Lecture 09

Object-Oriented Concurrent Programming

Error Handling in Concurrency

Object-Oriented Concurrent Programming, 2019-2020

v2.2, 20-11-2019

Miguel Oliveira e Silva
DETI, Universidade de Aveiro

Java: Exceptions and Threads

Java: Exceptions and Threads

- Exceptions are used in Java to handle faults in programs.
- However, this mechanism is defined to work only inside each thread.
- As presented in lecture 2, Java supports thread interruption using a particular exception type: `InterruptedException`.
- Class `Thread` provides three methods to implement this behavior:

```
package java.lang;
public class Thread implements Runnable {
   ...
   /** interrupt thread attached to object.
     */
   public void interrupt();

   /** thread attached to object interrupted?
     * (does not clear interrupted status flag) */
   public boolean isInterrupted();

   /** current thread interrupted?
     * (clears interrupted status flag)? */
   public static boolean interrupted();
}
```

Java: Exceptions and Threads (2)

- Interrupting a thread without its collaboration is an insecure operation (race condition), thus the algorithm for this task works in a volunteer and collaborative way.
- To interrupt a thread the method `interrupt` of its `Thread` object should be invoked.
- As a result of this invocation, its interruption flag is activated.
- If the thread is blocked in an interruptible native service (`wait`, `sleep`, cancellation point) a `InterruptedException` will be throwed; otherwise the thread continues its execution until such a service is invoked.
- To implement a cancellation point it is enough to check the interruption flag, preferably through method `isInterrupted`, and act accordingly.
- By default, Java opts for ignoring when threads are terminated with an exception (a stack trace will occur, but the remaining threads continue their execution).
- This behavior is almost always the wrong choice!

Library `pt.ua.concurrent`

- Library `pt.ua.concurrent` allows the definition of a thread termination policy with four alternatives:
  - **DEBUG**: A thread failure causes a stack dump to the console and the termination of the whole program (default behavior).
  - **IGNORE**: A thread failure is ignored for the whole program.
  - **IGNORE_DEBUG**: A thread failure causes a stack dumped to console, but thread failure is ignored for the whole program.
  - **PROPAGATE**: A thread failure causes the propagation of the failure to the "parent" thread (through the interrupt service).
- (See example `TestTerminationPolicy.java` for more details.)

Systematic Approaches for Error Handling

Systematic Approaches for Error Handling

- Programming without a proper error handling mechanism is absurd (at least from an Engineering point of view).
- Errors are inherent to problem solving; hence, also to programming.
- Programming methodologies should take errors as an inevitable part of the process, and as such, should incorporate error handling as one of its strategies.
- Errors may trigger faults in program entities; those faults may launch exceptions; which in turn may be handled for fault tolerance or a controlled program termination .
- Errors can have many origins: memory exhaustion, null reference, division by zero, fault in a storage device, network connection lost, false assertion, etc..

Systematic Approaches for Error Handling (2)

- Nevertheless, as diverse as origins might be, errors can be classified by its “ignition point”: internal and external errors.
- Internal errors are errors entirely of the responsibility the program. All other errors are external.
- False assertions, null references, division by zero, and all other logical errors of the program belong to the group of internal errors. Network connection failures, memory exhaustion, etc., are external errors.

Systematic Approaches for Error Handling (3)

- Although at first sight this distinction may appear artificial, developing reliable programs should use different methodologies to handle this two groups of errors.
- There are only two systematic approaches to handle errors in programs: Design by Contract (DbC) and Defensive Programming (DP).
- We Will show that Design by Contract is the best methodology for internal errors, and Defensive Programming for external errors.
- Internal errors are, by far, what primarily compromises the quality of software, and searching and eliminating those errors (BUGs) is typically where programmer lose most of their time (and patience).
- Program debugging for internal errors is not simply test its execution for detecting errors; it is also necessary the search for the maximum number of errors, its precise location in source code, and a clear distribution of responsibilities for its proper rectification.
- Design by Contract is unbeatable in all these aspects!

Error Handling in Modules

There are basically three possibilities:

- **Ostrich technique**:
  - Ignore the problem.
- **Defensive Programming**:
  - Accept all situations, have specific code to detect and deal with errors.
- **Design by Contract**:
  - Attach explicit contracts to the module;
  - The module has only to comply with its contract part;
  - Define contracts as assertions; executable assertions allow error detection in runtime (a false non-concurrent assertion is unambiguously considered a program error).

Ostrich technique

possible faults

```
public class Date {

  public Date(int day,int month,int year) {
    d = day; m = month; y = year;
  }

  public static int monthDays(int month,int year) {
    final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
    int result = days[m-1];
    if (month == 2 && leapYear(year))
      result++;
    return result;
  }

  protected int d,m,y;
}
```

```
public void main(String[] args) {
  Date d = new Date(25,4,1974);
  ...
  if (Date.monthDays(month,year) != 31)
     ...
}
```

Defensive Programming (approach #1)

```
public class Date {

  public Date(int day,int month,int year) {
    if (!valid(day,month,year))
      error = true;
    else {
      error = false;
      d = day; m = month; y = year;
    }
  }

  public static int monthDays(int month,int year) {
    int result;

    if (!validMonth(month))
      result = -1;
    else {
      final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
      result = days[m-1];
      if (month == 2 && leapYear(year))
        result++;
    }
    return result;
  }

  public boolean error() { return error; }
  protected boolean error = false;
  protected int d,m,y;
}
```

error stored in field

error in method result

method and field error

```
public void main(String[] args) {
  Date d = new Date(25,4,1974);
  if (d.error())
    doSomethingWithError;
  ...
  int r = Date.monthDays(month,year);
  if (r == -1)
    doSomethingWithError;
  if (r != 31)
    ...
}
```

Defensive Programming (approach #2)

```
public class Date {

  public Date(int day,int month,int year) throws IllegalArgumentException {
    if (!valid(day,month,year))
      throw new IllegalArgumentException();
    d = day; m = month; y = year;
  }

  public static int monthDays(int month,int year) throws IllegalArgumentException
  {
    if (!validMonth(month))
      throw new IllegalArgumentException();
    final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
    int result = days[m-1];
    if (month == 2 && leapYear(year))
      result++;
    return result;
  }

  protected int d,m,y;
}
```

error launched as exception

```
public void main(String[] args) {
  Date d;
  try {
    d = new Date(25,4,1974);
    ...
  }
  catch(IllegalArgumentException e) {
    doSomethingWithError;
  }
  try {
    if (Date.monthDays(month,year) != 31)
      ...
  }
  catch(IllegalArgumentException e) {
    doSomethingWithError;
  }
}
```

VERY IMPORTANT NOTE

The `catch` code should: terminate the program, throw an exception, or retry the `try` code (by inserting the block `try/catch` in a loop). Any other action may create serious robustness problems in the program!

Design by Contract

```
public class Date {

  public Date(int day,int month,int year) {
    assert valid(day,month,year);

    d = day; m = month; y = year;
  }

  public static int monthDays(int month,int year) {
    assert validMonth(month);

    final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
    int result = days[m-1];
    if (month == 2 && leapYear(year))
      result++;

    assert result >= 28 && result <= 31;
    return result;
  }

  protected int d,m,y;
}
```

assertions

```
public void main(String[] args) {
  Date d = new Date(25,4,1974);
  ...
  if (Date.monthDays(month,year) != 31)
    ...
}
```

Error Handling in Modules

- **Ostrich technique**:
  - Simple code, but not robust.
- **Defensive Programming**:
  - Module is internally robust, but without ensuring that clients detect error situations (not externally robust) In approach #2 (*checked* exceptions), the module might be externally robust as long as the advises given in box **VERY IMPORTANT NOTE** are followed;
  - More complex code.
- **Design by Contract**:
  - Simple code, internally and externally robust;
- **External Errors**:
  - External errors are errors which are not totally the responsibility of the program. Hence, it makes little sense to treat them as BUGs.
  - These errors should be handled with normal code, thus a defensive programming approach should be used (either with, or without, exceptions).

Instruction `try/catch` and exceptions types

Instruction `try/catch` and exceptions types

- In the majority of existing programming languages, errors are handled through `try/catch` instructions.
- A common practice is to attach to each kind of error a specific exception type, and, if desired, handle the fault elsewhere in the program using an appropriate `catch` block.
- However, this approach frequently ignores the fact that methods and classes are very powerful abstraction mechanisms; i.e. the use of methods and classes abstracts away implementation choices and details of these modular constructions.
- As such, we have the following consequence:
  - In general, a particular exception type is only meaningful near its origin point;
  - As the exception travels in the thread's execution stack, that meaning quickly disappears.
- Lets see an example:

Instruction `try/catch` and exceptions types (2)

Native Java

```
static double sqrt(double x) {
   if (x < 0)
      throw new IllegalArgumentException();
   ...
}
```

DP: failure as an exception

Native Java

```
/**
 * ax^2 + bx + c = 0
 * equation roots returned as a 4 length array
 * ({r1.re,r1.im,r2.re,r2.im}.
 */
static double[] quadraticEquation(double a,double b,double c) {
   if (a == 0)
      throw new IllegalArgumentException();
   ... sqrt(delta) ...
}
```

DP: failure as an exception

Instruction `try/catch` and exceptions types (3)

- If a client of `quadraticEquation` method catches an `IllegalArgumentException` who is responsible for this error? Is it the `quadraticEquation` method? Or is it the client's fault?
- The problem is that there is not a definitive response. Both cases may occur, meaning that the same exception type can notify completely different faults.
- Worst, the responsibility for this fault might be of the `quadraticEquation` client (precondition failure for a non-quadratic equation); or it might be of the method itself (invalid inner invocation of `sqrt` method).
- A possible solution attempt for this problem could be to systematically use different types of exceptions.
- However, that might require the creation of many new exception types, and a practical nightmare to handle all those situations.
- Another possibility would be to handle (catch) exception where they still are meaningful (i.e. closer to its origin).

Instruction `try/catch` and exceptions types (4)

- However, that would require an overwhelming (and boring) amount of work by programmers with lots of `try/catch` statements and re-throwing of different exceptions, and would undermine one of the first objectives of this mechanism: to be exceptional.
- Yet another possibility would be a programming language that automatically encapsulated exceptions in more generic exception types (e.g. `MethodFailureException`) as it was propagated up in the thread's execution stack.
- Surprisingly, even with any of these possible solutions, this mechanism could still break the algorithmic abstraction of methods.
- Methods create an abstraction barrier between clients and its implementation: The client's goal is to get the (complete) method's postcondition; The method's goal is to provide an algorithm (and data structure) that ensures it.
- However, the possible types of exceptions throwed depend on the chosen algorithm:

Instruction `try/catch` and exceptions types (5)

Native Java

```java
public static int daysOfMonth(int month, int year) {
  final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
  int result = days[month-1];
  if (month == 2 && leapYear(year))
    result++;
  return result;
}
```

ArrayIndexOutOfBoundsException?

Native Java

```java
public static int daysOfMonth(int month, int year) {
  int result = 0;
  switch(month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      result = 31;
      break;
    ...
    default:
      throw new InvalidMonthException();
  }
  return result;
}
```

different exception!

Instruction `try/catch` and exceptions types (6)

- Java's possibility to declare exceptions as part of the method's signature is not a solution for any of these problems because they fail to separate precondition failures from other internal errors.
- Object-Oriented programming aggravate this problem because a method signature no longer applies to a single implementation, but could (dynamically) encapsulate many different implementations (each one with a possible different set of exception types). Some of those implementations may exist only in the future!
- This clearly shows that, beyond simple methods, a client cannot trust in the type and meaning of exceptions throwed by methods.

Instruction `try/catch` and exceptions types (7)

- The simple (and yet powerful) way faults are handled in Design by Contract solves all these problems.
- In DbC a method can only have two possible outcomes: either it succeeds, fulfilling its postcondition (and other possible internal assertions and, is applicable, the object invariant); or it fails throwing an exception. There is no middle ground or ambiguity.
- In DbC's disciplined exception mechanism (not implemented in Java), a faulty method can retry its execution (giving it the chance to comply with its postcondition), or re-propagate the failure to clients.
- Towards that goals, a rescue block can be attached to methods.
- However, a precondition failure is not handled in the method's rescue block (because it is the client's responsibility).

Error Handling in Concurrent Programs

Error Handling in Concurrent Programs

- Exceptions are the best mechanism used to handle errors in a program (both in DbC and in DP).
- The idea is to separate normal code from exceptional code, and to automatically propagate exceptions up in the execution stack trace to a point where, eventually, the problem can be handled, or to become a cause for program termination.
- Essential to the idea is to first propagate the failure to the primary responsible for it (the method/class, or its client).
- With Defensive Programming (with or without exceptions) it is sometimes hard to unambiguously identify whose fault the failure is.
- On the other hand, Design by Contract the different kind of assertions make such identification easy and clear: Preconditions are the client's responsibility. Postconditions and invariants (and other internal assertions) are the module's responsibility.
- This responsibility distribution is of utmost importance because it allows the (automatic) verification of the sanity of the different parts of a program.

Error Handling in Concurrent Programs (2)

- For fault tolerance, only the parts that have failed need to be recovered.
- Exceptions in a sequential program have a global effect on its execution. However, the same thing does not necessarily occur in a concurrent program. A thread exceptional termination might have no effect on the remaining threads.
- However, exceptions are a means to an end. They exist for error handling, so they should promote a sane error handling facility. In particular, exceptions should propagate to contexts (if any) where the fault needs to be handled.
- It is necessary to generalize the mechanism to an OO concurrent context.
- To that goal the following aspects need to be considered:
  - thread communication model (shared object and message passing);
  - fault impact on the involved objects (client and method);
  - fault propagation between threads.

Error Handling: Shared Objects

Error Handling: Shared Objects

- The shared object model is an indirect thread communication mechanism. Thus, by default, there should not exist a direct propagation of exceptions between intervening threads.
- If a thread fails when executing a shared object, and the failure is the result of a contract of the responsibility of the object (any assertion failure that are not the invoked method precondition), then the object must be putted to a failure state (invariant broken).
- Any client, regardless of the thread involved, that attempts to execute the shared object should receive an exception signaling the unavailability (broken invariant) of the object.

Error Handling: Shared Objects (2)

- A possibility to implement this behavior is to add a boolean field to shared objects to register if the object has failed and include its verification in the invariant (which is the proper assertion to check this situation).
- The possibility for object recovery could be considered (in a fault tolerance context) by extending the object's interface with special recovery methods. Those methods should bring the object to a stable time (invariant verification).
- Java does not support such semantics, thus its up to the programmer to implement it.

```
abstract public class SharedObject<T> {
   ...
   public void doSomething() {
         // check invariant:
      assert !unstableState;
      assert invariant;
         // check precondition:
      assert precondition;
      ...
   }

   protected boolean unstableState = false;
}
```

Error Handling: Shared Objects (3)

- To implement fault detection in an object we have at least two possibilities.
- A first possibility (assuming a synchronization scheme with mutual exclusion of commands) is to change the fault indication field which the command is execution:

```java
abstract public class SharedObject<T> {
   ...
   public void doSomething() {
         // check invariant:
      assert !unstableState;
      assert invariant;
         // check precondition:
      assert precondition;
      unstableState = true;
      ...
      unstableState = false;
   }

   protected boolean unstableState = false;
}
```

Error Handling: Shared Objects (4)

- Another possibility is to put the implementation of commands inside a `try/catch` block:

```
abstract public class SharedObject<T> {
   ...
   public void doSomething() {
         // check invariant:
      assert !unstableState;
      assert invariant;
         // check precondition:
      assert precondition;
      try {
         ...
      catch(Throwable e) {
         unstableState = true;
         throw e;
      }
   }

   protected boolean unstableState = false;
}
```

Error Handling: Message Passing

Error Handling: Message Passing Communication

- The proper behavior for error handling in message passing results directly from the semantics of the mechanism that better mimics OOP: ACTORS (objects whose ADT is implemented by RPCs).
- Since there is a direct communication between threads, exceptions should propagate using that communication link.
- However, a problem arises when in presence of an asynchronous communication.
- In this situation, a contract failure might not be properly propagated to the client thread.
- The problem would be specially serious if the responsibility of the failure belonged to the client thread (preconditions), thus sequential preconditions are required to be verified synchronously with the client's invocation (to ensure a proper exception propagation).

Error Handling: Message Passing Communication (2)

- However, that same does not apply to the remaining assertions because they are the responsibility of the actor itself.
- A failure in one of these assertions should result in an actor in a failure state (similarly to a failure inside a shared object). Any latter interaction (e.g. to get the result from a future object, or other actor invocation) with the actor should result in an exception propagation (invariant failure) to the client.
- In any case, the actor will be in an incorrect state.
- Special services to recover actors can be defined (as happened in the case of shared objects).
- Java (again) does not support this behavior, thus all the work need to be done by the programmer.

Error Handling: Variants

Error Handling: Variants

- Regardless of the communication model, we can enumerate the following alternatives for managing failures:

  1. **Termination**: extreme response in which a failure determines the end of the whole program (useful for debugging).
  2. **Continuation**: Ostrich technique in a concurrent context (almost always unacceptable).
  3. **Rollback**: restore the state of the shared object/actor to the situation that preceded the invocation.
  4. **Rollforward**: put the state of the shared object/actor in a stable situation (correct invariant).
  5. **Fault tolerance**: apply a safe fault tolerance technique that causes the shared object/actor to successfully execute the invoked service.

- With the exception of options 1 and 5 (for obvious reasons), all other approaches force the notification of the fault to the object/thread that has requested the service to the shared object/actor (so that they themselves can manage the failure).