Životní cyklu a likvidace objektů

Závislosti mezi objekty

Vložení závislostí

- *Dependency Injection* (DI) je postup, kdy deklarativním způsobem (přímo uvnitř javového kódu nebo vnějším popisovačem, např. XML) označíme, kde je očekáváno na začátku životního cyklu objektu vložení odkazu na objekt, na němž náš závisí - tedy *vložení závislosti*.

Příklad bez DI

- Příklad použití závislosti:
- objekt manažeru dat závisí na objektu databáze = potřebuje ho pro svou činnost (metodu `getData`);
- objekt databáze se do něj dostane už při *konstrukci* = na začátku životního cyklu manažeru;
- objekt `database` může nebo nemusí (časteji) být po dobu života manažeru vyměněn.
- Výhodné je, když je `Database` rozhraní - může být implementováno různými třídami.

```
public class DataManager {
    private Database database;
    public DataManager(Database db) { this.database = db; }
    public Data getData() { return database.getData(); }
}
```

Variantně bez DI

- Nastavení závislosti metodou `set`
- Nevýhodou je, že mezi konstrukcí a nastavením závislosti je objekt manažeru očividně nepoužitelný.

```
public class DataManager {
    private Database database;
    public DataManager() { }
    public void setDatabase(Database db) { this.database = db; }
    public Data getData() { return database.getData(); }
}
```

Životní cyklus bez DI

- Nastavení závislosti musíme provést sami.
- Nejdříve vytvoř prvý objekt - databáze, poté druhý objekt (manažer dat) a propoj závislost.

```
public static void main(String[] args) {
    // database is hardwired here - only MyDatabase is used & no other
    Database db = new MyDatabase(...);
    DataManager manager = new DataManager();
    manager.setDatabase(db);
    // now the manager is ready to give data
    Data data = manager.getData();
}
```

EJ: Tip XXX: Prefer dependency injection over hardwiring dependencies

- Existují nástroje, které zvnejšku zajistí, že do objektu je přidán vhodný objekt závislosti.
- Například do manažeru dat je přidána správná databáze = vhodný objekt implementující rozhraní `Database`.
- Vnější vložení závislosti je posláním rámců (kontejnerů) pro vkládání závislostí, *dependency injection frameworks*.
- Velmi jednoduchý rámec pro DI je Google Guice
- Populární je *Spring Framework*, ale ten je významně obsáhlejší a kvůli samotnému DI je kanónem na vrabce.

Příklad s Guice

- Příklad viz https://github.com/google/guice/wiki/Motivation
- Třída `RealBillingService` závisí na `CreditCardProcessor` a `TransactionLog`

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,
      TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }
```

```
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Konfigurace DI v Guice

- Náš Module musí implementovat rozhraní Guice `Module`
- V konfiguraci přiřadíme každému rozhraní (vlevo) třídu (vpravo), kterou v jeho roli budeme používat.

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

Vlastní "sdrátování" DI v Guice

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector(new BillingModule());
    BillingService billingService = injector.getInstance(BillingService.class);
    ... // the TransactionLog and CreditCardProcessor follow
  }
```

Avoid unnecessary objects

Eliminate obsolete object references

Avoid finalizers and cleaners

- Finalizér, tzn. metoda `finalize()` vlastní všem objektům, může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup při zániku objektu - sestávající obvykle z uvolnění systémových zdrojů - síťové sokety, spojení na databázi atd.
- V Javě nicméně není zaručeno, že se finalizér skutečně zavolá - JVM jej nemusí volat, pokud nepotřebuje fyzicky uvolnit paměť obsazenou (již nepoužívaným) objektem.
- I kdyby finalizér zavolán byl, zůstává problém s určením okamžiku, *kdy* je volán a v jakém pořadí jsou finalizéry na mrtvých objektech volány.
- Celkově tedy na `finalize()` nespoléhat a nepoužívat je - zdroje uvolňovat explicitním zavoláním vhodné metody.