REST API és concurent package

Java PROGRAMOZÁS

10. GYAKORLAT

JSON és REST API

JSON

- JavaScript Object Notation rövidítése.
- Mint a neve is jelzi, a JavaScript –ben használták először, de már az egyik legelterjedtebb szöveg alapú adattárolásra használt formátum.
  - Az XML alternatívája a webes kommunikációban.
- 2006-ban az RFC4627-ben lett specifikálva.
- Legtöbbször strukturált adatok továbbítására és tárolására használják.
  - Az XML-nél rövidebb, ezért kisebb helyet foglal, kisebb a hálózati forgalma.
- Mi webservice-ekkel való kommunikációra fogjuk használni.

JSON példa

```
{
      „firstName": "Kovacs",
      „secondName": "Janos",
      „age": 25,
      „address":
      {
          „street": „Harsfa utca 11.",
          „city": „Budapest",
          „zipCode": „1162"
      },
      „weight”: 75.3
…
```

```
…
   „phoneNumbers":
      [
          {
            „type": „home",
            „number": „06 1 555-1234"
          },
          {
            "type": "fax",
            „number": "646 555-4567"
          }
      ]
 }
```

JSON

Adattípusok

- Szám (Double vagy Long)
- Karakterlánc (String: idézőjelek közt, Unicode karakterekből (alapból UTF-8 kódolásban) balra dőlő törtvonallal (backslash, \) escape-elve)
- Boolean (true (igaz) vagy false (hamis))
- Tömb (értékek sorrendhelyes felsorolása vesszővel elválasztva, szögletes zárójelek között; az értékeknek nem kell azonos típusúnak lennie benne)
- Objektum (kulcs:érték-párok rendezetlen gyűjteménye, amelyben ':' karakter választja el a kulcsot és az értéket, a párok egymástól vesszökkel vannak elválasztva, a lista kapcsos zárójelek között van; a kulcsok mindig String típusúak, és különbözniük kell egymástól)
- null (üres)

Próbáljuk ki

Nyissátok meg a `JsonDemo` projektet

- Javítsátok ki a encoder-t, hogy ugyan olyan JSON-t dekódoljon, mint az előző dián szerepelt
- Meg tudjátok nézni a generált JSON-t ha bemásoljátok ide.
- Módosítsátok a decoder-t, hogy
  - János megvizsgálja, hogy a súlya-e a nagyobb vagy az életkora?
  - Írja ki János városát
  - Mindegyik példa json-t tudja parsolni (ne kapjunk exception-t)

Példák:

```
JSONParser parser = new JSONParser();
 s = "{}";
obj = parser.parse(s);

JSONObject man = new JSONObject();
man.put("firstName", "Kovacs");
man.put("age", new Integer(25));
```

google-gson

- A gson egy nyílt forráskódú könyvtár, amit JSON szövegek és objektumok konvertálására való, mindkét irányban működik. Így nem kell kézzel megírni a dekódolást és enkódolást!
- Egyszerűen az osztály mezőinek a nevét használja kulcsként és az értékét írja az érték helyére.
- Gson előnye a konkurensekkel szemben, hogy nem szükséges Annotációk használata és teljesen kompatibilis a java generikusokkal.
- Így már előre meglévő (nem általunk definiált) osztályok objektumait is tudja JSON-ná konvertálni.
- Mi arra fogjuk használni, hogy egy szervertől lekért JSON fájlt dolgozzunk fel.

REST API

- A REST (Representational State Transfer) egy szoftverarchitektúra típus, elosztott kapcsolat (loose coupling), nagy internet alapú rendszerek számára, amilyen például a világháló.
- Azokat a rendszereket, amelyek eleget tesznek a REST megszorításainak, "RESTful"-nak nevezik.
- Egy REST típusú architektúra kliensekből és szerverekből áll.
  - A kliensek kéréseket indítanak a szerverek felé.
  - A szerverek kéréseket dolgoznak fel és a megfelelő választ küldik vissza.

6 megszorítás az architektúra definiálására (az implementáció nincs megkötve):

- Kliens-szerver architektúra
- Állapotmentesség
- Gyorsítótárazhatóság
- Réteges felépítés
- Igényelt kód (opcionális)
- Egységes interfész

REST API - megszorítások

- Kliens – szerver architektúra
  - Kliens és szerver elkülönített egy egységes interfész által (pl. kliens nem foglalkozik adattárolással, csak a megjelenítéssel a felhasználó felé)
  - Szerver és kliens áthelyezhető, külön-külön fejleszthető, csak az interfész kötött
- Állapotmentesség
  - A kliens – szerver kommunikáció további korlátozása azzal, hogy a kliens állapotát nem tárolja a szerver.
  - Minden kérés tartalmazza az összes információt a kliens kiszolgálásához
- Gyorsítótárazhatóság
  - A kliensek képesek gyorsítótárazni a válaszokat, ezért a válaszoknak tartalmazniuk kell, hogy gyorsítótárazhatóak-e.
  - Így elkerülhető, hogy a kliens téves vagy elavult adatokat használjon fel

REST API - megszorítások

- Réteges felépítés
  - Egy kliens általában nem tudja megmondani, hogy direkt csatlakozott-e a végpont szerverhez, vagy közvetítő segítségével.
  - A közvetítő szerverek megnövelhetik a rendszer skálázhatóságát terheléseloszlás kiegyenlítéssel és megosztott gyorsítótárak használatával.
- Igényelt kód (opcionális)
  - A szerverek képesek időlegesen kiterjeszteni vagy testre szabni egy kliens funkcionalitását, programrészek átadásával, amelyeket a kliens futtatni képes.
- Egységes interfész
  - Az egységes interfész kliens és szerver között egyszerűsíti és kettéválasztja az architektúrát, és lehetővé teszi, hogy egymástól függetlenül fejlődjenek az egyes részek.

Egységes interfész

Az egységes interfész tervezésekor négy vezérelvet kell figyelembe venni:

- Erőforrások azonosítása
  - Egyéni erőforrások azonosítása a kérésekben történik, például URI-k használatával. Tehát pl. a szerver nem küldi el a teljes adatbázist, csak a kért rekordokat egy JSON dokumentumban.
- Erőforrások manipulációja ezeken a reprezentációkon keresztül
  - A rekordok módosítása és törlése is ugyan ezen a reprezentáción keresztül történik, tehát elegendő, ha a kliens rendelkezik egy adat reprezentációval.
- Önleíró üzenetek
  - Minden egyes üzenet elegendő információt tartalmaz az üzenet feldolgozásához.
- Hipermédia, mint az alkalmazásállapot motorja
  - Minden, a kliensek állapotátmenetéhez szükséges adatot a szerver tartalmaz, melyeket a szerver a kliensek kérésére elküld.

Web szolgáltatások felépítése

HTTP alapú RESTful API-knak a következő komponenseik vannak:

- Base URL, pl. http://api.example.com/resources/
- MIME (Multipurpose Internet Mail Extensions) type
- Alapértelmezett HTTP függvények egyike (GET, PUT, POST, DELETE)

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as http://api.example.com/resources/	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element, such as http://api.example.com/resources/item17	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a	**Delete** the addressed member of the collection.

Próbáljuk ki

- Nézzük meg együtt a GsonDemo program működését:
  - http://fixer.io/ nyílt API-t használja pénznemek konvertálására
  - gson könyvtár segítségével parsolja a kapott json objektumot
- Javítsátok ki a programot, hogy ne csak USD-t, hanem HUF-ot is lehessen vele átváltani
  - Keressétek a „//TODO” kommenteket

Java.util.concurent

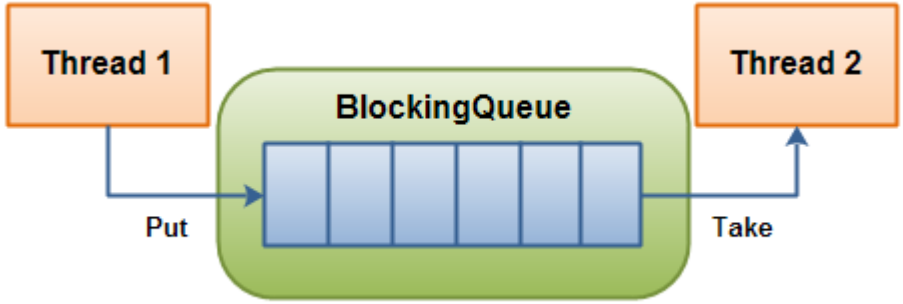
Doc és tutorial

Java.util.concurent csomag

- Kiegészítő osztályokat tartalmaz, melyek könnyebbé teszik a párhuzamos programok létrehozását.
- Tartalmaz adatszerkezeteket, mint a `BlockingQueue`, `BlockingDequeue` vagy a `ConcurentMap`
- Tartalmaz szálak indítását és kezelését leegyszerűsítő osztályokat, mint az `ExecutorService`, `ThreadPoolExecutor` vagy a `ForkJoinPool`
- És egyéb párhuzamos futáshoz kapcsolódó osztályokat, pl. `CyclicBarrier` vagy a `CountDownLatch`
- Ha többszálú programot írunk érdemes ezeket használni az egyszerű szálindítás helyett, mert jobban optimalizál és egyszerűbbé, rövidebbé teszi a kódot.
- Nézzük meg ezeket részletesebben!

BlockingQueue

- Sor interfész, melybe szálbiztosan lehet betenni és kivenni elemeket
- A meghívott metódus különböző képpen kezelheti le ha jelenleg nem lehetséges a művelet
  - **Kivételt dob.**
  - **Visszatér** egy speciális értékkel.
  - **Vár**akozik addig, amíg felszabadul a sor.
  - Egy **határidő** után visszatér egy speciális értékkel, addig vár.



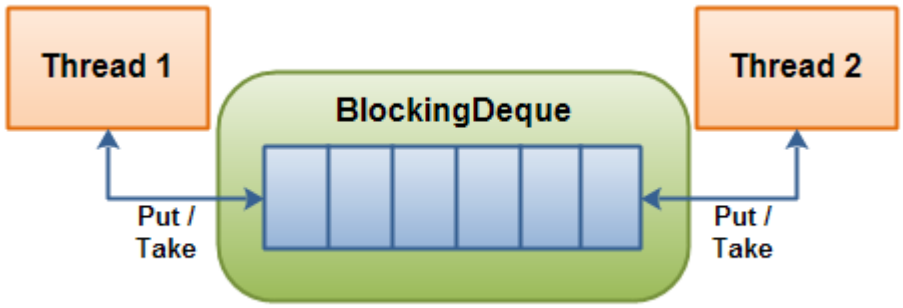
Viselkedés	Kivételt dob	Visszatér	Vár	Határidő
Beszúrás	`Add(o)`	`Offer(o)`	`Put(o)`	`Offer(o, timeout, tiemunit)`
Kivevés	`Remove(o)`	`Poll()`	`Take(o)`	`Poll(timeout, timeunit)`
Lekérés	`Element()`	`Peek()`		

`BlockingQueue` implementációk

- Mivel a `BlockingQueue` egy interfész, ahhoz hogy példányosítani tudjuk szükségünk van egy implementációra (pl. `List` interfész, `ArrayList` implementáció)
- A következő implementációkat tartalmazza a concurent csomag:
  - `ArrayBlockingQueue`
    - Korlátozott méretű
  - `DelayQueue`
    - Az elemeket egy általuk meghatározott ideig nem adja ki
  - `LinkedBlockingQueue`
    - Láncolt listában tárolja az elemeket
  - `PriorityBlockingQueue`
    - `PriorityQueue`-hoz hasonló, rendezve szúrja be az elemeket
  - `SynchronousQueue`
    - Egyszerre csak egy elemet tartalmazhat, addig nem engedi a beszúrást, amíg azt ki nem veszik

BlockingDeque

- BlockingQueue-hoz hasonló, csak mindkét végéhez hozzá lehet férni.
- Implementációk:
  - LinkedBlockingDeque



Viselkedés	Kivételt dob	Visszatér	Vár	Határidő
Beszúrás	AddFirst(o)	OfferFirst(o)	PutFirst(o)	OfferFirst (o, timeout, tiemunit)
Kivevés	RemoveFirst(o)	PollFirst ()	TakeFirst (o)	PollFirst (timeout, timeunit)
Lekérés	ElementFirst ()	PeekFirst ()		
Beszúrás	AddLast(o)	OfferLast (o)	PutLast (o)	OfferLast (o, timeout, tiemunit)
Kivevés	RemoveLast (o)	PollLast ()	TakeLast (o)	PollLast (timeout, timeunit)

ConcurentMap

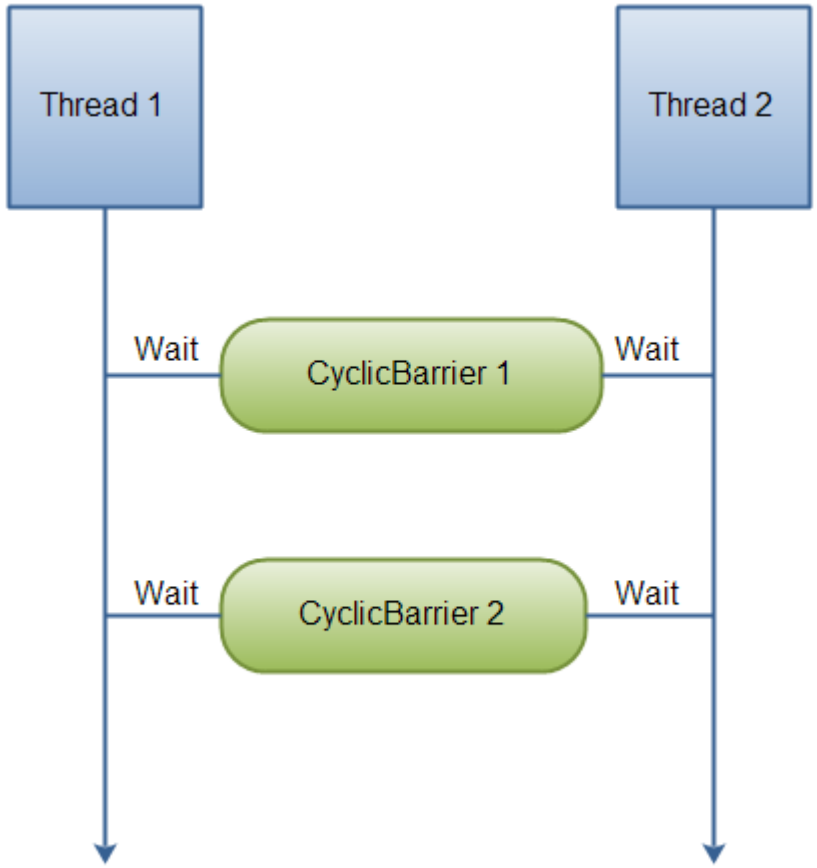
- Egy kulcs-érték párakat tartalmazó osztályt reprezentáló interfész, amely képes a konkurens hozzáférések kezelésére.
  - A `java.util.Map` interfészől származik és csak egy pár atomic függvénnyel egészíti ki azt
- Implementációk:
  - `ConcurrentHashMap`
- A `ConcurrentHashMap` nagyon hasonló a `java.util.HashTable`-hez, azzal a különbséggel, hogy olvasásnál nem blokkolja a táblát és írásnál is csak azt a részét, amelyikbe ír.
- Illetve nem dob `ConcurentModificationException`-t, ha megváltozott iteráció közben.

CountDownLatch

- Arra való, hogy több szál meg tudja várni ugyan azon események bekövetkeztét.
- Egy számmal kell inicializálni, amely minden `countDown()` hívásra csökken.
- Ha egy szál meghívja az `await()` függvényét addig amíg a számláló nem nulla, akkor felfüggeszti a futását a szálnak.
- Amikor a számláló eléri a 0-át, akkor felengedi az összes szálat, amelyiket addig várakoztatta.

CyclicBarrier

- Segítségével a párhuzamosan dolgozó szálak chekcpoint-ok ban képesek bevárni egymást.
- Inicializálásnál specifikálni kell, hogy hány szálat kell bevárni




```
CyclicBarrier barrier = new
CyclicBarrier(2);
```

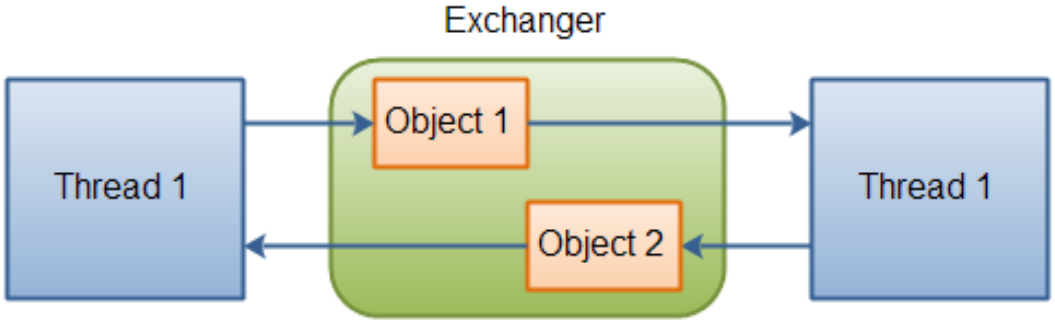
Utána pedig az `await()` hívással lehet checkpoint-ot definiálni a egy szálban:

```
barrier.await();
```

Amikor az előre meghatározott számú `await()` hívás történt, akkor a barrier tovább engedi a szálakat.

Exchanger

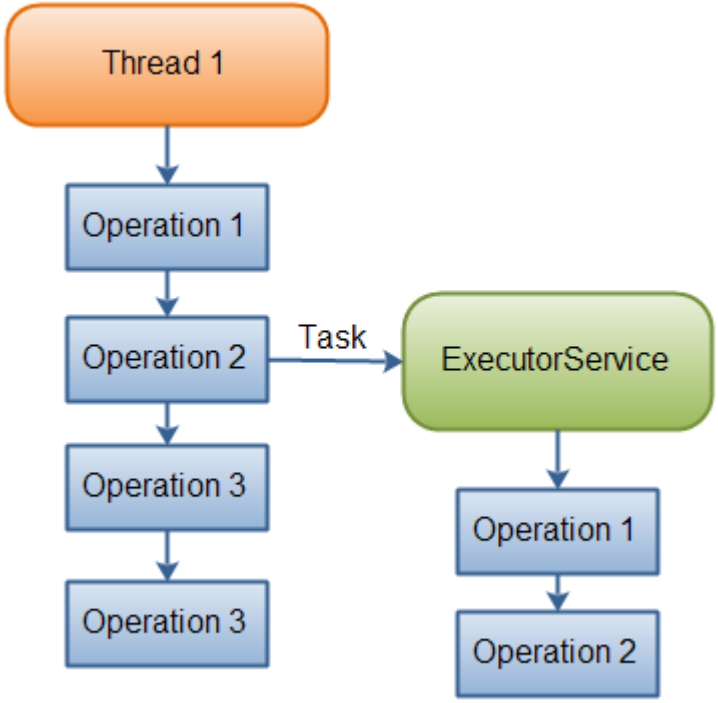
- Egy „találkozási pont” két szál futása között, ahol objektumokat cserélhetnek
- Fontos, hogy az objektumok azonos típusúak legyenek!




```
Exchanger exchanger = new Exchanger();
…
// in the threads after received exchanger:
this.object = this.exchanger.exchange(this.object);
```

ExecutorService

- Egy interfész, ami párhuzamos feladatvégrehajtást reprezentál a háttérben.
- Meg lehet határozni inicializáláskor, hogy hány szálat futtasson egyszerre az executor, majd `Runnable` objektumokat átadva neki elosztja azokat a szálakra.
- Előnye, hogy nem kell minden egyes alkalommal szálat indítani, ami erőforrásigényes.



ExecutorService

- Implementációk:
  - `SingleThreadExecutor`
  - `ThreadPoolExecutor`
  - `ScheduledThreadPoolExecutor`

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

Különböző lehetőségeink vannak rá, hogy delegáljuk a feladatot az executornak a következő függvények segítségével:

- `execute(Runnable)`
- `submit(Runnable)`
- `submit(Callable)`
- `invokeAny(...)`
- `invokeAll(...)`

Fontos, hogy a `shutdown()`-t meghívjuk, ha már nincs szükség az `ExecutorService`-re!

Future<V> és Callable<V>

- Callable<V>
  - Runnable-hez hasonló interfész.
  - Aszinkron utasítások definiálásakor használjuk, a különbség, hogy van visszatérési értéke
  - call() függvényét kell felülírni, ezt fogja meghívni az új szálon az ExecutorService
  - A call() függvény V visszatérési értékű, ami a számolás eredményének a típusa
- Future<V>
  - Interfész aszinkron számítás eredményének kinyeréséhez.
  - Ha Callable-t indítunk Runnable helyett, akkor az indításkor visszakapunk egy Future objektumot, aminek a segítségével megtudhatjuk, hogy véget ért-e már a számítás és lekérdezhetjük az eredményét.
  - Ugyan annak kell lennie a template típusának mint a Callable-nek.
  - Függvényei:
    - isDone(), isCancelled() állapot lekéréséhez
    - get(), get(timeout): eredmény lekéréshez. Addig blokkolják a szálat, amíg nem kapják meg az eredményt, vagy nem jár le a timeout.

ExecutorService

- `execute(Runnable)`
  - Egy `Runnable`-t vesz át és azt aszinkron futtatja.
  - Nem tér vissza az eredménnyel
  - `shutDown()` hívással le lehet állítani
- `submit(Runnable)`
  - Ez is `Runnable`-t vesz át és visszatér egy `Future` Objektummal
  - `Future.Get()` függvénnyel le lehet kérni, hogy fut-e még a `Runnable` (hasonló a működése mint a `Callable` esetén, csak a null a visszatérési érték)
- `submit(Callable)`
  - `Callable` objektumot kap
  - Hasonlóan a másik `submit`() függvényhez egy `Future` Objektummal tér vissza.

ExecutorService

- `invokeAny()`
  - `Callable` objektumokat `Collection`–jét veszi át és párhuzamosan futtatja őket
  - Amíg futnak blokkolja a hívó szálat
  - Amikor valamelyik lefutott visszatér annak a visszatérési értékével, és a többit leállítja
- `invokeAll()`
  - Szintén `Callable` objektumok `Collection`-jét veszi át
  - Egy `Future` objektum listával tér vissza, amin keresztül le lehet kérni az összes `Callable` eredményét
- `shutdown()`
  - Az `executorService` nem állítja le a szálakat addig, amíg `shutdown()`-t nem hívunk rá, úgyhogy fontos, hogy ezt ne felejtsük el, különben nem fog leállni a programunk.
  - Nem szakítja meg a szálak futását, csak már nem fog elfogadni új `Callable` vagy `Runnable` objektumokat

További hasznos osztályok

- `ThreadPoolExecutor`
  - Meghatározott számú szálat futtat egyszerre és ezen ütemezi be a kapott `Runnable` illetve `Callable` osztályok megfelelő metódusait
- `ScheduledExecutorService`
  - Ütemezni képes a feladatokat, vagy többször lefuttatni egymás után
- `ForkAndJoinPool`
  - `ExecutorService`-hez hasonló, viszont lehetővé teszi a feladatoknak, hogy tovább delegálják a feladatot részfeladatokra

Semaphore és Lock

- A `synchronized` blokk alternatívái.
- `Semaphore`
  - Előre meghatározott számú szál férhet hozzá egy adott kódrészlethez
  - `acquire`()-t és `release`()-t kell hívni a blokk elején és végén
- `Lock`
  - `lock`()-t és `unlock`()-ot kell hívni a szinkronizált kódrészlet elején és végén
  - Előnyei a `synchronized`-al szemben
    - Abban a sorrendben engedi fel a szálakat, ahogy a `lock`()-hoz értek
    - Timeout-olhat
    - Több metóduson keresztül lehet tartani a `lock`()-ot

Gyakorló feladatok

- JSON kipróbálása
- GSON kipróbálása
- REST Api tesztelése
- Concurrent csomagban található eszközök kipróbálása
- A következő szerkezetek/eszközök „nulláról” megírása
  - Szemafor
  - Lock
  - CountdownLatch
  - BlockingQueue