OMO

7 - Datové struktury a patterny

- Datové struktury - částečně persistentní, plně persistentní, konfluentní, funkční a retroaktivní
- Lazy Initialization
- Object pool
- Cache

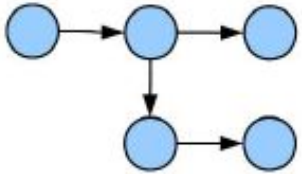
Ing. David Kadleček, PhD
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Persistentní struktury

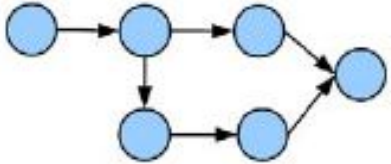
Datové struktury rozlišujeme z pohledu persistence na::

- **Ephemeral DS** - struktury, které nejsou perzistentní
- **Partial persistence DS (částečná persistence)** - můžeme se dotazovat na jakoukoliv minulou verzi dat, ale updatovat lze pouze poslední verzi. Podporovány jsou operace *read(var, version)* a *newversion = write(var, val)*. Jestliže dělám updaty pouze do poslední verze, tak lze verze uložit jako lineární seznam.

- **Full persistence DS (plná persistence)** - můžeme se dotazovat na a updatovat jakoukoliv minulou verzi dat. Podporovány jsou operace *read(var, version)* a *newversion = write(var, version, val)*. Jestliže provedeme update do starší verze, tak musíme udělat novou ***branch*** => stromová reprezentace



- **Confluent persistence DS (slévající se persistence)** - podporuje to co full persistence, ale navíc umožňuje složení více minulých verzí do jedné (***merge***). Podporovány jsou operace *read(var, version)* a *newversion = write(var, version, val)* a *newversion = combine(var, val, version1, version2).* Tento model Implikuje uspořádání verzí do acyklického přímého grafu.



Persistentní struktury

- **Functional persistence DS (funkční persistence)** - nese si jméno z funkcionálního programování , kde se pracuje výhradně s immutable datovými strukturami. Stejně i nody v tomto modelu jsou immutable - revize (updaty) nemění existující nody v datové struktuře. V každém kroku se vytváří klon celé datové struktury, kterou updatuju. Standardně *O(lg n).* Narozdíl od předchozích modelů, kdy update provádím pouze do objektu, který měním.

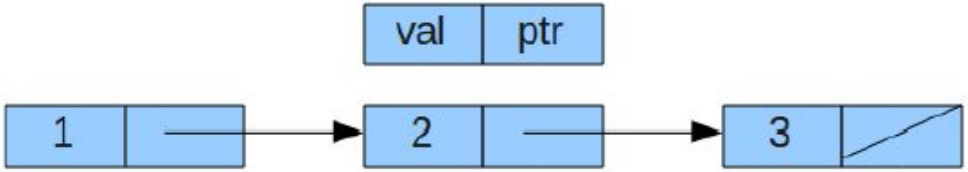
- **Retroactive DS (retroaktivní persistence)** - předchozí modely persistence fungují tak, že změna do starší verze vytváří novou branch do které probíhají všechny další updaty tak, že původní větev zůstává beze změny. Retroaktivní DS funguje tak, že změna do starší verze je provedena přímo do ní a ve všech navazujících verzích se znovu přehrají provedené operace.

~ analogie se strojem času, kterým se vrátíme do minulosti, tam provedeme změnu, která má ale vliv až do současnosti

Source: Demaine, Erik D; John Iacono; Stefan Langerman (2007). "Retroactive data structures". ACM Transactions on Algorithms. ***3***

Partial Persistence

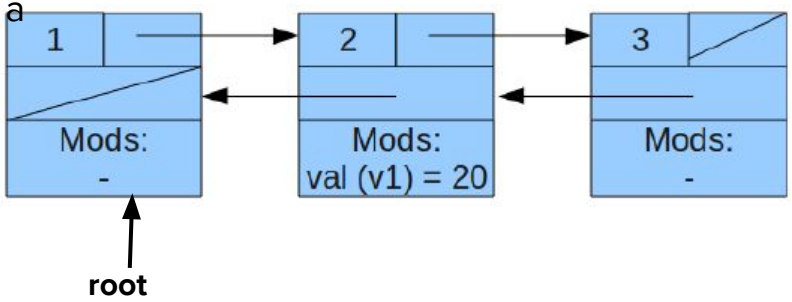
Ephemeral DS má následující strukturu:



Máme v ní tedy objekty, které mají property (val1 .. valN) naplněné hodnotami a ukazatele na další objekty (ptr1 .. ptrN). Do této struktury chceme začít dělat změny: změna hodnoty property, přelinkování na jiný objekt, vytvoření nového objektu.

Pro realizaci partial persistence rozšíříme ephemeral DS o:

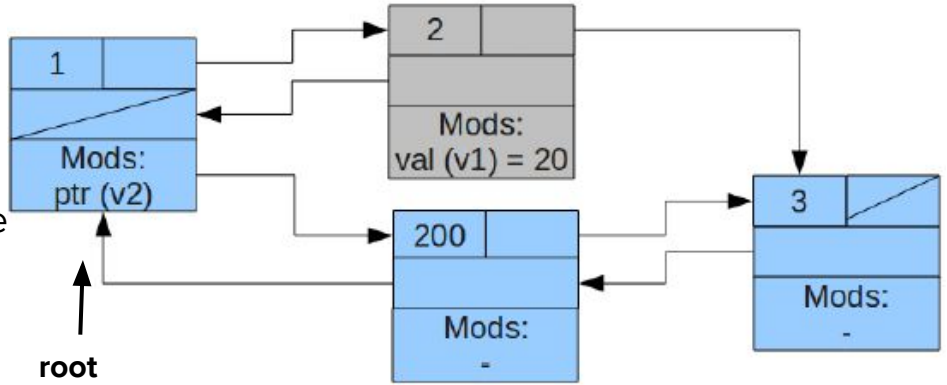
- log s modifikacemi (***mods***), který zapisuje updaty obsahu a ukazatelů ve formě obsah nové verze a její číslo *(field, version, new value)*
- Zpětné reference, které pro každý ukazatel v datové struktuře zavádí i druhý opačným směrem



Při změně upravujeme pouze části objektů, které se změnily ... každý update přidává nový záznam do *mods*.

Partial Persistence

V případě, že se nám zaplní mods, tak vytvoříme nový node (*node'*) a převážeme linky ze starého node do tohoto nového node - zpětné ukazatele na starší node nedržíme, což je výhoda partial persistence modelu.



read(var, version) - začneme root nodem a sledujeme ukazatele - vybíráme ukazatel ve verzi nejvyšší $w \leq v$ a v každém node vybíráme data ve verzi nejvyšší $w \leq v$

write(var, version, val) - provedeme updatem node, se kterým aktuálně pracujeme způsobem viz výše

Pozn. Takto implementovaný model partial persistence má konstantní asymptotickou složitost *O(1)* viz: James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, Robert Endre Tarjan: Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124 (1989)

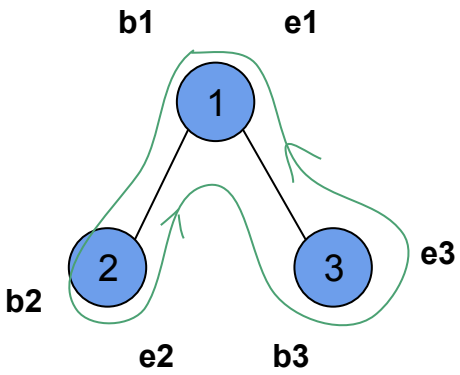
Full Persistence

Je realizována na stejných principech jako partial persistence. Nicméně pokud chci držet efektivitu algoritmu, tak musíme vyřešit:

- **Verzování zpětných referencí** - jestliže dělám update do node od kterého mám již vytvořenou *node',* tak potřebuju zpětnou referenci ***pro všechny verze**,* abych věděl na jakém node převázat přímé reference (jinak bych musel procházet celou datovou strukturu a hledat vazby opačným směrem).
- **Roztržení starých nodes při jejich update** - pokud dělám update do node od kterého mám již vytvořenou *node',* tak ten má již plný mods log. Řeším to tak, že z tohoto node udělám dva kusy, každý s polovinou záznamů v mods log a přelinkuji vazby vedoucí na tento node.
- **Udržování stromu verzí a jeho linearizace** - už si nevystačíme pouze s logováním změn přímo do datové struktury a ukazatelem na root node. Musíme mít reprezentaci, kde:
    - ... umožňujeme update do starší verze, kdy nám de facto vzniká nová branch.
    - ... zpětně zjišťovat v jakém časovém okamžiku a v jaké verzi došlo k větvení.
    - ... z každé verze provést tzv. Backtracking - tedy zpětně projít celou historií updatů, které vedly k mé verzi

  => to vede v podstatě na strom, kde každý node je nějaké verze, link mezi dvěma nody je update mezi dvěma verzemi, větvení znamená, že jsme udělali update do některé starší verze.

Full Persistence



Pro efektivní práci provedeme linearizaci takového stromu následujícím způsobem:

Jelikož, každý node reprezentuje okamžik v čase, kdy jsem dělal update, tak si zavedu index, který mi definuje okamžik před tímto updatem - *b* a okamžik po tomto updatu - *e*

Strom zlinearizuji jeho in order průchodem do zápisu

b1 b2 e2 b3 e3 e1

Což de facto znamená, že jsem provedl update 1 a z něj update 2 a 3. *Porovnejte se zápisem b1b2b3e3e2e1*

Pokud chci vložit update v node 2, tak to provedu jako

b1 b2 (b4 e4) e2 b3 e3 e1

Z toho jasně poznám, že čtvrtá verze byla provedena jako zpětný update do verze mezi b2 a e2 - tedy nová branch v bodě 2

Pozn. Takto implementovaný model full persistence má opět konstantní asymptotickou složitost *O(1)*

Lazy loading

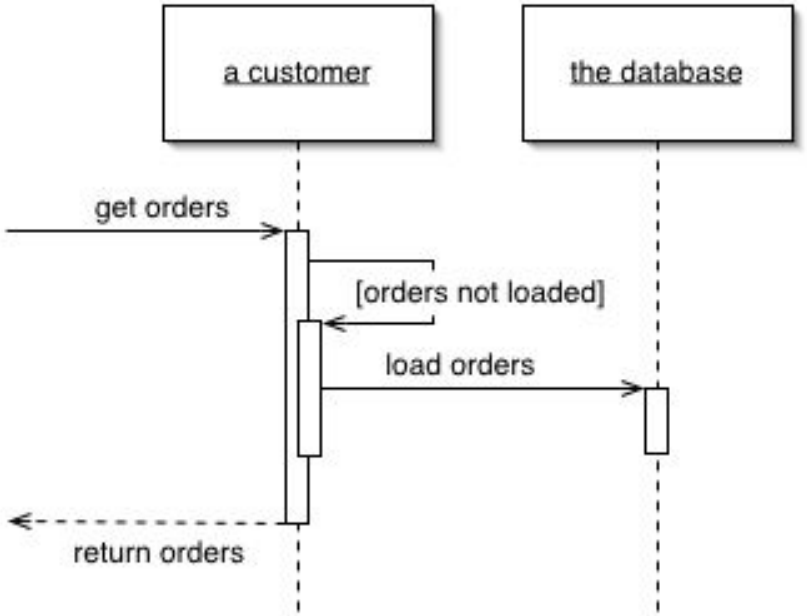
Lazy loading je přístup při kterém odkládáme nahrání objektu z nějakého repository až do doby, kdy ho potřebujeme. Lazy loading použijeme v případě, že nahrávání objektů z repository je náročné nebo nahrávané objekty zabírají nezanedbatelnou část v paměti.

Příklad: Nahráváme entitu *Company*. Tato entita v sobě obsahuje list všech zákazníků. Ke každému zákazníkovi máme uloženou celou řadu dalších entit (adresa, objednávky atd.) Jelikož zákazníků mohou být tisíce, tak budeme chtít dohrávat entity související s konkrétním zákazníkem až ve chvíli, kdy s tímto zákazníkem začneme pracovat v naší aplikaci.

Používají se čtyři druhy implementací lazy loadingu:

- Virtual proxy
- Lazy initialization
- Ghost
- Value holder

Lazy loading



Lazy loading - virtual proxy

Virtual proxy - objekt obsahuje místo referencí na konkrétní objekty (jejichž natažení chci odložit) tzv. virtuální proxy, která je dotáhne až na explicitní zavolání.

Naimplementujeme entity, které drží data o *Company* a *Customer*. List zákazníků je vrácen pomocí metody *ContactList getContactList()* . Ta místo vlastní kolekce objektů bude vracet proxy na tyto objekty.

```
class Company {
   String companyName;
   String companyAddress;
   String companyContactNo;
   ContactList contactList;
   public Company(String companyName, String
companyAddress, ContactList contactList)   {
       this.companyName = ...
   }
   ... field getters ...

   public ContactList getContactList()  {
      return contactList;
   }
}
```

```
class Customer {
   private String customerName;
   private double customerSatisfaction;
   private String customerContact;
   public Customer(String customerName,
      double customerSatisfaction, String
      customerContact)   {
        this.customerName = ....
   }

   ... field getters ...

   public String toString()   {
       return "customerName: " + customerName +
", customerContact : " + ...
   }
```

Lazy loading - virtual proxy

Realizujeme proxy *ContactListProxyImpl*, která si dotahuje obsah kontaktů až při prvním provolání. Tato proxy implementuje interface, který používáme pro dotažení objektů.

```
interface ContactList {
   public List<Customer> getCustomerList();
}
```

```
class ContactListProxyImpl implements ContactList {
   private ContactList contactList;
   public List<Customer> getCustomerList()   {
       if (contactList == null) {
           System.out.println("Fetching list of
employees");
           contactList = new ContactListImpl();
       }
       return contactList.getCustomerList();
   }
}
```

```
class ContactListImpl implements ContactList {
   public List<Customer> getCustomerList()   {
       return getCustList();
   }
   private static List<Customer> getCustList()   {
       List<Customer> custList = new ArrayList<Customer>(5);
       custList.add(new Customer("Novak", 0.3, "Stodolni 5, Ostrava"));
       custList.add(new Customer("Karel Upir", 0.9, "Rumunska 10, Praha"));
       custList.add(new Customer("Barova", 0.5, "K potoku 13, Praha"));
       custList.add(new Customer("John Chainsaw", 1.0, "Austin, Texas"));
       return custList;
   }
}
```

Lazy loading - virtual proxy

```
class LazyLoadingClient {
   public static void main(String[] args)   {
       ContactList contactList = new ContactListProxyImpl();
       Company company = new Company ("ToysforFreaks", "Praha Zizkov", contactList);
       System.out.println("Fetched company:");
       System.out.println("Company Name: " + company.getCompanyName());
       System.out.println("Company Address: " + company.getCompanyAddress());
       System.out.println("Requesting for contact list");
       List<Customer> empList = company.getContactList().getCustomerList();
       for (Customer emp : empList) { System.out.println(emp); }
   }
}
```

Data o zákaznících se dotahují až při zavolání *getCustomerList()* na naší proxy.

=>

```
Fetched company:
Company Name: ToysforFreaks
Company Address: Praha Zizkov
Company Contact No.: +429-777-284589
Requesting for contact list
Fetching list of employees
customerName: Karel Prazak, customerContact : Stodolni 5, Ostrava,
customerSatisfaction : 0.3
customerName: Karel Upir, customerContact : Rumunska 10, Praha,
customerSatisfaction : 0.9
customerName: Jana Barova, customerContact : K potoku 13, Praha,
customerSatisfaction : 0.5
customerName: John Chainsaw, customerContact : Austin, Texas,
customerSatisfaction : 1.0
```

Lazy loading - lazy initialization

Při prvním přístupu k property objektu se testuje na null, v kladném případě se nahrává obsah.

```
enum CarType {none, Audi, BMW,}
class Car {
   private static Map<CarType, Car> types = new HashMap<>();
   private Car(CarType type) {}
   public static Car getCarByTypeNameConcurrent(CarType type)  {
       if (!types.containsKey(type)) {
           synchronized(types)  { //Check after acquired the lock that the instance was not created meanwhile
               if (!types.containsKey(type)) {  // Lazy initialisation
                   types.put(type, new Car(type));
               }
           }
       }
       return types.get(type);
   }

   public static void showAll() {
      if (types.size() > 0) {
      System.out.println("# of instances=" + types.size());
      for (Entry<CarType, Car> entry : types.entrySet()){
          String car = entry.getKey().toString();
          car = Character.toUpperCase(Car.charAt(0)) + car.substring(1);
          System.out.println(car);
      }
}
```

Lazy loading - lazy initialization

```
class Client {
   public static void main(String[] args)
   {
       Car.getCarByTypeName(CarType.BMW);
       Car.showAll();
       Car.getCarByTypeName(CarType.Audi);
       Car.showAll();
       Car.getCarByTypeName(CarType.BMW);
       Car.showAll();
   }
}
```

=>

```
Number of instances made = 1
BMW

Number of instances made = 2
Audi
BMW

Number of instances made = 2
Audi
BMW
```

Lazy loading - value holder

Value Holder je velice podobný Virtuální proxy, používá se však většinou v případě, kdy jsou potřeba generika. Sám ValueHolder se stará o lazy loading. K získání hodnoty obaleného objektu je volána metoda *GetValue()*, která přistupuje k databázi pouze poprvé, kdy je tato metoda volána, dále už jen vrací inicializovaný objekt.

```
public class StorageBox {
   private ValueHolder<String> storedProducts;
   private ValueHolder<Integer> productsCount;

   public StorageBox(ValueHolder<String>
storedProducts, ValueHolder<Integer> productsCount) {
       this.storedProducts = storedProducts;
       this.productsCount = productsCount;
   }

   public ValueHolder<Integer> getProductsCount() {
       return productsCount;
   }

   public ValueHolder<String> getStoredProducts() {
       return storedProducts;
   }
}
```

```
public class ValueHolder<T> {
   private T value;
   private Function<Object, T> valueLoader;

   public ValueHolder(Function<Object, T>
valueLoader){
       this.valueLoader = valueLoader;
   }

   public T GetValue(Object parameter) {
       if (value == null)
           // physical data fetch is done here
           value =
valueLoader.apply(parameter);
       return value;
   }
}
```

Lazy loading - value holder

Vytvořili jsme objekt StorageBox a iniciovali jsme ho funkcemi, které na explicitní zavolání vrátí data.

```
public class Client {
   public static void main(String[] args){
       Function<Integer,String> loader = a -> "Products data to depth " + a ;
       ValueHolder productsDataVH = new ValueHolder(loader);

       Function<Integer, Integer> counter = (a) -> 10 + a ;
       ValueHolder productsCountVH = new ValueHolder(counter);

       StorageBox sb = new StorageBox(productsDataVH, productsCountVH);

       System.out.println("Products data: " + sb.getStoredProducts().GetValue(3));
       System.out.println("Products count: " + sb.getProductsCount().GetValue(10));
   }
}
```

=>

```
Products data:Products data to depth 3
Products count:20
```

Lazy loading - ghost

Ghost, neboli duch, je implementace lazy loadingu, kdy reálný objekt existuje jen v částečném stavu.

Příkladem je nahrávání grafu objektů z databáze, kdy místo instancí některých objektů z grafu pracujeme pouze s jejich identifikátory (id). Plné objekty jsou dohrávány pomocí těchto unikátních identifikátorů až ve chvíli, kdy je opravdu potřebujeme.

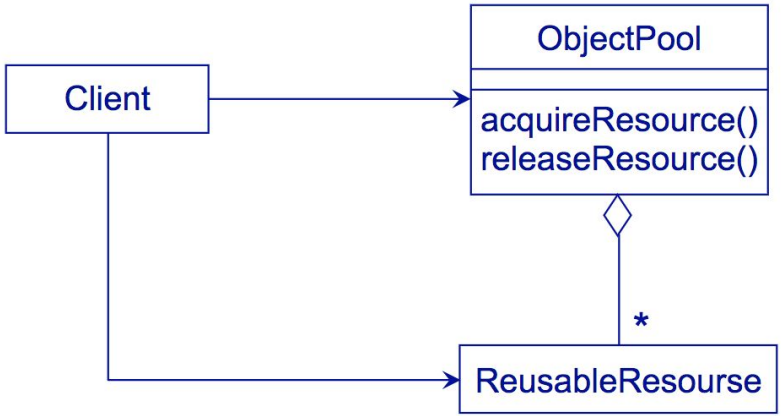
Tento mechanismus hojně využívají *ORM (Object Relationship Mapping)* nástroje jako např. Hibernate, kdy při natažení grafu objektů z databáze obsahují některé entity místo vazeb přímo na delší entity pouze jejich ids a vy si je dotahujete přes tato ids až když je potřebujete. ORM tool sám poskytuje konfigurační možnost, kdy volíte do jaké úrovně se budou dotahovat kompletní objekty a od které pouze jejich unikátní identifikátory.

Object pool

Object pool je de facto zásobárna přepoužitelných resources, které jsou náročné na vytvoření a proto si je po vytvoření raději ponecháme pro další použití místo toho, abychom je likvidovali a vytvářeli znovu a znovu.

Příklady jsou pooly databázových spojení, file handles, proxy na provolání externí služby, java objekty vašeho programu, které jsou náročné na vytvoření a přitom se dají přepoužívat.

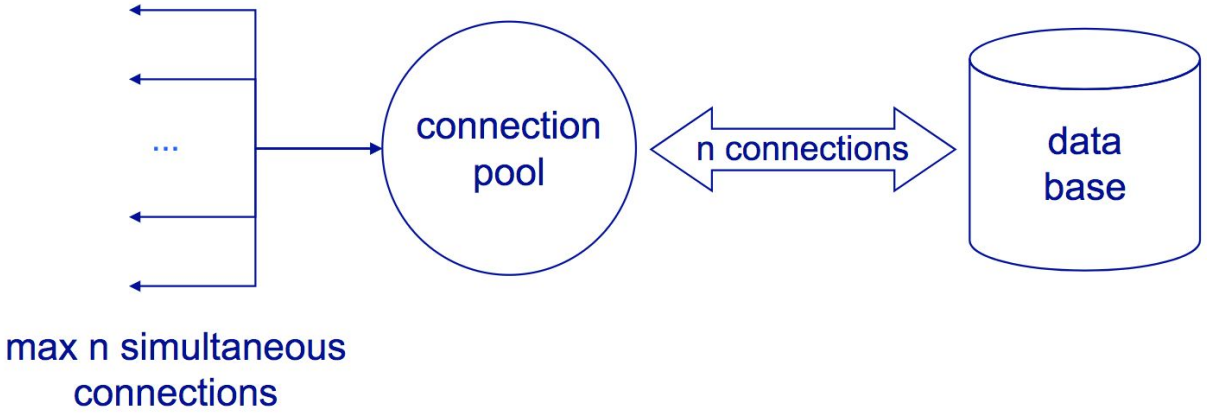
Pooling těchto resources poskytuje výrazné zlepšení performance zejména v případech, kdy se jsou náročné na vytvoření a používají se velmi masivně. Výhoda pooling je také, že máte predikovatelný čas a memory footprint při práci se zdroji.



Object pool

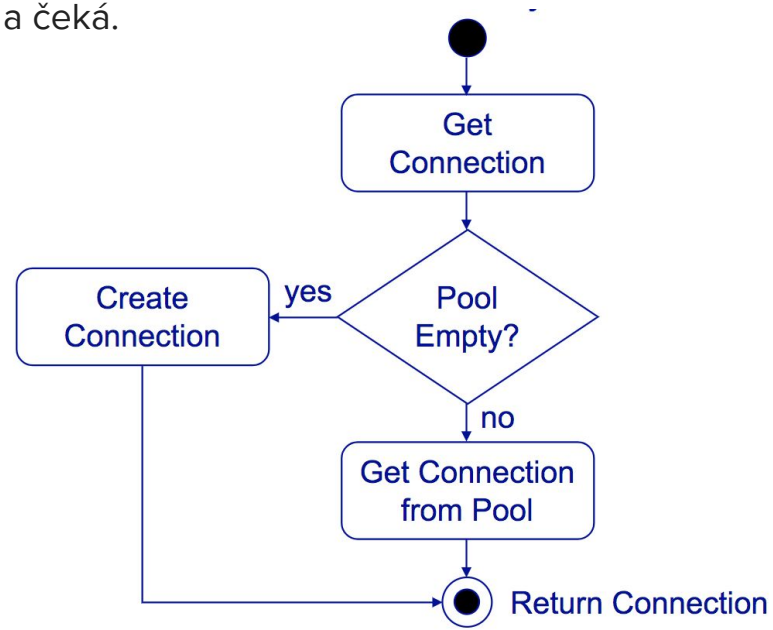
Object pool má parametry, kterými dále ladíme propustnost vašeho programu, maximalizujete memory footprint atd.:

- *Maximální počet resources* - množství resources je omezeno zhora => maximalizace memory footprint
- *Minimální počet resources* - množství resources je omezeno zdola => resources se vytvoří dopředu, takže nemusíte čekat ani při prvních voláních, kdy ještě nejsou resources vytvořeny
- Doba nepoužívání po které se resource likviduje => zamezujete tomu, aby v poolu byly drženy nepoužívané resources



Object pool - database connection

Dobrým příkladem je mechanismus poolingu spojení k databázi v JDBC API. JDBC poskytuje v API třídy *PooledConnection* - spojení a *ConnectionPoolDataSource* - datový zdroj. Z pohledu programátora neexistuje žádný rozdíl mezi standardními a poolovanými spojeními. Když je spojení uzavřeno, tak je vráceno do poolu, takže je k dispozici pro opětovné použití. Množství připojení je omezeno zhora pro každého klienta, který pracuje s databází, aby mohly být databázové zdroje dobře sdíleny. V případě, že nejsou volné spojení, tak je klientský thread blokován a čeká.



Object pool

```
public class PooledObject {
   public String temp1;
   public String temp2;
   public String temp3;
   public String getTemp1() {
       return temp1;
   }
   public void setTemp1(String temp1) {
       this.temp1 = temp1;
   }
   public String getTemp2() {
       return temp2;
   }
   public void setTemp2(String temp2) {
       this.temp2 = temp2;
   }
   public String getTemp3() {
       return temp3;
   }
   public void setTemp3(String temp3) {
       this.temp3 = temp3;
   }
}
```

```
public class ObjectPool {
   private static long expTime = 60000;//6 seconds
   public static HashMap<PooledObject, Long> available = new
HashMap<PooledObject, Long>();
   public static HashMap<PooledObject, Long> inUse = new HashMap<PooledObject,
Long>();
   public synchronized static PooledObject getObject() {
       long now = System.currentTimeMillis();
       if (!available.isEmpty()) {
           for (Map.Entry<PooledObject, Long> entry : available.entrySet()) {
               if (now - entry.getValue() > expTime) { //object has expired
                   popElement(available);
               } else {
                   PooledObject po = popElement(available, entry.getKey());
                   push(inUse, po, now);
                   return po;
               }
           }
       }
       // either no PooledObject is available or each has expired, so return a
new one
       return createPooledObject(now);
   }
```

Object pool

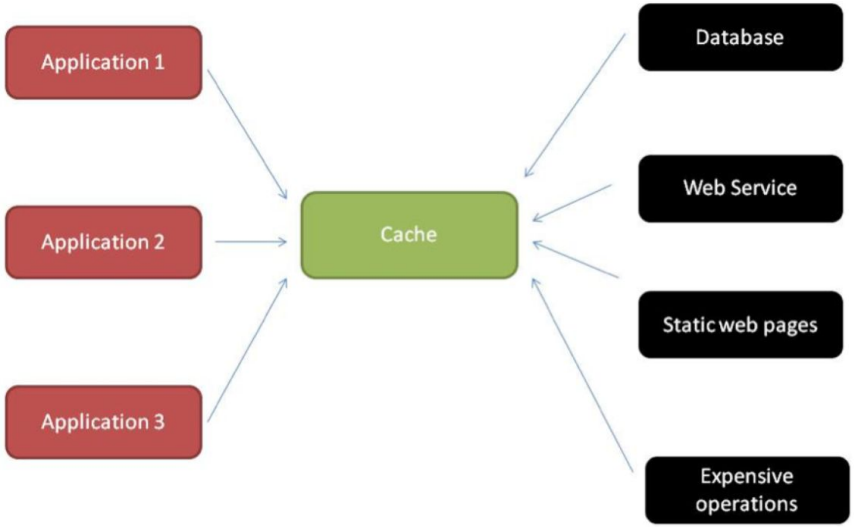
```
   private synchronized static PooledObject createPooledObject(long now) {
       PooledObject po = new PooledObject();
       push(inUse, po, now);
       return po;
   }
   private synchronized static void push(HashMap<PooledObject, Long> map,
                                         PooledObject po, long now) {
       map.put(po, now);
   }
   public static void releaseObject(PooledObject po) {
       cleanUp(po);
       available.put(po, System.currentTimeMillis());
       inUse.remove(po);
   }
   private static PooledObject popElement(HashMap<PooledObject, Long> map) {
       Map.Entry<PooledObject, Long> entry = map.entrySet().iterator().next();
       PooledObject key= entry.getKey(); //Long value=entry.getValue();
       map.remove(entry.getKey());
       return key;
   }
   private static PooledObject popElement(HashMap<PooledObject, Long> map, PooledObject
key) {
       map.remove(key);
       return key;}
   public static void cleanUp(PooledObject po) {
       po.setTemp1(null);
       po.setTemp2(null);
       po.setTemp3(null);
}}
```

Cache

Cache použijeme v případě, že chceme načíst resource nebo data jednou, ale používat vícekrát. Jedná se o další design pattern, který se hojně používá pro zlepšení performance.

Cache funguje takto: Aplikace požaduje data z cache pomocí klíče. Pokud klíč není nalezen, aplikace načte data z externího zdroje dat (pomalého) a vloží je do cache. Další požadavek na klíč je obsluhován z cache.

Cache nalezneme na různých vrstvách systému, většinou se používá několik cache najednou (jedna na každé vrstvě)



Pozn.: Design pattern object pool se liší od cache v tom, že spravuje řádově menší množství resources, resources se od sebe neliší a de facto je zapůjčují klientským threadům dokud ty je nevrátí.

Cache

Performance charakteristika cache

Hlavní charakteristikou výkonnosti cache je poměr ***hit / miss ratio***. Vysoký poměr znamená efektivní cache. Nízký poměr naopak znamená, že buď se cachují data, která by se neměla cachovat nebo je cache příliš malá => optimalizace cache, aby se necachovala nevhodná data a aby cache měla optimální velikost.

Cache eviction policy

Algoritmus podle kterého jsou prvky odebírány z cache a naopak jsou přidávány nové tak, aby nebyla překročena kapacita cache. Jako vhodný se jeví algoritmus LRU (Last Recently Used). Typickou implementací LRU eviction policy je *Map* a *LinkedList*. Map drží cachované prvky, které se vybírají podle klíče a LinkedList pak udržuje pořadí v jakém byly prvky naposledy použity (znovu čtený nebo nově přidaný prvek je na začátku seznamu)

Faktory ovlivňující řešení cache

- TTL (Time to Live)
- Frekvence čtení a zápisu
- Škálovatelnost

Existuje celá řada caching frameworků, které se liší dle účelu, škálovatelnosti, distribuovatelnosti

- EHCache
- Redis
- ExtremeScale
- Hazelcast

Pozn. Realizace enterprise class cache zahrnuje implementaci netriviálních operací jako je zajištění koherence (data jsou konzistentní přes více zejména distribuovaných instancí cache), invalidace změněných dat, elastické škálování atd.

Application cache v Java

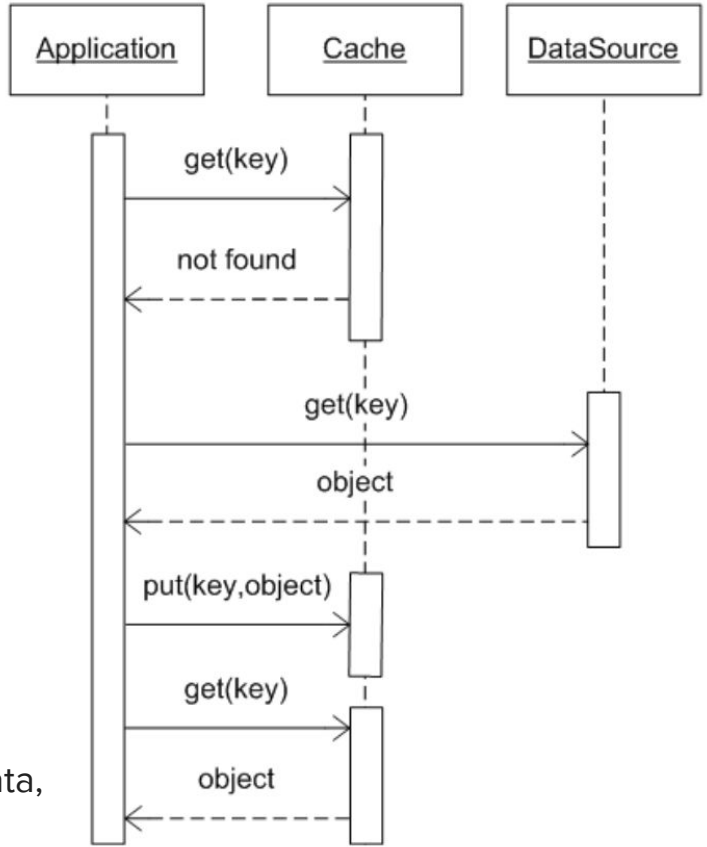
Application cache je cache, ke které aplikace přistupuje napřímo. Aplikace využívá toho, že nejčastěji používaná data jsou uložena v paměti.

Základní cache pattern v Java má následující vlastnosti:

- Serializované Java objekty
- Uložení v paměti nebo souboru
- Objekty jsou uloženy jako key-value páry

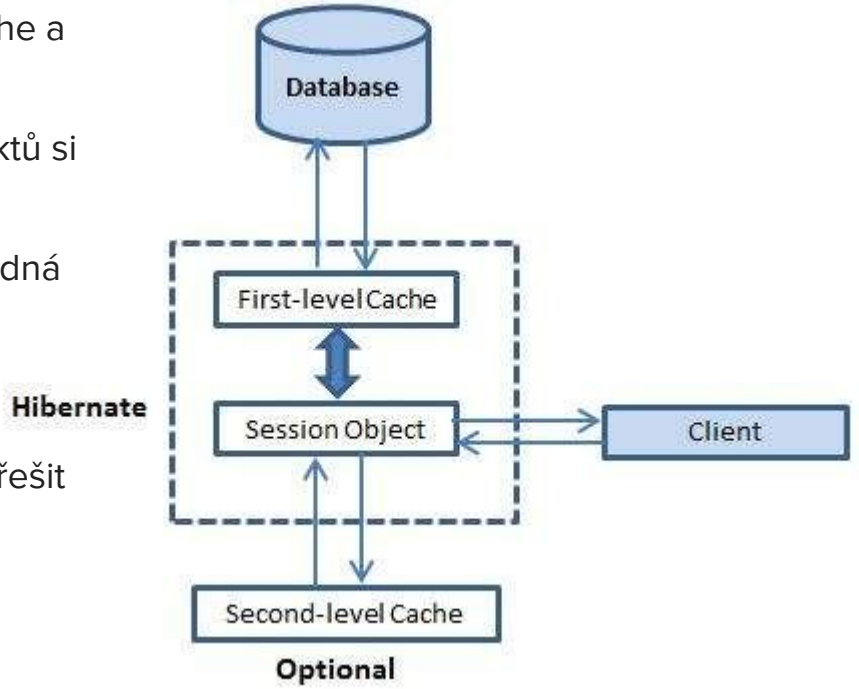
Požití je např. pro:

- Statická data
- Číselníky
- Pseudo statická data - obsah katalogu, směnný kurz atd.
- Kontextově specifická data - uživatelský profil, session data, data z externích systémů



Cache v ORM toolech (např. Hibernate)

- **L1 cache** - její scope je *Session* (uživatelská session) - s ní vzniká a zaniká. Je unikátní pouze pro daného uživatele (jeho session). V prvním kroku se dotazuje právě L1 cache a až pak se dotazují další úrovně cache.
- **Query cache** - funguje jako L1 cache, pouze místo objektů si pamatuje databázový dotaz a co ten dotaz vrátil
- **L2 cache** - její scope je *SessionFactory* (factory odpovědná za vytváření uživatelských session) a je sdílená mezi transakcemi a uživateli. Výhodou L2 cache je, že se dostaneme i k datům a dotazům, které provedli ostatní uživatelé. Nevýhodou je, že musí být dost velká a musí řešit neaktuální (stale) data.



Cache

Příklad implementace jednoduché in memory cache pro Java objekty

Konstruktor obsahuje parametry specifikující kapacitu a cache eviction policy.

Konstruktor startuje thread, který čistí cache od starých (dlouho nepoužitých) objektů

Pozn. *LRUMap* a *MapIterator* je z *org.apache.commons.collections*

```
public class InMemoryCache<K, T> {
   private long timeToLiveInSec;
   private LRUMap cacheMap;
   protected class cacheObject {
       public long lastAccessed = System.currentTimeMillis();
       public T value;
       protected cacheObject(T value) {
           this.value = value;
       }
   }
  public InMemoryCache(long timeToLiveInSec, final long cleanupInterval, int maxItems){
       this.timeToLiveInSec = timeToLiveInSec * 1000;
       cacheMap = new LRUMap(maxItems);
       if (timeToLiveInSec > 0 && cleanupInterval > 0) {
           Thread t = new Thread(new Runnable() {
               public void run() {
                   while (true) {
                       try {
                           Thread.sleep(cleanupInterval * 1000);
                       } catch (InterruptedException ex) {}
                       cleanup();
                   }
               }
           });
           t.setDaemon(true);
           t.start();
       }
   }
```

Cache

```
public void put(K key, T value) {
    synchronized (cacheMap) {
        cacheMap.put(key, new cacheObject(value));
    }
}
public T get(K key) {
    synchronized (cacheMap) {
        cacheObject c = (cacheObject) cacheMap.get(key);
        if (c == null)
            return null;
        else {
            c.lastAccessed = System.currentTimeMillis();
            return c.value;
        }
    }
}
public void remove(K key) {
    synchronized (cacheMap) {
        cacheMap.remove(key);
    }
}
public int size() {
    synchronized (cacheMap) {
        return cacheMap.size();
    }
}
```

Cache

Při *cleanup()* z cache odstraňujeme snepoužívané objekty. Cleanup se dělá v druhém cyklu, protože použitá Map neumožňuje mazat prvky v průběhu iterování

```
    public void cleanup() {
        long now = System.currentTimeMillis();
        ArrayList<K> deleteKey = null;
        synchronized (cacheMap) {
            MapIterator it = cacheMap.mapIterator();
            deleteKey = new ArrayList<K>((cacheMap.size() / 2) + 1);
            K key = null;
            cacheObject c = null;
            while (it.hasNext()) {
                key = (K) it.next();
                c = (cacheObject) it.getValue();
                if (c != null && (now > (timeToLiveInSeconds +
c.lastAccessed))){
                    deleteKey.add(key);
                }
            }
        }

        for (K key : deleteKey) {
            synchronized (cacheMap) {
                cacheMap.remove(key);
            }
            Thread.yield();
        }
    }
}
```