los lenguajes orientados a objeto permiten crear un objeto de alguna clase y rellenarlo con información.

También se puede pasar una excepción que sea un **objeto**:

```
1   class NegativeSQRT {
2     float  value;
3   };
4
5   try {
6      if (x<0) throw NegativeSQRT(x);
7      y = sqrt(x);
8      ...
9   }
10  catch (NegativeSQRT xcp) {
11     cerr << "you tried to sqrt " << xcp.x << endl;
12  }
```

El tipo de objeto que se puede utilizar depende del lenguaje.

Java: El objeto creado debe ser de un tipo especial, definido mediante herencia de una clase determinada.
Como ventaja ofrece infraestructura adicional de forma automática, implementada en la clase base.

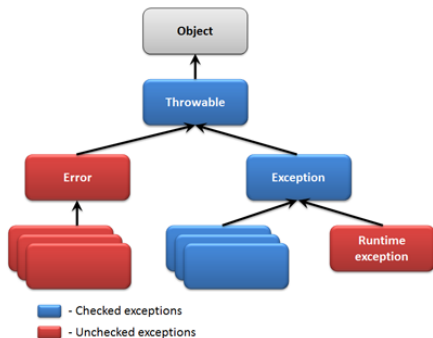
La distinción entre distintas excepciones se implementa mediante varios bloques **catch**, basados en la herencia.

El tipo de objeto que se puede utilizar depende del lenguaje.

C++: El objeto creado puede ser de cualquier tipo, pero existen algunas clases estándar que nos ofrecen ciertas facilidades.

La distinción entre distintas excepciones se implementa mediante varios bloques **catch**, basados en la herencia.

Las excepciones en Java **deben** derivarse de la clase base `Exception`:



Se implementa mediante un bloque `try-catch`:

```
1  try {
2    // Codigo que puede generar una excepcion
3    if (...)
4      throw new Exception();
5  }
6  catch (Exception e) {
7    // Bloque de gestion de la excepcion
8  }
```

Podemos usar directamente la clase Exception:

```
1  class main {
2    public static void main(String[] args) {
3      try {
4        int    x;
5        int[]  v = new int[2];
6        double r = Math.random();
7        if (r<0.333) { x    = 1/0; }
8        if (r>0.666) { v[4] = 1;   }
9        System.out.println("OK");
10     }
11     catch (Exception e) {
12       System.out.println(e);
13     }
14     System.out.println("Done");
15   }
16 }
```

Podemos usar directamente la clase `Exception`:

```
:~> java main
OK
Done

:~> java main
java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 2
Done

:~> java main
java.lang.ArithmeticException: / by zero
Done
```

¿Cómo podemos diferenciar excepciones ?

Especializando las excepciones mediante clases derivadas de `Exception` y varios bloques de captura:

```
try {
    ...
}
catch (ArithmeticException e) {
    ...
}
catch (ArrayIndexOutOfBoundsException e) {
    ...
}
```

Mismas reglas que la sobrecarga de métodos.

Podemos definir nuevas excepciones: problemas en el motor...

```
1  abstract class EngineException extends Exception
2  {
3  }
```

Podemos definir nuevas excepciones: problemas en el motor...

```
1   class LowFuelException extends EngineException
2   {
3     private float level; // in %
4     public LowFuelException(float level) {
5       this.level = level;
6     }
7     public int getLevel( ) {
8       return level;
9     }
10  }
```

Podemos definir nuevas excepciones: problemas en el motor...

```
1   abstract class TemperatureException extends EngineException
2   {
3     private int temperature; // in Celsius
4     public TemperatureException(int temperature) {
5       this.temperature = temperature;
6     }
7     public int getTemperature() {
8       return temperature;
9     }
10  }
```

Podemos definir nuevas excepciones: problemas en el motor...

```
1   class TooColdException extends TemperatureException {
2     public TooColdException(int temperature) {
3      super(temperature);
4     }
5   }
6
7   class TooHotException extends TemperatureException {
8     public TooHotException(int temperature) {
9      super(temperature);
10    }
11  }
```

Nuestro código puede generar distintas excepciones...

```
1   class Engine {
2     public void run() {
3       if (fuelLevel<10) throw LowFuelException(fuelLevel);
4       if (temp<0) throw TooColdException(temp);
5       start();
6       while(...) {
7         if (temp>100) throw TooHotException(temp);
8         if (fuelLevel<10) throw LowFuelException(fuelLevel);
9         ...
10      }
11    }
12  }
```

Y capturar cada una de ellas:

```
1   try {
2     engine.run();
3   }
4   catch (TooHotException e) {
5     ...
6   }
7   catch (TooColdException e) {
8     ...
9   }
10  catch (LowFuelException e) {
11    ...
12  }
```

Se puede usar una sintaxis compuesta:

```
1  try {
2    engine.run();
3  }
4  catch (TooColdException|TooHotException e) {
5    ...
6  }
7  catch (LowFuelException e) {
8    ...
9  }
```

Se puede hacer una gestión jerárquica de las excepciones:

```java
try {
  engine.run( );
}
catch (TemperatureException e)
{ ... }
catch (LowFuelException e)
{ ... }
catch (EngineException e)
{ ... }
```

Se puede hacer una gestión jerárquica de las excepciones:

```
1  try {
2    engine.run();
3  }
4  catch (TemperatureException e)
5  { ... }
6  catch (LowFuelException e)
7  { ... }
8  catch (EngineException e)
9  { ... }
```

¡¡ Cuidado con el orden !!

Se puede hacer una gestión jerárquica de las excepciones:

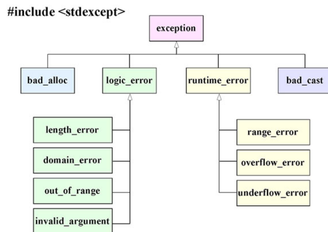
```
1   try {
2     engine.run();
3   }
4   catch (EngineException e)
5   { ... }
6   catch (TemperatureException e)
7   { ... }
8   catch (LowFuelException e)
9   { ... }
```

¡¡ Cuidado con el orden !!

El tipo de datos usado en la excepción puede ser cualquiera, pero la biblioteca estándar nos ofrece algunos tipos predeterminados:

```
#include <exception>
class exception
{
public:
  exception() noexcept;
  exception(const exception&) noexcept;
  exception& operator=(const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
}
```

El tipo de datos usado en la excepción puede ser cualquiera, pero la biblioteca estándar nos ofrece algunos tipos predeterminados:



Las clases **logic_error** y **runtime_error** permiten definir en su constructor el mensaje devuelto por **what()**.

Se implementa mediante un bloque try–catch:

- La excepción se se lanza por valor
- Y se captura por referencia

```
1  try
2  {
3    if (...)
4      throw logic_error("schade...");
5  }
6  catch (exception& e) {
7    cout << "exception: " << e.what() << endl;
8  }
```

La captura por referencia permite usar el polimorfismo:

```
1   try {
2     if (...) throw logic_error("error-1");
3     if (...) throw runtime_error("error-2");
4     ...
5   }
6   catch (exception& e) {
7     cout << "problem: " << e.what() << endl;
8   }
```

Mismas reglas que la sobrecarga de métodos.

La captura por referencia permite usar el polimorfismo y la herencia:

```
1   try {
2     if (...) throw logic_error("error-1");
3     if (...) throw runtime_error("error-2");
4     ...
5   }
6   catch (logic_error& e) {
7     cout << "logic problem: " << e.what() << endl;
8   }
9   catch (runtime_error& e) {
10    cout << "runtime problem: " << e.what() << endl;
11  }
12  catch (exception& e) {
13    cout << "generic problem: " << e.what() << endl;
14  }
```

Podemos definir nuevas excepciones:

```
class engine_error : public exception {
  const char* what() const noexcept override {
    return "Engine Error";
  }
};
```

O bien:

```
class engine_error : public runtime_error {
  engine_error(const string& msg) : runtime_error(msg)
  {}
};
```

Podemos definir nuevas excepciones:

```
1  class lowfuel_error : public engine_error
2  {
3    float level;
4    lowfuel_error(float l)
5          : engine_error("lowfuel"), level(l)
6    {}
7  }
```

Y capturar cada una de ellas:

```
1  try {
2      engine.run();
3  }
4  catch (toohot_error& e) {
5      ...
6  }
7  catch (toocold_error& e) {
8      ...
9  }
10 catch (engine_error& e) {
11     ...
12 }
```

Propagación de excepciones

Además de solventar el error que ha provocado el lanzamiento de una excepción, en los bloques de gestión podemos operar con la propia excepción.

¿ Qué podemos hacer en el bloque catch con la excepción ?

¿ Qué podemos hacer en el bloque catch con la excepción ?

a) Ignorarla: se propaga automáticamente.

¿ Qué podemos hacer en el bloque catch con la excepción ?

b) Capturarla y cortarla:

```cpp
// C++
try {
  engine.run();
}
catch (engine_error& e) {
  cout << e.what() << endl;
}
```

```java
// Java
try {
  engine.run();
}
catch (EngineException e) {
  System.out.println(e.getMessage());
}
```

¿ Qué podemos hacer en el bloque catch con la excepción ?

c) Capturarla y propagarla de forma manual:

```cpp
// C++
try {
  engine.run();
}
catch (engine_error& e) {
  cout << e.what() << endl;
  throw;
}
```

```java
// Java
try {
  engine.run();
}
catch (EngineException e) {
  System.out.println(e.getMessage());
  throw e;
}
```

¿ Qué podemos hacer en el bloque catch con la excepción ?

d) Capturarla y cambiarla:

```cpp
// C++
try {
  engine.run();
}
catch (engine_error& e) {
  cout << e.what() << endl;
  throw runtime_error("engine");
}
```

```java
// Java
try {
  engine.run();
}
catch (EngineException e) {
  System.out.println(e.getMessage());
  throw new OutOfResources();
}
```

¿ Qué podemos hacer en el bloque catch con la excepción ?

e) Capturarla y encadenarla: Java

```
1   try {
2     engine.run( );
3       // Lanza LowFuelException( )
4   }
5   catch (LowFuelException e) {
6     // La convierte en una más estándar
7     throw new OutOfResources(e);
8   }
```

Luego se puede usar `Exception.getCause( )`...

¿ Qué podemos hacer en el bloque catch con la excepción ?

e) Capturarla y encadenarla: C++

```
1 try {
2   engine.run( );
3     // Lanza lowfuel_error( )
4 }
5 catch (lowfuel_error& e) {
6   // La convierte en una más estándar
7   throw_with_nested(logic_error("second"));
8 }
```

Luego se puede usar `rethrow_if_nested( )`...

¿ Qué ocurre en este caso si salta una `IOException` ?

```
1   try {
2     BufferedReader reader =
3       new BufferedReader(new FileReader(filename));
4     // hacemos cosas con el fichero...
5     reader.close();
6   }
7   catch (IOException e) {
8     System.out.println("no se leer");
9     reader.close();
10  }
```

¿Y si salta otra excepción cualquiera ?

```java
try {
  BufferedReader reader =
    new BufferedReader(new FileReader(filename));
  // hacemos cosas con el fichero...
}
catch (IOException e) {
  System.out.println("no se leer");
}
finally {
  reader.close();
}
```

La sección `finally` se ejecuta siempre que salgamos del bloque `try-catch`, por la razón que sea.

Para usos sencillos se puede usar la sentencia try y definir un recurso que se cerrará automáticamente:

```
1  try (BufferedReader reader =
2          new BufferedReader(new FileReader(filename))) {
3      // hacemos cosas con el fichero...
4  }
5  catch (IOException e) {
6      System.out.println("no se leer");
7  }
```

El recurso (objeto) debe implementar el interfaz `java.io.AutoCloseable`:

```
void close();
```

RAII vs finally

C++ no ofrece un bloque **finally**, porque se usa un concepto denominado **RAII** (*Resource Acquisition Is Initialization*).

```
mutex mtx;
...
try {
  mtx.lock();
  if (....) throw runtime_error("uh oh ...");
  mtx.unlock();
}
catch (...) {
  // mtx ???
}
```

RAII vs finally

C++ no ofrece un bloque **finally**, porque se usa un concepto denominado **RAII** (*Resource Acquisition Is Initialization*).

La gestión de excepciones garantiza que se destruyen todos los objetos ordenadamente durante el *stack unwinding*.
Es responsabilidad del destructor ”hacer limpieza”.

RAII vs finally

C++ no ofrece un bloque **finally**, porque se usa un concepto denominado **RAII** (*Resource Acquisition Is Initialization*).

```
1  mutex mtx;
2  ...
3  try {
4    lock_guard<mutex> lock(mtx); // mtx.lock()
5    if (....) throw runtime_error("uh oh ...");
6    // ~lock_guard() -> mtx.unlock()
7  }
8  catch(...) {
9    // mtx OK
10 }
```

Verificación de Excepciones

Verificación de Excepciones

En Java se utilizan las excepciones de forma extensiva tanto para señalar errores normales (*file–not–found*), como para problemas más serios (índice fuera de rango en un vector).

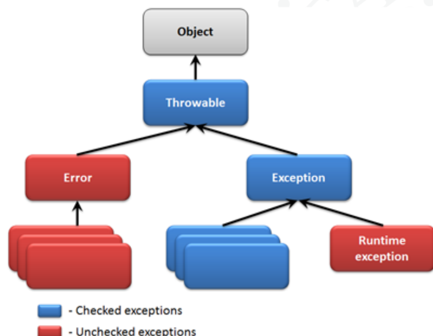
Los errores del segundo caso deberían eliminarse y se supone que en un programa correcto no deberíamos preocuparnos por ellos, podrían ser ignorados en los bloques *catch* de gestión (porque **no existen**).

En Java se utilizan las excepciones de forma extensiva tanto para señalar errores normales (*file–not–found*), como para problemas más serios (índice fuera de rango en un vector).

Los errores (avisos ?) del primer tipo son parte del interfaz de programación de un módulo y siempre hay que estar pendientes de ellos y tenerlos en cuenta. Nunca deberían ser ignorados.

Por ello Java distingue varios tipos de excepciones:

- *checked* (verificadas): `Exception`
- *unchecked*: `RuntimeException`
- *unchecked* de sistema: `Error`



Las excepciones *checked* (verificadas) no pueden ignorarse, y si no se capturan hay que indicar explícitamente que se van a propagar mediante la cláusula *throws*:

```
1 // esto puede pasar, pero lo ignoramos explicitamente
2 // que se encargue alguien por encima...
3 void do_something()
4        throws IOException, ArrayIndexOutOfBoundsException
5 { ... }
```

Si tenemos código con excepciones verificadas y no se capturan ni se explicita su propagación, tenemos un error de compilación.

Las excepciones *unchecked* se usan a veces para no ”molestar” al usuario de la biblioteca obligándole a capturar las excepciones.

Pero las excepciones son un interfaz más de la especificación de una función, igual que el valor devuelto o el número y tipo de los parámetros.

Ideas:

- **Exception** (*checked*): si el problema puede ocurrir sólo en ciertos puntos concretos del código, y es fácil recuperarse del error.
- **RuntimeException** (*unchecked*): el problema no es localizado y no es fácil o posible recobrarse.

En C++ todas las excepciones son *unchecked* (no es obligatorio comprobarlas).

Existe una palabra clave `nothrows` que especifica que una función (o método) nunca lanza excepciones (permite ciertas optimizaciones).

Más información

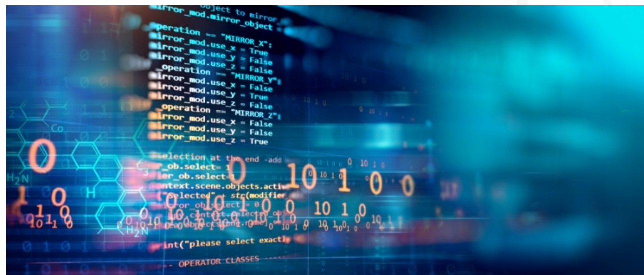
Más información

http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

http://en.wikibooks.org/wiki/C++_Programming/Exception_Handling

Excepciones

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza
1542

2023-01-20