Dependency Injection (DI)

Dependency Injection: Motivation

```
public class Musician {
   private Guitar g;
   public Musician(){ g = new Guitar(); }
   public void perform(){ g.play(); }     }

class Guitar {
   public void play() {
               System.out.println("Gitarrenklänge"); }   }
```

- An sich ist der Code zu speziell formuliert
  - Es gibt nicht nur Gitarristen!.
- Unsere Fähigkeit (perform) hängt vom Instrument ab. Dieses instanziieren wir selbst.

Verwendung von Schnittstellen

```
public interface Instrument {  void play(); }

public interface Performer {  void perform(); }

public class Musician implements Performer {
   private Instrument g;
   public Musician(){ g = new Guitar(); }
   public void perform(){ g.play(); } }

class Guitar implements Instrument {
   public void play() {
               System.out.println("Gitarrenklänge"); }   }
```

- So allgemein wie möglich, so speziell wie nötig

Abhängigkeit nicht selbst auflösen

```
public interface Instrument {  void play(); }

public interface Performer {  void perform(); }

public class Musician implements Performer {
   private Instrument g;
   public Musician(Instrument i){ g = i; }
   public void perform(){ g.play(); } }

class Guitar implements Instrument {
   public void play() {
               System.out.println("Gitarrenklänge"); }   }
```

Dependency Injection (DI)

- Durch die Übergabe der benötigten Attribute von außen, erreichen wir eine gute Entkopplung der einzelnen Bausteine.
- Der Aufrufende Kontext übernimmt nun die konkrete Instantiierung und versorgt die Bausteine per Konstruktor oder auch Setter-Methoden mit Ihren benötigten Referenzen.
  - **Das ist abstrakt betrachtet, die Konfiguration der Anwendung!**
- Das Spring-Framework betrachtet das als eines der zentralen Prinzipien.
  - Die Konfiguration erfolgt hierbei deskriptiv:
    XML-Datei, Annotationen oder in Java selbst (ab Version 3.0)
- Neben Spring gibt es weitere Frameworks mit Fokus auf DI:
  - Google Guice
  - JBoss Seam
  - Context and Dependeny Injection (CDI) ab JEE 6 als Erweiterung der bisherigen DI-Möglichkeiten in JEE 5

DI im JEE-Container

```
...
Properties props = new Properties();
props.put("java.naming.provider.url",
          "jnp://localhost:1099");
InitialContext context = new InitialContext(props);
HelloWorldRemote hello =
   (HelloWorldRemote)context.lookup("HelloWorld/remote");
DataSource myDS =
    (DataSource) naming.lookup( "jdbc/Oracle" );
...
```

- In komplexen Szenarien und vor allem im verteilten Fall werden die benötigten Referenzen .über einen Auskunftsdient, in der Regel JNDI, erfragt.
- Leider ist der Zugriff auf die Auskunft und die Namensgebung nicht hinreichend standardisiert
  - Deshalb ist es hier sehr viel einfacher, wenn die Umgebung die benötigten Ressourcen injiziert (nur möglich innerhalb der selben VM)

Beispiel für DI via Annotationen im JEE-Container

```
...
@Resource
private DataSource myDS;
...
private HelloWorldRemote myservice;
@EJB
public setHelloWorldRemote (HelloWorldRemote service) {
        myservice = service; }
...
```

- Injizierung via Annotationen, die in diesem Beispiel durch den Sun/Oracle-JEE-Standard festgelegt sind.

Dependency Injection

Vorteile:

- Der Entwickler der nutzenden Komponente kennt nur die Schnittstellen seiner Abhängigkeiten, keine Implementierungsdetails oder Infrastrukturunterschiede.
- Die Bausteine lassen sich gut testen.

Nachteile:

- Da die Injizierung in der Regel auf jeden Fall erfolgt, sollte die Abhängigkeit dann auch wirklich benötigt werden (insbesondere bei aufwendigen Resourcen)
- Die Fehlersuche wird schwieriger, da der Compiler die Abhängigkeiten weniger prüfen kann.