

Pokročilé programování na platformě Java:

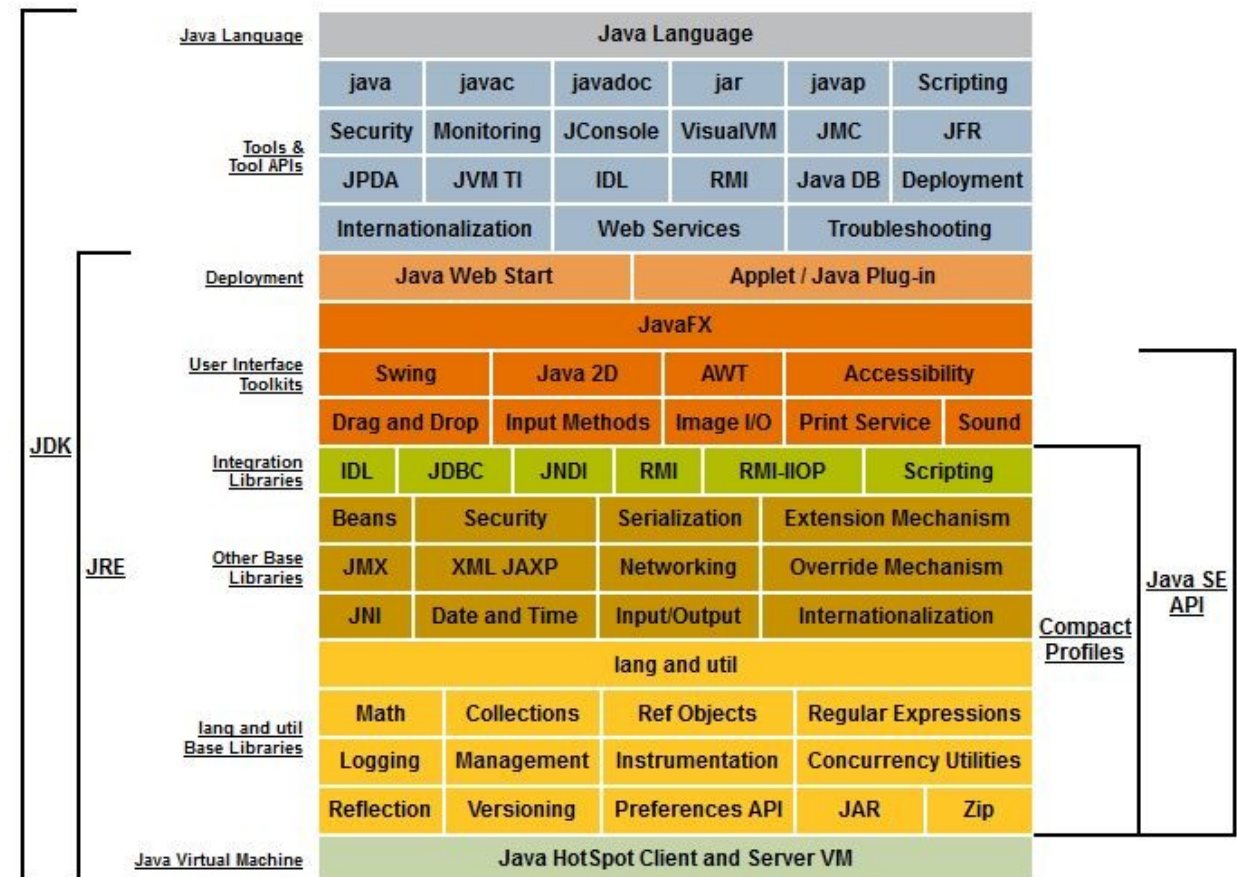
Framework Spring

Programovací jazyk a platforma Java

- Jazyk Java
 - vznikl původně pro interní potřeby Sun Microsystems (nyní součást Oracle)
 - existuje několik implementací
 - Sun v roce 2006 svoji implementaci zveřejnil (licence GPL)
- Platforma Java
 - Sada programů umožňujících vývoj a běh programů napsaných v jazyce Java
 - Kompilátor, knihovny funkcí, JVM,
 - Skládá se z několika dílčích platforem:
 - Java SE - definice jazyka a základní knihovny
 - Java EE - rozšíření pro tvorbu větších infrastruktur
 - Java ME – mobilní aplikace
 - Tyto platformy spojují společné koncepty: jazyk Java, JVM,

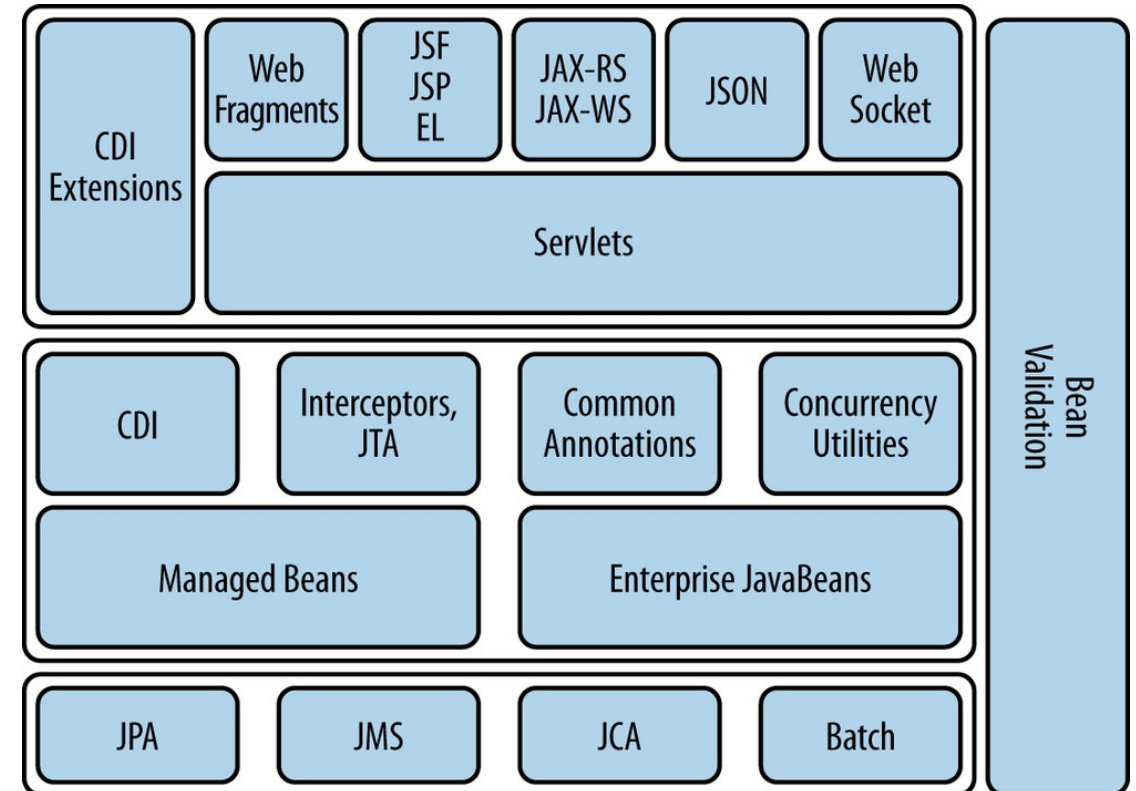
Platforma Java SE

- Součásti
 - JVM
 - Knihovny
 - Práce se soub. systémem
 - Síť
 - Databáze
 - Core API
 - Kolekce
 - Datové typy
 - ...
- Distribuována v různých edicích:
 - JRE – běhové prostředí
 - JDK – JRE + nástroje pro vývoj (debug)



Platforma Java EE

- Základem je Java SE
- Umožňuje navíc tvorba škálovatelných a více vrstevých architektur
- Aplikace komunikují především pomocí sítě
- Aplikace jsou rozděleny do vrstev
 - Klientská
 - Servlety
 - Java Server Pages (JSP)
 - Java Server Faces (JSF)
 - Doménová (business)
 - Java Persistence API (JPA)
 - Enterprise Java Beans (EJB)
 - Datová
 - JPA
 - JDBC (Database Connectivity)
 - JTA (Transaction API)



Spring (<http://spring.io>)

- Vznikl jako alternativní platforma k Java EE (od 2002)
- Snaha vyřešit problémy Java EE platformy
 - Komplexnost - EJB
 - Špatná testovatelnost
 - Spousta boilerplate kódu (nutné duplikovat)
 - Pomalý vývoj, nemodulární
- Lepší podpora pro aktuální technologie
 - Cloud
 - BigData
- Podpora i pro jiné jazyky (Groovy, Scala)
- Technologicky kompatibilní s Java EE – sdílí specifikace
- OpenSource projekt
- Modulární systém

Spring – moduly

- Moduly, které budeme používat
 - Core
 - Data - Relační i NoSQL (MySQL, MongoDB)
 - Web - MVC, Rest
 - Hateoas
 - AMPQ - Rabbitmq
 - Security
- Další moduly
 - Social - Facebook, Twitter, Google
 - Cloud
 - Data - Haddop, Elasticsearch, Cassandra
 - Grails - Ruby on Rails na JVM (v Groovy)

Co je to Spring.IO?

- Moduly Springu existují v různých verzích
 - Vzniká řada problémů
 - Např. jaké verze jsou vzájemně kompatibilní?

=>

- Spring.IO
 - Označení pro verzovanou platformu zastřešující jednotlivé moduly Springu
 - Definuje jako celek různé funkce a možnosti celé platformy pro vývoj aplikací
 - Definuje vzájemně kompatibilní verze jednotlivých modulů
 - Spring IO lze využít v rámci Dependency Managementu při konfiguraci projektu pomocí Mavenu
 - POM platformy Spring IO je možné a) importovat nebo b) použít jako parent POM (viz minulá před.)
 - Není pak nutné definovat verze u jednotlivých importovaných modulů !!

Spring.IO – příklad (import POMu Spring.IO):

```
<groupId>com.example</groupId>
<artifactId>your-application</artifactId>
<version>1.0.0-SNAPSHOT</version>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.spring.platform</groupId>
      <artifactId>platform-bom</artifactId>
      <version>2.0.3.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
</dependencies>
```


Proč používat Spring?

- Viz výše:
 - Celá řada modulů a nadstavbových knihoven na Javou
 - Odlehčení oproti JEE
 -
- +
- Využívá návrhový vzor (princip) Inversion of Control (IoC) – inverze kontroly

Motivace pro použití Springu:
Problém svázání se s implementací
a inverze řízení

Java: svázání se s implementací – příklad

- Dvě třídy, každá tiskne denní zprávu jiným způsobem

```
public class DynamicMessageOfTheDay {  
    private final String[] messages = new String[] {  
        "Hello, today is Sunday",  
        .....  
        "Hello, today is Saturday"  
    };  
    public String getMessage() {  
        return messages[GregorianCalendar.getInstance().get(Calendar.DAY_OF_WEEK) - 1];  
    }  
}  
  
public class BasicMessageOfTheDay {  
    private String myMessage = "Hello";  
    public String getMessage() { return myMessage; }  
    public void setMessage(String message) { this.myMessage=message; }  
}
```

Java: svázání se s implementací – příklad

- Třída MessagePrinterService je zaobalující třída pro tisk
 - Vnitřně využívá jednu z předchozích tříd

```
public class MessagePrinterService {  
    private BasicMessageOfTheDay service = new BasicMessageOfTheDay ();  
    public void printMessage () {  
        System.out.println(service.getMessage());  
    }  
}  
  
public class Main {  
    public static void main(String[] args){  
        MessagePrinterService myMessagePrinter = new MessagePrinterService();  
        myMessagePrinter.printMessage();  
    }  
}
```

Java: svázání se s implementací – problémy

- Třída `MessagePrinterService` je pevně svázána s konkrétní třídou `BasicMessageOfTheDay`

=> problémy:

- Ve třídě `MessagePrinterService` vytvářím objekt typu `BasicMessageOfTheDay`
 - Musím umět nastavit a inicializovat všechny jeho parametry
 - Co když jde o složitou třídu využívající připojení k databázi a další třídy?
 - Je třeba zásah do kódu při
 - Změně parametrů třídy `BasicMessageOfTheDay` (konstruktor, veřejná metoda, ..)
 - Co když tuto třídu vytváří jiný vývojář nebo jiný tým? Jak se o změně dozvím já?
 - Když chci uvnitř `MessagePrinterService` použít jinou třídu, která dělá podobnou věc, ale trochu jinak (`DynamicMessageOfTheDay`)

Java: programování do interface – varianta 1

- Vytvoříme rozhraní, které budou třídy Basic.. a Dynamic implementovat:

```
public interface MessageOfTheDayService {
    public String getMessage();
}

public class DynamicMessageOfTheDay implements MessageOfTheDayService ...
public class BasicMessageOfTheDay implements MessageOfTheDayService ...

public class MessagePrinterService {
    private MessageOfTheDayService myService;
    public void printMessage () {
        System.out.println(myService.getMessage());
    }
    public void setMyService(MessageOfTheDayService service) {
        this.myService = service;
    }
}

    public class Main {
        public static void main(String[] args){
            MessageOfTheDayService myMessageService = new BasicMessageOfTheDay();
            MessagePrinterService myMessagePrinter = new MessagePrinterService();
            myMessagePrinter.setMyService(myMessageService);
            myMessagePrinter.printMessage();
        }
    }
```

Java: programování do interface – v.1 – vlastnosti

- Třída `MessagePrinterService` je nyní pevně svázána s rozhraním `MessageOfTheDayService`

⇒ Výhody:

- Většina kódu programu funguje stejně pro různé implementace rozhraní
 - Kód je odstíněn od konkrétní implementace
 - Problém ovšem nastává při změně rozhraní!

=> Přetrvávají stále tyto problémy:

- Ve třídě `Main` musím umět nastavit a inicializovat všechny parametry konkrétní třídy, kterou jsem vybral a která rozhraní implementuje
 - Co když jde o třídu se složitým nastavením využívající připojení k databázi a další třídy?
- Je třeba zásah do kódu třídy `Main`
 - Při změně parametrů třídy `BasicMessageOfTheDay` (konstruktor, veřejná metoda, ..)
 - Co když tuto třídu vytváří jiný vývojář nebo jiný tým? Jak se o změně dozvím já?
 - Když chci použít uvnitř `Main` jinou třídu (`DynamicMessageOfTheDay`)

Java: programování do interface – varianta 2

- Dtto, ale inicializace třídy MessagePrinterService přes konstruktor

```
public interface MessageOfTheDayService {
    public String getMessage();
}

public class DynamicMessageOfTheDay implements MessageOfTheDayService ...
public class BasicMessageOfTheDay implements MessageOfTheDayService ...

public class MessagePrinterService {
    private final MessageOfTheDayService myService;
    public void printMessage () {
        System.out.println(myService.getMessage());
    }
    public MessagePrinterService(MessageOfTheDayService service) {
        this.myService = service;
    }
}

public class Main {
    public static void main(String[] args){
        MessagePrinterService myMessagePrinter = new MessagePrinterService(new BasicMessageOfTheDay());
        myMessagePrinter.printMessage();
    }
}
```


Java: programování do interface – v.2 – vlastnosti

```
public class MessagePrinterService {  
    private final MessageOfTheDayService myService;  
    public MessagePrinterService(MessageOfTheDayService service) {  
        this.myService = service;  
    }  
}
```

- Platí vše jako při inicializaci přes metodu set (setter) **ALE většinou lepší varianta**
- Inicializace přes konstruktor (a současně existence pouze jednoho konstrukturu) totiž zaručuje, že nemohu vytvořit příslušnou třídu (MessagePrinterService) bez toho, aniž bych nastavil danou vnitřní proměnnou (myService)
 - => Nelze předat instanci bez nastavení myService - mohlo by vyvolat NPE (NullPointerException) !!
- Použití slova final při deklaraci proměnné myService navíc dále zaručuje, že kompilátor hlídá, zda je tato proměnná inicializována
 - ⇒ Nelze definovat ani žádný jiný (prázdný) konstruktor třídy MessagePrinterService, který by tuto proměnnou neinicializoval !!

Jak problém svázání se s
implementací vyřešit lépe?

Pomocí principu
„Inversion of Control“

Spring: Inversion of Control (IoC)

- Závislosti mezi třídami existují v rámci jejich definice stále stejně
- Při inicializaci programu jsou ale konkrétní instance tříd vytvořeny pomocí kontejneru frameworku Spring na základě připravené konfigurace.
- Takto vytvořené instance jsou pak použity v místech programu, kde jsou deklarovány příslušné závislosti = Dependency Injection (DI)
- **Proč název „inverze kontroly“?**
 - Jde o to, že kontrolu nad vytvářením instancí tříd přebírá Spring
 - Na základě dodané konfigurace

Výhody IoC (vzájemně souvisí)

- V místě, kde jsem deklarovat závislost na třídě, nepotřebuji vytvořit konkrétní instanci a nemusím nastavovat všechny její parametry!
 - Spring instanci vytvoří na základě konfigurace a pak ji použije (Dependency Injection)
- Kontejner může podle konfigurace vytvořit jen jednu instanci třídy a tu použít na více místech programu!
 - Hodí se, pokud daný objekt reprezentuje např. připojení k databázi
- Kontejner lze konfigurovat pomocí anotací (v kódu) či externě pomocí XML!
 - Konfiguraci může vytvořit (dodat) někdo jiný!
 - Konfigurovatelnost kontejneru umožňuje
 - Pracovat s různými profily, kdy např. v rámci testování využívá aplikace jiný objekt pro připojení k databázi než v produkčním režimu!
 - Vybírat různé implementace jednoho rozhraní, aniž by bylo třeba provádět zásahy do programu!

Jak sehnat pizzu

1. Udělat si ji sám (Řešení 1)
 - Ingredience, trouba, čas, výsledek
2. Zajít do pizzerie (Řešení 2)
 - Adresa, provozní doba, cena
3. Zavolat do pizzerie (Řešení 3)
 - Tel. číslo, vychladne
4. Deklarovat – že vyžaduji pizzu (status)
 - Prostředí se mi pokusí dodat pizzu
 - Nezávislé na okolí (dva zkrachují, jeden zabloudí, poslední dodá)
 - Bez starostí 😊
 - Inverze kontroly

Jak IoC v rámci Springu v praxi použít?

- Například:

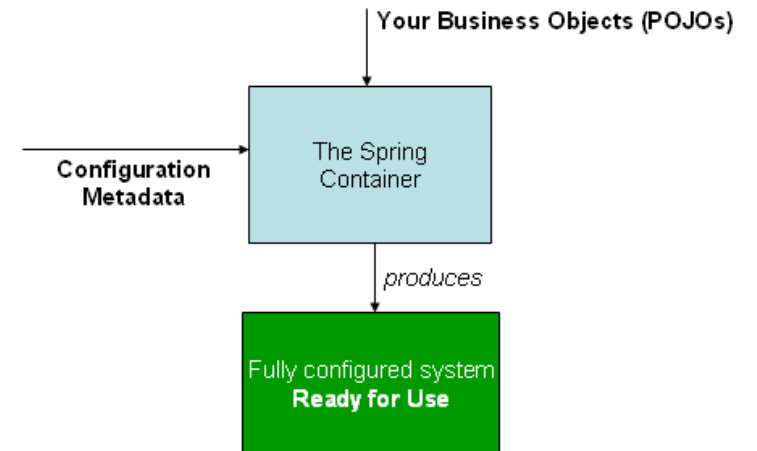
- Vytvořit projekt typu Maven
- Přidat závislost na kontejneru frameworku Spring

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.2.4.RELEASE</version>
  </dependency>
</dependencies>
```

- Provést konfiguraci kontejneru (pomocí XML nebo anotací nebo Java konfigurační třídy)
 - Označit v rámci konfigurace ty třídy, na kterých ostatní závisí, a které se mají automaticky injektovat, jako tzv. „beans“
- V programu:
 - Vytvořit instanci kontejneru respektive jeho kontextu
 - Objekty označené jako beans se získají z kontejneru např.
 - ručně metodou context.getBean....
 - automaticky pomocí DI v příslušném místě prog., kde je definována závislost
 - Rozdíl mezi kontejnerem a kontextem (součást kontejneru bude vysvětlen dále)

IoC kontejner Springu

- Definován v rámci balíků `org.springframework.beans` a `org.sp...rk.context`
- Existují různá rozhraní reprezentující Spring kontejner, typicky
 - `org.springframework.context.ApplicationContext`
- Úkolem tříd reprezentujících kontejner je
 - Vytvoření kontejneru
 - Konfiguraci kontejneru
 - Spojení komponent aplikace do jednoho celku
- Způsob vytvoření
 - Ručně pro standalone aplikace (desktopové, CLI)
 - Z Java nebo XML konfigurace:



```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("application.xml");
```

- Deklarací pro webové aplikace
 - Např. při konfiguraci kontextu Servletu – později v dalších přednáškách

Co je to Bean? ...fazole 😊

- Bean je objekt spravovaný frameworkem Spring
- Je vytvořený a řízený na základě konfigurace
 - Ta určuje všechny jeho další vlastnosti:
 - Třídu, ke které náleží
 - Jméno
 - Rozsah platnosti (Scope)
 - Způsob inicializace
 -
- Každý bean má také svůj životní cyklus
 - Lze např. definovat metodu, která bude zavolána po vytvoření nebo zničení beanu

http://www.tutorialspoint.com/spring/spring_bean_definition.htm

XML konfigurace a příklady

Příklad využití IoC: Injekce přes konstruktor

```
public class DynamicMessageOfTheDay implements MessageOfTheDayService ...
public class BasicMessageOfTheDay implements MessageOfTheDayService ...

public interface MessageOfTheDayService {
    public String getMessage();
}

public class MessagePrinterService {
    private final MessageOfTheDayService myService;
    public void printMessage () {
        System.out.println(myService.getMessage());
    }
    public MessagePrinterService(MessageOfTheDayService service) {
        this.myService = service;
    }
}
```

Příklad: Injekce přes konstruktor – třída Main

- Nejprve je vytvořen kontext z příslušného XML
- Poté je vytvořen objekt myMessagePrinter pomocí metody `getBean(MessagePrinterService.class)`

```
public class Main {  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("application.xml");  
        MessagePrinterService myMessagePrinter = context.getBean(MessagePrinterService.class);  
        myMessagePrinter.printMessage();  
    }  
}
```

- Pokud by v rámci XML existovalo více beanů této třídy, bylo by nutné vybrat jen jeden přes konkrétní ID:

```
MessagePrinterService myMessagePrinter = context.getBean("printer", MessagePrinterService.class);
```

Příklad: Injekce přes konstruktor- XML definice

- XML je validováno proti příslušnému XSD
 - Idea: podporuje napovídání, kontroluje správnost beans oproti kódu, vkládání nového prvku ALT+INS
- Obsahuje dva beany
 - Bean třídy MessagePrinterService má jako parametr konstruktoru nastaven odkaz (ref) na bean třídy BasicMessageOfTheDay

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="basicMessageOfTheDay" class="cz.tul.service.impl.BasicMessageOfTheDay"/>

    <bean id="printer" class="cz.tul.service.MessagePrinterService">
        <constructor-arg ref="basicMessageOfTheDay" ></constructor-arg>
    </bean>
</beans>
```

Schéma závislostí bez IoC

- Třída MessagePrinterService
 - Závisí na
 - Rozhraní MessageOfTheDayService
 - Implementaci BasicMessageOfTheDay
 - Vytváří instanci BasicMessageOfTheDay

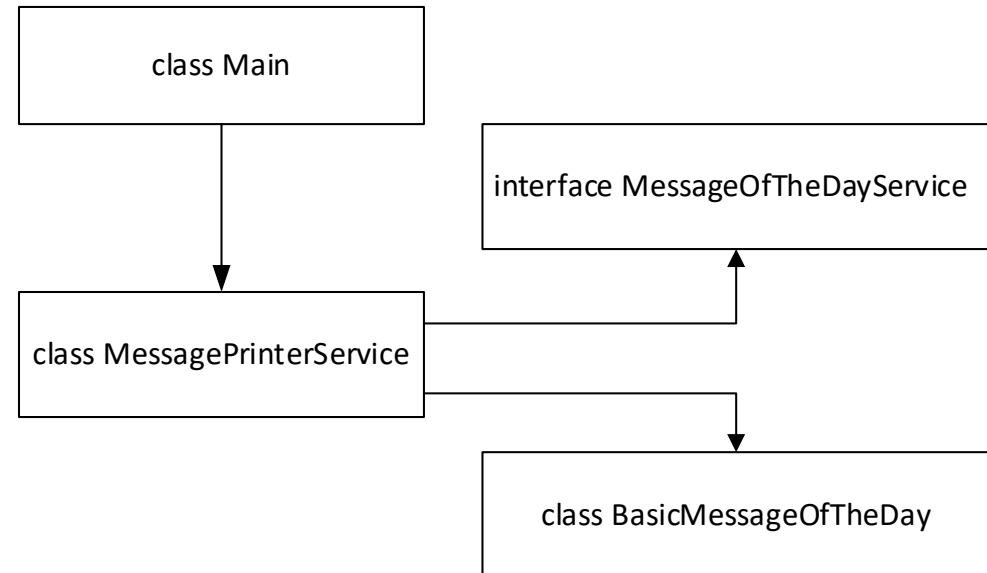
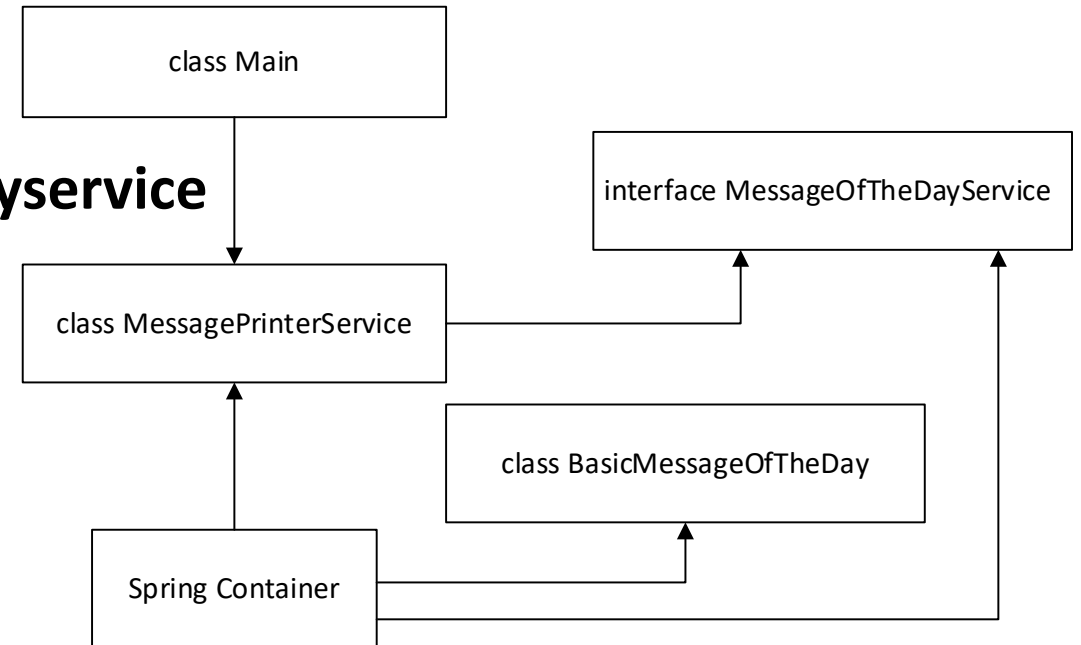


Schéma závislostí s IoC

- Třída MessagePrinterService
 - **Závisí pouze na rozhraní MessageOfDayService**
- Kontejner
 - Závisí na
 - Třídě MessagePrinterService
 - Rozhraní MessageOfDayService
 - Implementaci BasicMessageOfDay
 - Vytváří
 - Implementaci BasicMessageOfDay
 - **Dodává instanci třídy BasicMessageOfDay (= konkrétní implementaci rozhraní MessageOfDayService) třídě MessagePrinterService**



Příklad využití IoC: Injekce přes setter

```
public class DynamicMessageOfTheDay implements
MessageOfTheDayService ...
public class BasicMessageOfTheDay implements
MessageOfTheDayService ...

public interface MessageOfTheDayService {
    public String getMessage();
}

public class MessagePrinterService {
    private MessageOfTheDayService myService;
    public void printMessage () {
        System.out.println(myService.getMessage());
    }
    public void setMyService(MessageOfTheDayService service) {
        this.myService = service;
    }
}
```

Příklad: Injekce přes setter- XML definice a Main

- XML obsahuje opět dva beany
 - Závislost je nyní definovaná pomocí reference v rámci property beanu printer
 - Reference je opět na bean basicMessageOfTheDay
 - Jméno property (myService) je odvozeno podle konvence od jména setteru (setMyService)
 - setMyService -> myService
 - Bean basicMessageOfTheDay je vytvořen obdobně:
 - Pomocí setteru je nastavena property message na „Hello there“
- Třída Main je stejná jako při injekci přes konstruktor

```
<beans xmlns="http://www.springframework.org/schema/beans"
...
<bean id="basicMessageOfTheDay" class="cz.tul.service.impl.BasicMessageOfTheDay">
  <property name="message" value="Hello there"></property>
</bean>

<bean id="printer" class="cz.tul.service.MessagePrinterService">
  <property name="myService" ref="basicMessageOfTheDay"></property>
</bean>
</beans>
```


Další příklady injekce a definice beanů přes XML

- Injekce přes konstruktor s více parametry:

```
public class MessagePrinterService {  
    private final MessageOfTheDayService myService;  
    private final String myText;  
  
    public void printMessage () {  
        System.out.println(myService.getMessage());  
    }  
    public MessagePrinterService(String text, MessageOfTheDayService service) {  
        this.myText=text;  
        this.myService = service;  
    }  
}
```

```
<bean id="basicMessageOfTheDay" class="cz.tul.service.impl.BasicMessageOfTheDay"/>
```

```
<bean id="printer" class="cz.tul.service.MessagePrinterService">  
    <constructor-arg index="0" value="My text" ></constructor-arg>  
    <constructor-arg index="1" ref="basicMessageOfTheDay"></constructor-arg>  
</bean>
```

Další příklady injekce a definice beanů přes XML

- Injektovat lze i další datové typy i pole a kolekce (list, map, ..)
 - Příklad: pole hodnot přes setter:

```
public class DynamicMessageOfTheDay implements MessageOfTheDayService {
    private String[] messages;
    public void setMessages(String[] messages) {
        this.messages = (String[]) messages.clone();
    }

    public String getMessage() {
        return messages[GregorianCalendar.getInstance().get(Calendar.DAY_OF_WEEK) - 1];
    }
}

<bean id="dynamicMessageOfTheDay" class="cz.tul.service.impl.DynamicMessageOfTheDay">
    <property name="messages" >
        <array>
            <value>"Sunday message"</value>
            <value>"Monday message"</value>
        </array>
    </property>
</bean>

<bean id="printer" class="cz.tul.service.MessagePrinterService">
    <constructor-arg index="0" value="My text" ></constructor-arg>
    <constructor-arg index="1" ref="dynamicMessageOfTheDay" ></constructor-arg>
</bean>
```

Autowire v rámci XML #1

- Funkce Autowire umožňuje automaticky injektovat bean y
 - Defaultně je vypnutá
- Typy Autowire:
 - byName
 - Využívá injekci přes settery a hodí se, pokud existuje více beanů požadovaného typu
 - Umožňuje vybrat jen ten, jehož jméno odpovídá jménu proměnné, která se přes setter nastavuje

```
public class MessagePrinterService {  
  
    private MessageOfTheDayService myService;  
    public void printMessage () {  
        System.out.println(myService.getMessage());  
    }  
    public void setMyService(MessageOfTheDayService service) {  
        this.myService = service;  
    }  
}
```

```
<bean id="dynamicMessageOfTheDayMessageOfTheDay"  
      class="cz.tul.service.impl.DynamicMessageOfTheDay"  
      scope="singleton">  
</bean>
```

```
<bean id="myService"  
      class="cz.tul.service.impl.BasicMessageOfTheDay" scope="singleton">  
    <property name="message" value="Hello there"></property>  
</bean>
```

```
<bean id="printer" class="cz.tul.service.MessagePrinterService"  
      autowire="byName">  
</bean>
```

Autowire v rámci XML #2

- Typy Autowire:
 - byType
 - Využívá také injekci přes settery a hodí se, pokud existuje **jeden** bean požadovaného typu
 - constructor
 - Obdoba byType, ale využívá injekci přes konstruktor

Výhody a nevýhody XML konfigurace

- Nevýhody:
 - Obtížnější refaktoring a kontrola správnosti (vše zapsané pomocí řetězců)
 - vyžaduje podporu IDE (Idea umí)
 - Konfigurace není validováno při kompilaci
 - Často dochází k runtime chybám
 - Existují dvě místa, kde je problém popsán: kód + související XML
- Výhody:
 - XML je možné generovat (například pro různé zákazníky)
 - Načítat externě konfigurační parametry je ale možné i v případě ostatních způsobů konfigurace

Konfigurace pomocí anotací

Konfigurace pomocí anotací

- Novější způsob konfigurace než XML
 - Čím dál více populární
- Často flexibilnější než XML (např. v případě Autowire funkce)
- Lze kombinovat s XML konfigurací nebo s Java konfigurací (viz dále)
 - Kombinace s XML je problematická, kvůli tomu, že konfigurace je pak na dvou místech (v kódu a v XML)
 - Je např. kontroverzní definovat beany v XML a autowire definovat pomocí anotací v kódu
 - Naopak výhodná je kombinace s Java konfigurací, protože celý problém je pak pospán na jednou místě (pomocí funkcí a tříd) a jejich anotací a součástí kódu jsou i konfigurační třídy

Autowire přes anotace – příklad

- XML obsahuje definice dvou beanů:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

    ...

    <context:annotation-config></context:annotation-config>

    <bean id="basicMessageOfTheDay" class="cz.tul.service.impl.BasicMessageOfTheDay"/>

    <bean id="printer" class="cz.tul.service.MessagePrinterService"/>

</beans>
```


Autowire přes anotace a settery

- Ve třída MessagePrinterservice se anotují příslušné settery:
 - Anotace `@Autowired` se defaultně chová jako „byType“
 - Hledá bean příslušného typu (zde v XML konfiguraci) a použije reference na něj
 - Anotace `@Value` zajistí injekci požadované hodnoty

```
public class MessagePrinterService {  
  
    private MessageOfTheDayService service;  
    private String myText;  
  
    @Autowired  
    public void setMessageOfTheDayService(MessageOfTheDayService service) {  
        this.service = service;  
    }  
    @Value("New text")  
    public void setMyText(String text) {  
        this.myText = text;  
    }  
    public void printMessage () {System.out.println(service.getMessage()); }  
}
```

Autowire přes anotace a konstruktor

- Ve třídě MessagePrinterservice se anotuje konstruktor:
 - Anotace `@Autowired` se tentokrát přidá na konstruktor
 - Opět hledá bean příslušného typu (zde v XML konfiguraci) a použije referenci na něj
 - Anotace `@Value` se přidá k parametru konstruktoru

```
public class MessagePrinterService {  
  
    private final MessageOfTheDayService myService;  
    private final String myText;  
  
    @Autowired  
    public MessagePrinterService(@Value("My Text") String text, MessageOfTheDayService service) {  
        this.myText=text;  
        this.myService = service;  
    }  
    public void printMessage () {  
        System.out.println(myService.getMessage());  
    }  
}
```

Autowire přes anotace a atributy

- Ve třída MessagePrinterservice se anotují přímo atributy:
 - Není třeba konstruktor s parametry ani settery!
 - Spring využívá reflexy – pokročilá vlastnost jazyka Java, umožňuje za běhu získávat informace o třídách a jejich proměnných a metodách, aniž by bylo nutné znát tyto informace v době kompilace.
 - Pokud se neuvede @Autowired(required=false) je vyžadována existence příslušného beanu v kontejneru
- Často používaná varianta

```
public class MessagePrinterService {  
  
    @Autowired  
    private MessageOfTheDayService myService;  
    @Value("My Text")  
    private String myText;  
  
    public void printMessage () {  
        System.out.println(myService.getMessage());  
    }  
}
```

Autowire přes tributy – kvalifikace beanů

- Spring skenuje kontejner a hledá beany, jejichž ID = jméno atributu
 - Pokud odpovídající bean nenajde, hledá bean podle typu
- Pokud chceme tuto strategii změnit, lze vybrat konkrétní bean přes jeho ID pomocí anotace `@Qualifier`:

```
public class MessagePrinterService {  
  
    @Autowired  
    @Qualifier("requested_ID")  
    private MessageOfTheDayService myService;  
  
    @Value("My Text")  
    private String myText;  
  
    public void printMessage () {  
        System.out.println(myService.getMessage());  
    }  
}
```

Anotace @Component

- V předchozím příkladu byly beany definovány v rámci XML
 - Pomocí anotací se pak injektovaly do příslušných tříd
 - Konfigurace byla rozdělena do XML a kódu ☹
- Pokud chceme definici beanů z XML vyjmout, je nutné je definovat v kódu
- Pro tento účel slouží anotace @Component
 - Spring při inicializaci takto anotované třídy vyhledává a automaticky z nich v případě potřeby vytváří beany
 - V rámci XML konfigurace je nutné nastavit, v jakém balíku budou třídy vyhledávány
 - Je třeba je přidat na ClassPath:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"

    ...
    <context:component-scan base-package="cz.tul.service"></context:component-scan>

</beans>
```

Příklad použití @Component

```
@Component("basicMessageOfTheDay")
public class BasicMessageOfTheDay implements MessageOfTheDayService {...}

@Component("dynamicMessageOfTheDay")
public class DynamicMessageOfTheDay implements MessageOfTheDayService {...}

@Component
public class MessagePrinterService {

    @Autowired
    @Qualifier("dynamicMessageOfTheDay")
    private MessageOfTheDayService myService;

    @Value("My Text")
    private String myText;

    public void printMessage () {System.out.println(myService.getMessage());}

}

public static void main(String[] args){
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("application.xml");
    MessagePrinterService myMessagePrinter = context.getBean(MessagePrinterService.class);
    myMessagePrinter.printMessage();
}
```

Anotace typu stereotype

- Kromě anotace `@Component` existují i další podobné, které dohromady tvoří množinu tzv. **stereotypů**
 - Mají společnou vlastnost, že pokud je zapnuto autoskenování, Spring z takto anotovaných tříd v případě potřeby vytváří bean
 - Slovo stereotype značí, že tyto anotace **definují roli příslušných tříd** v rámci Springu (mimo jiné jsou to kandidáti na vytvoření beanů)
- Další anotace typu stereotype:
 - `@Service`
 - Pouze vhodnější název anotace než `@Component` pro třídy, které se chovají jako služba.
 - `@Controller`
 - Takto anotovaná třída je navíc v rámci modulu Spring Web MVC chápána jako kontroler
 - Na její metody lze pomocí anotace `@RequestMapping` směřovat příchozí HTTP požadavky
 - `@Repository`
 - Využívá se v rámci modulu Spring Data, poskytuje navíc překlad výjimek

Java konfigurace

Java konfigurace

- V přechozím příkladech se postupně nahrazovalo XML anotacemi
 - Část XML ale stále přetrvávala
 - Konfigurace tak byla na dvou místech (v anotacích v kódu a v XML)
- Java konfigurace představuje způsob, jak XML eliminovat
 - Nejnovější způsob konfigurace
 - Čím dál více populární
 - Umožňuje detekci chyb již v rámci kompilace
 - Založena na využití anotací a konfiguračních tříd

Java konfigurace – konfigurační třída – příklad #1

- Jako konfigurační třída může sloužit kterákoli třída
 - Je pouze nutné ji anotovat pomocí `@Configuration`
 - Musí mít definovaný prázdný konstruktor
 - Pokud není definovaný žádný konstruktor, Java kompilér prázdný vytvoří defaultně
- Kontext Springu se pak nevytváří z XML, ale z této třídy:
(Anotace `@ComponentScan` nad konfigurační třídou v příkladu nahrazuje XML tag `<context:component-scan base-package="cz.tul.service"></context:component-scan>`)

```
@Configuration
@ComponentScan("cz.tul.service")
public class AppConfig {
}

public static void main(String[] args){
    AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
    MessagePrinterService myMessagePrinter = context.getBean(MessagePrinterService.class);
    myMessagePrinter.printMessage();
}
```

Java konfigurace – konfigurační třída – příklad #2

- Zbytek programu může být beze změny s využitím `@Component` a `@Autowired`

```
@Component("basicMessageOfTheDay")
public class BasicMessageOfTheDay implements MessageOfTheDayService {...}

@Component("dynamicMessageOfTheDay")
public class DynamicMessageOfTheDay implements MessageOfTheDayService {...}

@Component
public class MessagePrinterService {

    @Autowired
    @Qualifier("dynamicMessageOfTheDay")
    private MessageOfTheDayService myService;

    @Value("My Text")
    private String myText;

    public void printMessage () {System.out.println(myService.getMessage());}

}
```

Konfigurační třídy – importování

- Java konfigurace se může skládat z více konfiguračních tříd
 - Konfigurační třída, z které se vytváří kontext, pak musí zvolené třídy importovat pomocí anotace `@Import({class1.class,...,classN.class})`:

```
@Configuration
@ComponentScan("cz.tul.service")
@Import(AppConfig1.class)
public class AppConfig {

}
```

- Lze importovat i konfiguraci z XML pomocí anotace `@ImportResource`

Konfigurační třídy – definice beanů

- Beany lze definovat i v rámci konfigurační třídy
 - Pomocí anotace `@Bean` nad metodou, která vrací instanci dané třídy
- Odpadá použití anotace `@Component`, `@Autowired` u jednotlivých tříd
 - **Třídy programu nejsou závislé na Springu**
 - Pouze konfigurační třída ano
 - **Konfigurace není rozeseta v kódu v rámci definice jednotlivých tříd!**
 - **Na rozdíl od XML konfigurace je už během kompilace detekováno, zda v rámci konfigurace existuje vhodný bean či zda naopak není beanů více (nejednoznačnost)!**
- Dle dané závislosti je nutné provést provázání jednotlivých Beanů
 - Ideálně přes konstruktor

Konfigurační třídy – definice beanů – příklad

- Provázání beanů v rámci konfigurační třídy:
 - Metoda `dynamicMessageOfTheDay()` definující bean třídy `DynamicMessageOfTheDay` je předána jako parametr do konstruktoru třídy `MessagePrinterService`
 - Tento konstruktor je volán v rámci metody `messagePrinterService`, která vrací instanci třídy `MessagePrinterService`

```
@Configuration
public class ServiceConfig {
    @Bean
    public DynamicMessageOfTheDay dynamicMessageOfTheDay() {
        return new DynamicMessageOfTheDay();
    }
    @Bean
    public MessagePrinterService messagePrinterService () {
        return new MessagePrinterService("New text",dynamicMessageOfTheDay());
    }
}
```

Konfigurační třídy – Autowire

- V rámci konfiguračních tříd je možné používat funkci Autowire
 - A to i pro beany odpovídající konfiguračním třídám
- Při použití `@Autowire` je možné použít anotaci `@Qualifier` pro vybírání si z více beanů daného typu:
 - V příkladu níže je toto využito pro výběr konkrétní implementace rozhraní `MessageOfTheDayService`

```
@Configuration
@Import(ServiceConfig.class)
public class AppConfig {
}

@Configuration
public class BasicConfig {
    @Bean
    public BasicMessageOfTheDay basicMessageOfTheDay () {
        return new BasicMessageOfTheDay();
    }
}

@Configuration
public class DynamicConfig {
    @Bean
    public DynamicMessageOfTheDay dynamicMessageOfTheDay () {
        return new DynamicMessageOfTheDay();
    }
}
```

```
@Configuration
@Import({DynamicConfig.class, BasicConfig.class})
public class ServiceConfig {

    @Autowired
    @Qualifier("basicMessageOfTheDay")
    public MessageOfTheDayService service;

    @Bean
    public MessagePrinterService messagePrinterService () {
        return new MessagePrinterService("New text", service);
    }
}
```

Další vlastnosti beanů a
možnosti práce s nimi

Bean Scope (rozsah platnosti) #1

- Singleton (defaultní rozsah platnosti)
 - Daný bean existuje v kontejneru pouze jeden
 - Hodí se např. při vytváření objektů reprezentujících připojení k databázi
 - Vícenásobné zavolání metody `getBean` vrátí vždy stejný objekt

```
<bean id="basicMessageOfTheDay" class="cz.tul.service.impl.BasicMessageOfTheDay" scope="singleton">
  <property name="message" value="Hello there"></property>
</bean>
```

```
public class BasicMessageOfTheDay implements MessageOfTheDayService {
    private String myMessage = "Hello";
    public String getMessage() {return myMessage;}
    public void setMessage(String message) { this.myMessage=message; }
}
```

```
public static void main(String[] args){

    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("application.xml");

    BasicMessageOfTheDay concreteService = context.getBean(BasicMessageOfTheDay.class);
    concreteService.setMessage("New message");
    System.out.println(concreteService.getMessage());

    BasicMessageOfTheDay concreteService1 = context.getBean(BasicMessageOfTheDay.class);
    System.out.println(concreteService1.getMessage());

}
```

Bean Scopes #2

- **Prototype**
 - Bean existuje v kontejneru vícekrát
 - Zavolání metody `getBean` vrátí vždy nový objekt
- **Request, session, global-session**
 - Definovány pouze v rámci kontejnerů (kontextů) webových aplikací
 - Omezují rozsah platnosti na danou HTTP žádost či session

Bean Lifecycle (životní cyklus) #1

- Spring vkládá a odstraňuje beany z kontejneru v rámci posloupnosti řady operací
 - Na tyto operace lze navěsit metody, které budou zavolány, pokud bude daná operace životního cyklu provedena
 - Základní operace jsou Init() a Destroy():
 - V rámci XML lze:
 - Definovat metody init() a destroy() pro jednotlivé beany
 - Definovat defaultní metody init() a destroy() pro všechny beany v XML společně
 - V rámci definice tříd lze využít např. interface InitializingBean a/nebo DisposableBean
 - Daná třída může tyto interface implementovat
 - Pak je nutné implementovat příslušné call-back metody afterPropertiesSet() a/nebo destroy()
 - Lze využít anotace @PostConstruct a @PreDestroy nad vybranými metodami třídy

Bean Lifecycle (životní cyklus) #2

- Pokud je použito více způsobů najednou, platí pořadí:
 - Metoda anotovaná `@PostConstruct`
 - Metoda `afterPropertiesSet()` z `InitializingBean` interface
 - Metoda `init()`
- Obdobné pořadí platí i pro operaci `Destroy`
- Existují i další interface, jejichž implementací je možné získat callback metody na další operace životního cyklu

JSR anotace

- JSRs jsou Java Specification Requests (požadavky na změny v jazyce Java).
 - Jsou shromažďovány a vyřizovány v rámci pevně definovaného procesu (<https://jcp.org/en/procedures/overview>)
- Některé anotace, které Spring podporuje, vychází z některého JSR
 - JSR-250: `@PostConstruct` a `@PreDestroy` (a také `@Resource` = obdoba `@Autowire`)
 - JST-330: `@Inject`, `@Named`
 - Obdoba anotací `@Autowire` a `@Qualifier`
- Použití JSR anotací má tu výhodu, že kód je pak použitelný i pro jiný framework implementující DI, než Spring (např. Google Juice)
 - Podpora těchto anotací je součástí Java EE – knihovna Javax (java extensions)

Dědičnost Beanů

- Má smysl hlavně v případě XML konfigurace
 - Umožňuje dědičnost konfigurace v tom smyslu, že konfigurace jednoho beanu může sloužit jako šablona pro konfiguraci druhého
- Rozlišují se beany
 - Od kterých se konfigurace dědí – Parent
 - Které dědí konfiguraci od své rodiče – Children
- Zděděnou konfiguraci lze u odvozených beanů také přepsat popřípadě rozšířit
- Bean může být označen také jako abstraktní
 - Nelze vytvořit jeho instanci

Dědičnost Beanů- příklad

```
<?xml version="1.0" encoding="UTF-8"?>
...
<bean id="beanTemplate" abstract="true">
    <property name="message1" value="Hello World!"/>
    <property name="message2" value="Hello Second World!"/>
    <property name="message3" value="Namaste India!"/> </bean>

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia"
parent="beanTemplate"> <property name="message1" value="Hello
India!"/>
    <property name="message3" value="Namaste India!"/> </bean>
```

Způsob inicializace beanu

- Eagerly (včasná, okamžitá)
 - Všechny beany jsou na základě konfigurace vytvořeny ihned během inicializace kontejneru
 - Využívá se v rámci implementací Spring kontejneru typu ApplicationContext
 - Výhody
 - Chyby se mohou projevit hned při startu aplikace, ne až po několika hodinách běhu
 - Nevýhody
 - Start může být pomalejší a paměťové náročnější
- Lazy (líná)
 - Beany se vytvoří, až když jsou skutečně potřeba (např. zavolání metody `getBean`)
 - Využívá se v rámci implementací Spring kontejneru typu BeanFactory
- V rámci kontejneru typu ApplicationContext lze definovat i línou inicializaci:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
```


Interface BeanPostProcessor #1

- Tento interface definuje callback metody, jejichž implementací lze definovat způsob, jak kontejner Springu např.:
 - Vytváří instance beanů
 - Konfiguruje beany
 - Inicializuje beany
 - Zpracovává závislosti, atd.
- Na rozdíl od interface pro správu životního cyklu ovlivňuje tento interface chování všech beanů, ne jen jednoho daného.

Interface BeanPostProcessor #2

- Aby postprocesing fungoval na všechny beany, platí, že
 - V rámci kontejneru je jako první vytvořen bean reprezentující třídu, která implementuje interface BeanPostProcessor definující call-back metody
 - `postProcessBeforeInitialization()`
 - `postProcessAfterInitialization()`
 - Call-back metody
 - Mají jako argument proměnnou typu `Object` reprezentující jméno vstupního beanu a `String` reprezentující ID beanu
 - Vrací proměnnou typu `Object` reprezentující modifikovaný bean
 - Jsou součástí životního cyklu beanu a platí při tom následující pořadí volání:
 - `BeanPostProcessor.postProcessBeforeInitialization()`
 - `init()`
 - `BeanPostProcessor.postProcessAfterInitialization()`
- Lze také vytvořit více tříd implementujících BeanPostProcessor a pak určit pořadí, v kterém budou volány.

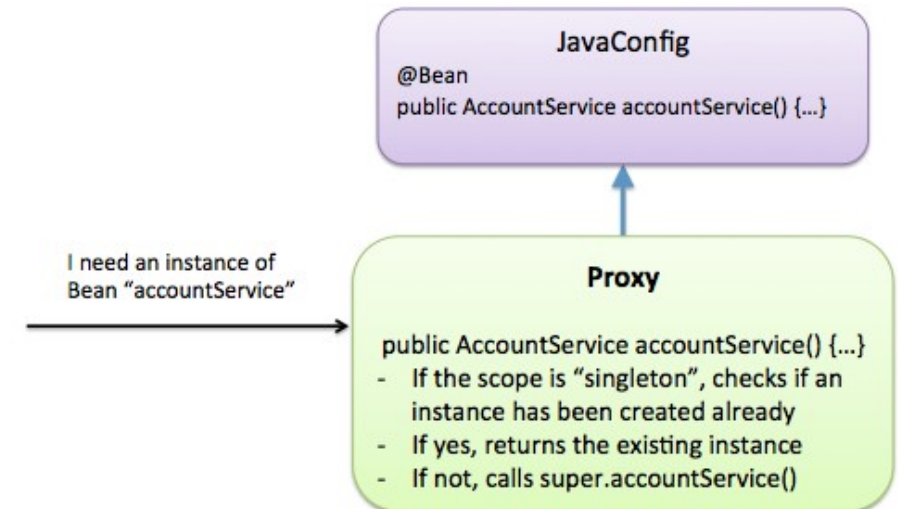
Další vlastnosti kontejneru Spring

Spring proxy #1

- V některých situacích vytváří Spring tzv. proxy objekty (prostředníky)
- Příklad: Java konfigurace a beanu typu singleton
 - Bean je
 - Reprezentovaný metodou vracející vždy novou instanci třídy
 - Typu singleton
 - Využíván jako parametr v konstruktorech dalších beanů
 - ⇒ Daná metoda pro vytvoření se volá z více míst
- Jak tuto situaci efektivně vyřešit?

Spring proxy #2

- Vytvořením proxy objektu (prostředníka)
 - Proxy vrátí novou instanci jen v případě prvního volání – viz schéma
 - Funguje principiálně dobře, pokud proxy reprezentuje rozhraní
 - Proxy objekt může mít dodatečné metody a atributy nad rámec požadavků rozhraní
 - Problematické v případě, že injektujeme konkrétní implementaci
 - Objekt je definován pevně, jak vytvořit jeho přesnou bitovou kopii s metodami navíc?
- => Další důvod, proč injektovat konkrétní implementace není vhodné!



Spring proxy #3

- Další využití proxy ve Springu:
 - Caching
 - Transakce
 - Aspect oriented programming (AOP)
- <https://spring.io/blog/2012/05/23/transactions-caching-and-aop-understanding-proxy-usage-in-spring>

Spring events (události)

- Jádro Springu tvoří kontejner (kontext), kteří spravuje beany
- Správa a načítání beanů je doprovázena celou řadou událostí
 - I kontejner má svůj životní cyklus (podobně jako bean)
- Tyto události lze obsluhovat (event handling) pomocí třídy `ApplicationEvent` a pomocí rozhraní `ApplicationListener`
 - Pokud nějaká třída reprezentovaná beanem implementuje toto rozhraní, je tento bean obeznámen s každou událostí typu `ApplicationEvent`, která nastane v rámci kontejneru

Spring events – příklady událostí

- ContextRefreshedEvent
 - Inicializace nebo obnovení kontextu
- ContextStartedEvent
 - Spuštění kontextu
- ContextStoppedEvent
 - Zastavení kontextu
- ContextClosedEvent
 - Ukončení (vypnutí) kontextu

Spring events – interface ApplicationListener

- ApplicationListener je rozhraní s generickým typem
 - Je parametrizováno přes typ respektive jeho parametrem je typ
 - Tento typ rozšiřuje třídu ApplicationEvent:

Interface ApplicationListener<E extends ApplicationEvent>

- Parametrem rozhraní ApplicationListener je tedy nějaká konkrétní podtřída třídy ApplicationEvent
- V rámci implementace rozhraní Application Listener je třeba definovat metodu onApplicationEvent(E)

Spring events – ApplicationListener – příklady

- ApplicationListener pro událost ContextStartedEvent (+ definovat bean):

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStartedEvent;

public class CStartEventHandler implements ApplicationListener<ContextStartedEvent> {
    public void onApplicationEvent(ContextStartedEvent event) {
        System.out.println("ContextStartedEvent Received");
    }
}
```

- ApplicationListener pro událost ContextStopEvent (+ definovat bean):

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStopEvent;

public class CStopEventHandler implements ApplicationListener<ContextStopEvent> {
    public void onApplicationEvent(ContextStopEvent event) {
        System.out.println("ContextStopEvent Received");
    }
}
```

Spring events – vlastní událostí (custom events)

- Rozšířením třídy `ApplicationEvent` lze definovat vlastní události, které je pak možné obsluhovat stejně jako standardní události Springu
- Před tím je ale ještě nutné zajistit, aby kontejner tyto události také oznamoval (stejně jako oznamuje ostatní události)
 - Je třeba vytvořit bean implementující rozhraní `ApplicationEventPublisherAware` a definovat některé jeho metody.....
- Ve výsledku je takto možné zajistit obsluhování jakékoli události některé vlastní třídy:
 - Událostí přitom může být vykonání nějaké konkrétní metody této třídy!

Testování

Jednotkové testy

Framework JUnit

Jednotkové testy – Framework JUnit

- Cílem je otestovat jednu funkci – konkrétní metodu
 - Jde o test nejnižší úrovně
 - Vyšší úroveň tvoří testy sestavení aplikace, integrační testy, ... ,akceptační testy (u zákazníka)
- Framework Junit
 - Knihovna pro jednotkové testy v jazyce Java
 - Patří do skupiny testovacích frameworků xUnit
 - Existují tedy obdobné nástroje i pro jiné platformy....
 - Základní principy
 - Jednotlivé testy se deklarují pomocí anotace @Test jako metody třídy
 - V rámci každého testu (metody) se využívá funkce Assert vyhodnocující danou podmínku
 - Třídy obsahující testy se mohou slučovat do souborů testů (test suites)

JUnit – Assertions (tvrzení)

- Junit poskytuje metody Assert pro všechny primitivní datové typy a objekty a pole primitivních datových typů a objektů
- Funkce Assert mají 3 parametry
 - První je řetěz vypsáný při neúspěchu (volitelný parametr)
 - Druhý je očekávaná hodnota
 - Třetí je skutečná hodnota
- Existují také funkce AssertThat - mají také 3 parametry:
 - Prvním je řetěz vypsáný při neúspěchu (volitelný parametr)
 - Druhý je skutečná hodnota
 - Třetí je objekt provádějící porovnání

JUnit – příklady funkce Assert

```
public class AssertTests {
    @Test
    public void testAssertArrayEquals() {
        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        assertEquals("failure - byte arrays not same", expected, actual);
    }
    @Test
    public void testAssertEquals() {
        assertEquals("failure - strings are not equal", "text", "text");
    }
    @Test
    public void testAssertNotNull() {
        assertNotNull("should not be null", new Object());
    }
    @Test
    public void testAssertNotSame() {
        assertNotSame("should not be same Object", new Object(), new Object());
    }
    @Test
    public void testAssertNull() {
        assertNull("should be null", null);
    }
    @Test
    public void testAssertSame() {
        Integer aNumber = Integer.valueOf(768);
        assertSame("should be same", aNumber, aNumber);
    }
}
```

JUnit – příklady funkce AssertThat

```
public class AssertThatTests {  
    @Test  
    public void testAssertThatHasItemsContainsString() {  
        assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));  
    }  
    @Test  
    public void testAssertThatEveryItemContainsString() {  
        assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(containsString("n")));  
    }  
}
```


Knihovny pro Assert

- AssertJ <http://joel-costigliola.github.io/assertj/>
- Google Truth <http://google.github.io/truth/>
- Lepší popis chyby
- (JUnit) `assertTrue(list.isEmpty)` ->
`java.lang.AssertionError at org.junit.Assert.fail(Assert.java:92)`
- `assertThat(list).isEmpty()` ->
`org.truth0.FailureStrategy$ThrowableAssertionError: Not true that is empty`

JUnit – Test suites

- Představují skupinu souvisejících testů
 - Chceme je spouštět dohromady
- Deklarují se pomocí anotací @Suite a @RunWith

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
//JUnit Suite Test

@RunWith(Suite.class)
@Suite.SuiteClasses({
    AssertTests.class ,AssertThatTests.class
})

public class JunitTestSuite {
}
```

JUnit – Spouštění testů #1

- Testy lze spustit pomocí metody `runClasses`:
 - Výsledek je uložen do objektu typu `Result`
 - Obsahuje jednotlivé chyby, s kterými lze dále pracovat

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args)
    {
        Result result =
JUnitCore.runClasses(JUnitTestSuite.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

JUnit – Spouštění testů #2

- IDE
 - Většina má integrovanou grafickou podporu pro spouštění testů
 - Daná třída se dá spustit jako test a následně je zobrazen výsledek
- Příkazový řádek
 - JUnitCore lze využít pro spuštění testu i z příkazové řádky:
 - Předpokládá, že modul JUnit i třída s testy (TestClass1) je uvedena na ClassPath

```
java org.junit.runner.JUnitCore TestClass1 [...other test classes...]
```

JUnit a Maven- závislosti

- Je třeba přidat závislost:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

- Příkaz „mvn test“ Implicitně pouští testy pojmenované *Test.java
 - Je ale možné spustit i jeden konkrétní test (může jít o suit zahrnující další testy)

```
mvn -Dtest=Tests.java test
```

- Nebo nakonfigurovat příslušný plugin a definovat masku:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.14.1</version>
      <configuration>
        <includes>
          <include>**/Tests.java</include>
        </includes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

JUnit a navázání na Spring

- Během testování je možné využívat Spring kontejner a jeho beany
 - Lze použít stejnou konfiguraci
- V praxi jsou ale během testování odlišné požadavky než během vývoje
 - Chceme například použít jiný typ databáze (In-memory)

=>

- Konfiguraci Springu lze parametrizovat přes externí soubory
- Lze také vytvářet profily – některé konfigurační třídy mohou být platné jen v některých profilech
- Parametrizace konfigurace a další věci řeší efektivně modul Spring Boot – viz další přednáška.

Odkazy a další zdroje informací

- Tutoriály a kurzy
 - Spring
 - Kurz pro začátečníky:
 - <https://www.udemy.com/springcore/learn/#/>
 - Kurz pro středně pokročilé
 - <https://www.udemy.com/springcore2/learn/#/>
 - <http://www.tutorialspoint.com/spring/index.htm>
 - JUnit
 - <http://www.tutorialspoint.com/junit/index.htm>
- Dokumentace
 - <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/>

Otázky ke zkoušce #1

- Co znamená pojem „svázání se s implementací“?
- Co je to interface a jaké jsou výhody programování do interface v jazyce Java?
- Co znamená pojem Inversion of Control a jaké má hlavní výhody?
- Co znamená pojem Dependency Injection a jaké má hlavní výhody?
- Jaké jsou výhody Springu pro vývoj enterprise aplikací?
- Co je to kontejner Springu, k čemu slouží a jak ho lze konfigurovat?
- Jaké jsou výhody a nevýhody jednotlivých možností konfigurace kontejneru?
- Jaký je rozdíl mezi dependency injection přes settery a přes konstruktor?
- Jak se provádí dependency injection v rámci XML konfigurace?
- Jak se provádí dependency injection pomocí anotací?
- Jak se provádí dependency injection v rámci Java konfigurace?
- K čemu je anotace @Autowire a jak se Autowire provádí v rámci různých typů konfigurace?
- K čemu je anotace @Qualifier?
- Co jsou to stereotypy?
- K čemu je anotace @ComponentScan?

Otázky ke zkoušce #2

- Co je to bean?
- Co je to bean scope a jaký bean scope je defaultní?
- Životní cyklus beanů a jejich využití.
- K čemu jsou anotace `@PostConstruct` and `@PreDestroy`?
- K čemu slouží `BeanPostprocessors`?
- Způsoby inicializace beanů.
- K čemu jsou ve Springu proxy?
- Spring events a jejich využití.