Concorrência em Java

- Java suporta a criação explícita de threads, com objectos passivos, e adopta uma variante simplificada do conceito de monitores.
- Java tem algum suporte directo na linguagem para concorrência, incluindo *keywords* como `synchronized`, para exclusão mutua.
- Tal oferece vantagens face a um suporte de biblioteca como a API POSIX Threads; e.g. pares de lock-unlock são balanceados, evitando alguns erros acidentais.
- O modelo oferecido na linguagem é demasiado restritivo; é útil recorrer a bibliotecas de concorrência, como a desenvolvida por Doug Lea, agora integrada no Java5.

Criação de threads

- Threads podem ser criadas e manipuladas por operações da classe `java.lang.Thread`.
- Para definir o comportamento de uma thread, podemos herdar de `Thread` e redefinir o método `run`.
- Também podemos criar uma classe que implementa a interface `java.lang.Runnable`.

```
public interface java.lang.Runnable { void run() }
Runnable r = ...
Thread t = new Thread(r);
t.start();
```

- Tal é mais versátil do que extender `Thread`, pois permite ultrapassar a falta de herança múltipla em Java.

Uso de inner classes anónimas

- Muitas vezes é necessário criar uma thread que possa manipular objectos acessíveis no contexto da criação.
- Torna-se pouco prático ter que, para cada contexto: criar uma nova classe `Runnable`, declarar variáveis de instância, declarar um construtor com parâmetros apropriados.
- É frequente o uso de `inner classes` anónimas em programação concorrente. Exemplo: criação de uma thread que fica a invocar a operação `vender` de um `Artigo`:

```
final Artigo a = ...
final int quant = ...
Runnable r = new Runnable() {
  public void run() {
    a.vender(quant);
  }
};
(new Thread(r)).start();
```

Ciclo de vida de uma thread

- Uma thread criada só começa a correr quando é executado o método `start()`, que leva à invocação de `run()` do objecto com que a thread é iniciada.
- A thread termina a execução quando `run()` retorna.
- Depois de começar a correr e antes de terminar, uma thread pode estar num dos estados: *runnable*, *running* ou *blocked*.
- Uma thread tendo terminado não pode recomeçar a execução. (Não pode ser invocado novamente `start`.)

Alguns métodos de Thread

- O método `isAlive()` é um predicado que devolve `true` se a thread começou a correr mas ainda não terminou.
- `t.join()` bloqueia a thread invocadora até a thread `t` terminar (até `t.isAlive()` devolver `false`).
- O método de classe (`static`) `currentThread()` devolve uma referência para a thread a executar.
- O método de classe `sleep(long msecs)` faz a thread invocadora suspender a execução pelo menos `msecs` milisegundos.

Exclusão mútua

- Em Java, existe um *lock* associado a cada objecto.
- Acesso ao objecto em exclusão mútua pode ser efectuado através do uso de `synchronized`, que pode ser utilizado:
  - em métodos:

```
class Contador {
  int i;
  synchronized void inc() {
    i++;
  }
}
```

  - em blocos de código, utilizando o *lock* de um objecto `obj`:

```
synchronized (obj) {
  ...
}
```

Locking recursivo

- Uma thread que adquiriu um *lock*, através de `synchronized`, pode em seguida invocar métodos ou código `synchronized` relativamente ao mesmo objecto sem ficar bloqueada.
- É diferente do comportamento por omissão em POSIX Threads.
- O *lock* conta quantas vezes foi adquirido, bloqueando outras threads até ser libertado o mesmo número de vezes.
- Isto pode ser designado de *locking* recursivo, pois permite métodos `synchronized` serem recursivos:

```
class Contador {
  int i;
  synchronized void soma(int n) {
    if (n>0) { ++i; soma(n-1); }
  }
}
```

Sincronização

- Java usa uma variante simplificada de monitores para a sincronização entre objectos.
- A cada objecto está associado um *lock* e uma única variável de condição (implícita) com a correspondente fila de espera.
- Os métodos relevantes, existentes na classe `Object`, são:

  `public final void wait() throws InterruptedException` a thread invocadora liberta o lock associado ao monitor e fica à espera de ser notificada. Quando notificada readquire o lock antes de recomeçar a execução.

  `public final void notify()` acorda uma thread bloqueada na fila de espera de `wait()` do objecto.

  `public final void notifyAll()` acorda todas as threads bloqueadas na fila de espera de `wait()` do objecto.

Exemplo: bounded buffer

Um *bounded buffer* clássico com as operações `get` e `put`:

```
class Buffer {
  final int N = 10;
  int i = 0;

  public synchronized Object get() throws InterruptedException {
    while (i == 0) wait();
    i--;
    ...
    notifyAll();
    return ...
  }
  public synchronized void put(Object o) throws InterruptedException {
    while (i == N) wait();
    i++;
    ... = o
    notifyAll();
  }
}
```

Exemplo: bounded buffer—produtor

Produtor que invoca num ciclo infinito a operação `put`:

```
class Producer implements Runnable {
  Buffer b;
  Producer(Buffer b1) {
    b = b1;
  }
  int i;
  public void run() {
   try {
    while(true) {
      i++;
      b.put(...);
      Thread.sleep(1000);
    }
   } catch (InterruptedException e) { }
  }
}
```

Exemplo: bounded buffer—consumidor

Consumidor que invoca num ciclo infinito a operação `get`:

```
class Consumer implements Runnable {
  Buffer b;
  Consumer(Buffer b1) {
    b = b1;
  }
  int i;
  public void run() {
    try {
    while(true) {
      i++;
      b.get();
      Thread.sleep(2000);
    }
    } catch (InterruptedException e) { }
  }
}
```

Exemplo: bounded buffer—Main

Classe main que instancia um buffer, um produtor e um consumidor:

```
class Main {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    Thread t1 = new Thread(p);
    Thread t2 = new Thread(c);
    t1.start();
    t2.start();
  }
}
```

Programação concorrente com Objectos

Tópicos:

- Objectos e actividades: objectos activos e passivos.
- Controlo de concorrência intra-objecto: objectos atómicos, quase-concorrentes e concorrentes.
- Controlo de concorrência inter-objecto.
- Imutabilidade.

Objectos e actividades

Objectos passivos

objectos e threads são considerados conceitos independentes

- ambos são manipulados explicitamente pelo programador.
- o mais frequente nas linguagens comuns, como C++ ou Java.

Objectos activos

existe uma unificação entre objecto e thread

- um objecto activo pode ter uma thread associada;
- a execução de uma operação pode ter uma thread dedicada;
- a concorrência é criada implicitamente:
  - pela instanciação assíncrona;
  - pela invocação assíncrona.

Objectos activos

Instanciação assíncrona

O objecto criado fica a executar um *body* (que pode ser implícito ou explícito) que fica em ciclo à espera de pedidos.

Invocação assíncrona

o cliente prossegue concorrentemente e o resultado é obtido mais tarde, de diferentes modos:

one way invocations não devolvem resultados; se necessário o servidor envia o resultado através de outra invocação.

objectos futuros podem ser devolvidos por invocações assíncronas. O futuro é usado pelo cliente para obter o resultado. Os futuros podem ser implícitos ou explícitos.

Objectos activos como design pattern

- Normalmente é oferecido o conceito de objecto passivo.
- Tal não impede que se possa utilizar o conceito de objecto activo, quando apropriado, para estruturar o software.
- Objectos activos, quando não suportados directamente pela linguagem, podem ser construidos como um *design pattern*.
- Ver [POSA2] Pattern-Oriented Software Architecture, Vol 2 (Patterns for Concurrent and Networked Objects), Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, Wiley, 2000.

Classificação da concorrência intra-objecto

Relativamente à concorrência intra-objecto, podemos classificar um objecto de:

Sequencial ou atómico quando não suporta concorrência intra-objecto; processa uma mensagem de cada vez.

Quase-concorrente quando várias invocações podem coexistir mas no máximo uma não está suspensa; semelhante ao conceito de monitor.

Concorrente suporta verdadeira concorrência entre invocações, exigindo controlo a ser especificado pelo programador.

Objectos sequênciais ou atómicos

- A forma mais simples de objecto para uso concorrente processa uma invocacão de cada vez: as invocações são serializadas.
- Todos os seus métodos adquirem um *lock* no inicio da execução, libertando-o quando terminam. (Métodos `synchronized` em Java.)
- Todos os métodos acabam em tempo finito, não ficando bloqueados, e garantidamente libertam os *locks*.
- O estado do objecto obedece aos invariantes no ínicio e no fim de cada método.
- Facilita construção do software com garantias formais de correcção.

Exemplo: conta bancária em Java

Uma classe com todos os métodos `synchronized`, sendo os objectos atómicos:

```
class Conta {
  int saldo;

  public synchronized int consulta() {
    return saldo;
  }

  public synchronized void deposito(int valor) {
    saldo = saldo + valor;
  }

  public synchronized void levantamento(int valor) {
    saldo = saldo - valor;
  }
}
```

Monitores / objectos quase-concorrentes

- O conceito de monitor permite a obtenção de objectos quase-concorrentes: que suportam várias invocações em curso, ainda que só uma no máximo esteja não bloqueada.
- Ao contrário dos objectos atómicos, as invocações não são serializadas: ainda que bloqueadas, podem já ter executado parcialmente, estando à espera de um evento externo a elas.
- Permite controlar a colaboração entre clientes que fazem uso de um serviço.
- Cada método faz uso de um *mutex* e de variáveis de condição. (Java usa apenas uma variável de condição, implícita.)
- Exemplo: produtor-consumidor em Java. Um *bounded buffer* é um monitor para uso por threads produtoras e consumidoras.

  O buffer poderá estar cheio ou vazio, podendo ser necessário bloquear pedidos.

Objectos concorrentes

- Um serviço poderá disponibilizar operações que poderão ser demoradas a executar (por exemplo input/output), ainda que não dependam umas das outras.
- Nestes casos deverá ser implementado como um servidor concorrente, ou seja como um objecto que permita verdadeira concorrência entre invocações a ser processadas.
- Tal leva a uma implementação com controlo de concorrência de granularidade mais fina.
- Em vez de *locking* a nível do objecto, são por exemplo usados *locks* relativamente a sub-objectos.
- A implementação é mais complexa, necessitando mais cuidado.

Exemplo: operações sobre objectos em repositórios

```
interface Operacao { void aplica(Object o); }

class Repositorio {
  public synchronized void insere(String nome, Object o) {
    // insere o objecto no repositorio
  }

  public void aplica(String nome, Operacao op) {
    Object obj;
    synchronized (this) {
      obj = ...  // procura o objecto pelo nome
    }
    synchronized (obj) {
      op.aplica(obj);       // operacao potencialmente demorada
    }
  }
}
```

Controlo de concorrência inter-objecto

- Concorrência inter-objecto: na invocação de operações em objectos diferentes.
- Controlo de concorrência pode ser necessário para garantir coerência no estado global do sistema (e não de objectos individuais).
- Tal pode acontecer quando existem dependências entre operações a ser efectuadas em objectos diferentes.

Exemplo: operações sobre duas contas bancárias

- Para realizar uma transferência é realizada uma operação de levantamento na primeira conta e outra de depósito na segunda.

```
c1.levantamento(3000);
c2.deposito(3000);
```

- Suponhamos que é consultado concorrentemente o saldo de cada conta (para por exemplo obter a soma dos saldos).

```
i = c1.saldo();
j = c2.saldo();
```

- Se tal for efectuado depois do levantamento mas antes do depósito, o resultado é inválido.
- Nestes casos é necessário prevenir interferência entre cada conjunto de operações: obter isolamento.
- Uma hipótese é forçar serialização das operações.

Exemplo: operações sobre duas contas bancárias

- Uma solução para o problema pode passar por utilizar um objecto `mutex`.
- Um `mutex` pode ser um exemplo de um objecto distribuído, conhecido por ambos os clientes e a prestar um serviço de controlo de concorrência.

```
Conta c1, c2; Mutex m;

// cliente 1; realiza uma transferencia
  m.lock();
  c1.levantamento(3000);
  c2.deposito(3000);
  m.unlock()

// cliente 2; consulta as duas contas
  m.lock();
  i = c1.saldo();
  j = c2.saldo();
  m.unlock()
```

Exemplo: operações sobre duas contas bancárias

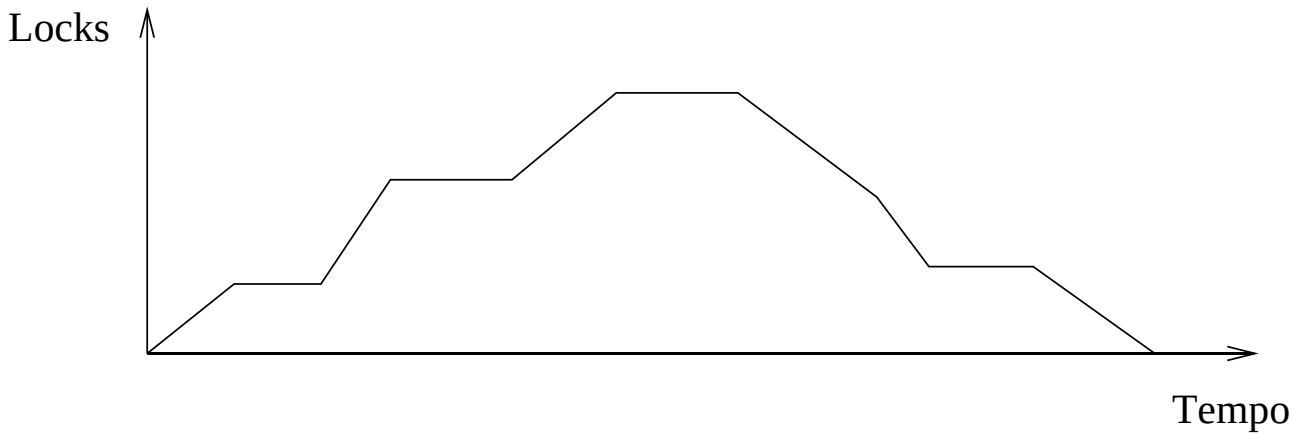
- No caso geral vários clientes podem manipular várias contas.
- Solução: cada cliente adquire os *locks* dos objectos a manipular, efectua as operações em questão, e finalmente liberta os *locks*.

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    lc2.lock();
    c1.levantamento(3000);
    c2.deposito(3000);
    lc1.unlock();
    lc2.unlock();

// cliente 2; consulta as duas contas
    lc1.lock();
    lc2.lock();
    i = c1.saldo();
    j = c2.saldo();
    lc1.unlock();
    lc2.unlock();
```

Two-phase locking

- Técnica de controlo de concorrência usada em bases de dados e sistemas de objectos distribuídos, para obter isolamento entre transacções ao garantir equivalência a serialização.
- Cada transacção envolvida passa por duas fases: aquisição de *locks*; libertação de *locks*.
- Depois de algum *lock* ser libertado, mais nenhum é adquirido.



- Um *lock* de um objecto só é libertado quando a transacção já possui todos os *locks* de que necessita.

Exemplo: operações sobre duas contas com 2PL

- O exemplo anterior pode ser refinado para reduzir a latência dos locks aproveitando a estratégia Two-phase locking:

```
// cliente 1; realiza uma transferencia
    lc1.lock();
    c1.levantamento(3000);
    lc2.lock();
    lc1.unlock();
    c2.deposito(3000);
    lc2.unlock();

// cliente 2; consulta as duas contas
    lc1.lock();
    i = c1.saldo();
    lc2.lock();
    lc1.unlock();
    j = c2.saldo();
    lc2.unlock();
```

Modos de locks

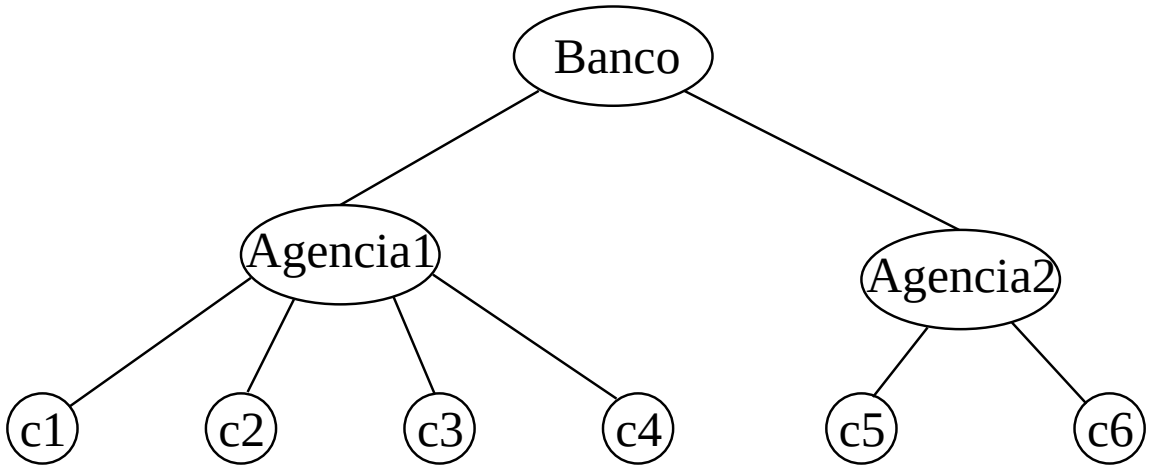
- Locks de exclusão mútua (binários) podem ser demasiado restrictivos.
- É útil distinguir diferentes tipos de acesso e oferecer *locks* de leitura e de escrita,
- o que permite que várias leituras possam prosseguir concorrentemente.
- Tabela de compatibilidade de *locks*:

	read	write
read	+	-
write	-	-

Locking hierárquico

- Em sistemas de objectos hierárquicos certos objectos são *containers* de um conjunto de objectos componentes.
- *Locking* hierárquico permite obter o *lock* de todos os componentes de um *container* fazendo *lock* deste.
- São oferecidas duas operações: `lock` e `intention-lock`.
- Para fazer *lock* a X (*container* ou componente) é feito um `intention-lock` em todos os *containers* desde a raiz até ao *container* de X, seguida de um `lock` de X.
- O *locking* hierárquico é vantajoso quando existem muitos objectos e uma hierarquia de composição pouco profunda.

Locking hierárquico



Processo 1: lock de c1

```
Banco.lock(intention_write);
Agencia1.lock(intention_write);
c1.lock(write);
```

Processo 2: lock de Agencia2

```
Banco.lock(intention_write);
Agencia2.lock(write);
```

Compatibilidade de locks hierárquicos

	intention read	read	intention write	write
intention read	+	+	+	-
read	+	+	-	-
intention write	+	-	+	-
write	-	-	-	-

Imutabilidade

- Objectos que não mudam de estado podem ser usados concorrentemente sem restrições.
- Estes podem ser úteis para uso na implementação (como sub-objectos) de objectos concorrentes, diminuindo as necessidades de controlo de concorrência.
- Sempre que possível deve ser usado suporte da linguagem para ter garantias que um dado objecto é imutável; por exemplo a palavra-chave `final` em Java.
- É necessário evitar que se escapem referências para o futuro objecto imutável enquanto este está a ser construído.