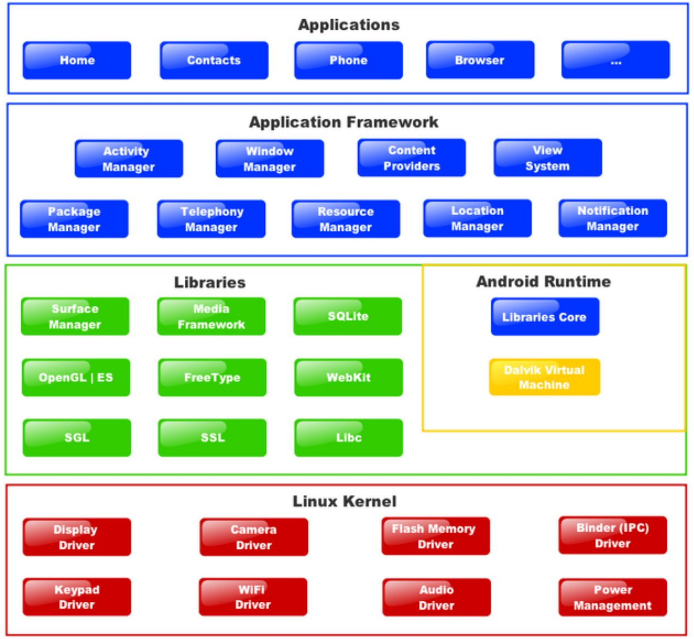
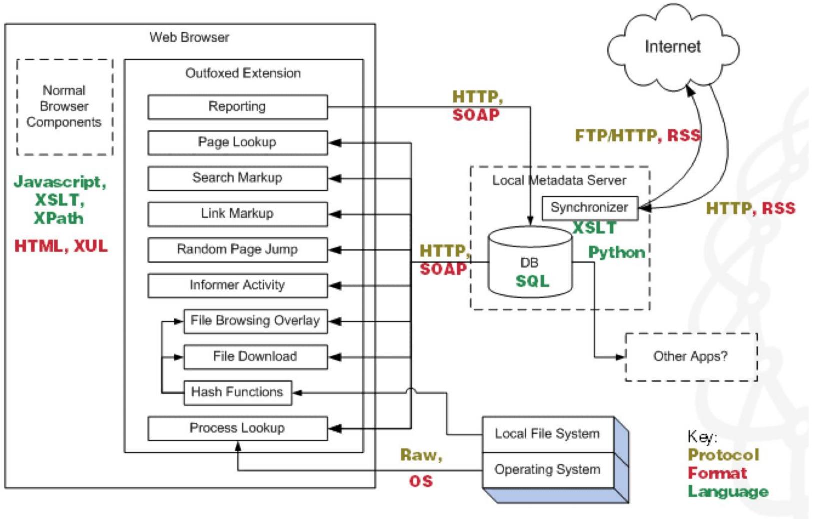
Introduction to Enterprise Application Architecture and Spring Framework

Charles Zhang

Fall 2019

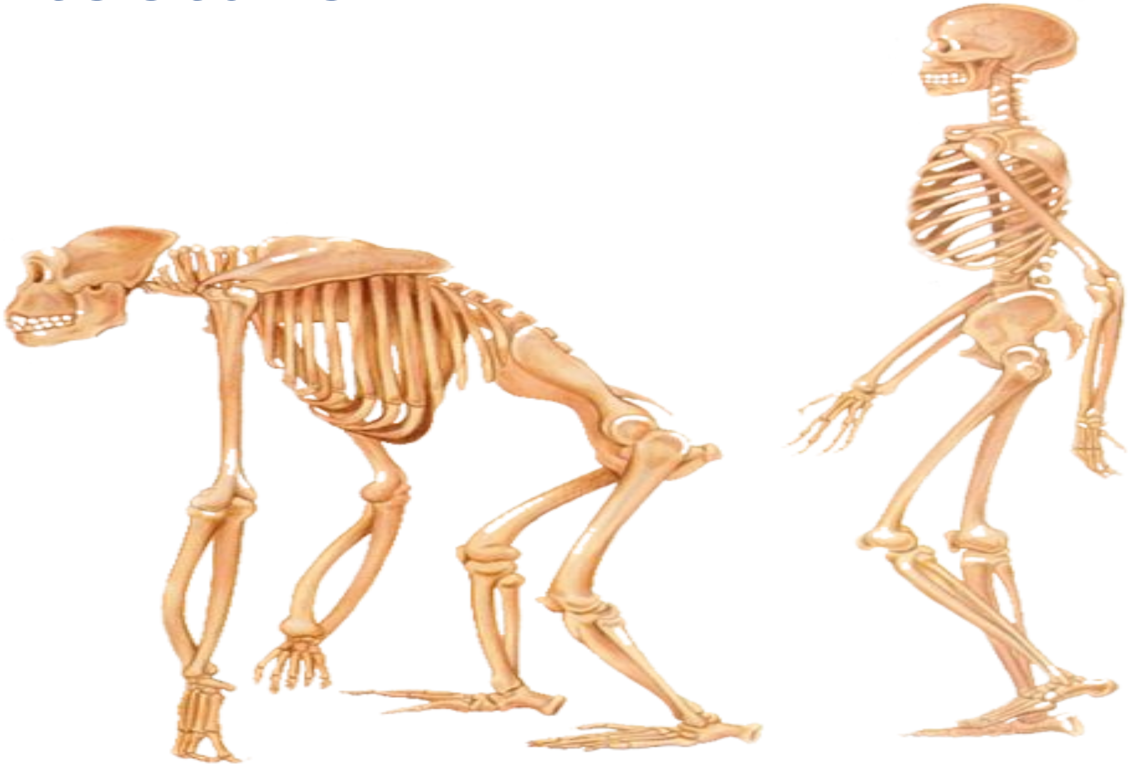
Software Architecture





- High level breakdown to the major components
- How they interact with each other

Do humans and apes have the same architecture?



Why is architecture important?

- Vehicle for communication
  - Bring developers, QA's, and stakeholders to the same page
- Manifests early design decisions
  - Affect technological choices
- Affects quality attributes
  - Predict the quality by studying the architecture
- Hard to change
  - High complexity and cost to change

Performance measurement and Scalability

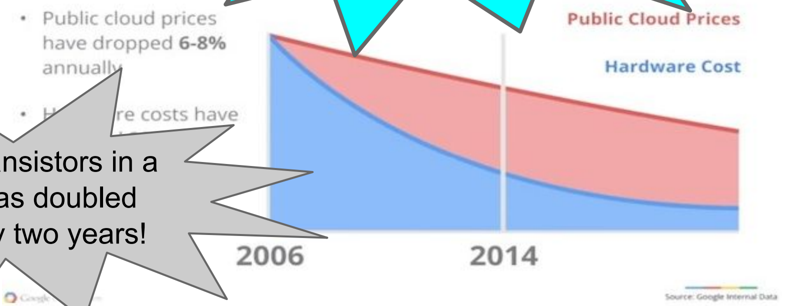
- Round trip time
- First-response time
  - Responsiveness
  - Direct impacts UX
- Load
  - Current stress
  - Measured by # of users, # of queries
- Throughput
  - QPS, TPS
- Capacity
  - Maximum throughput
- Scalability
  - How adding hardware improves performance
  - Vertical scalability vs horizontal scalability

Optimize for scalability!

Avoid software bloat!!!

- Capacity can be complex to improve for a given hardware configuration
- Buying new hardware may be cheaper than getting it run on existing older hardware
- Adding servers can be cheaper than adding programmers
  - Moore's law

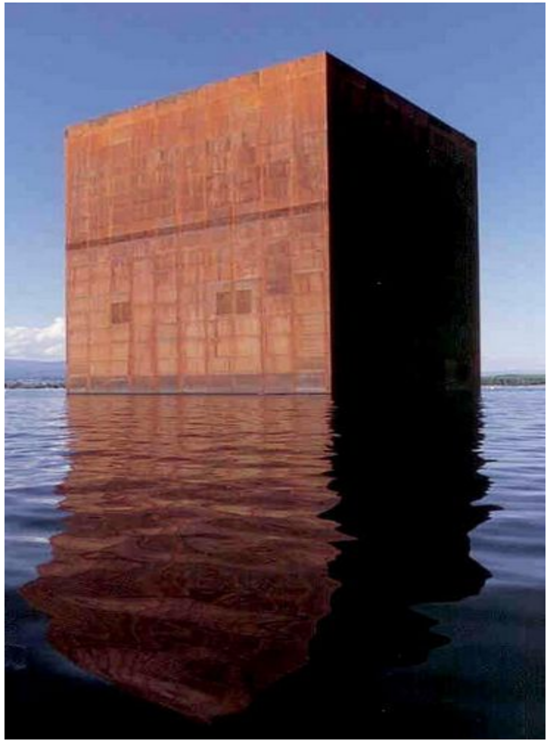
The # of transistors in a dense IC has doubled about every two years!



- Public cloud prices have dropped **6-8%** annually
- H re costs have

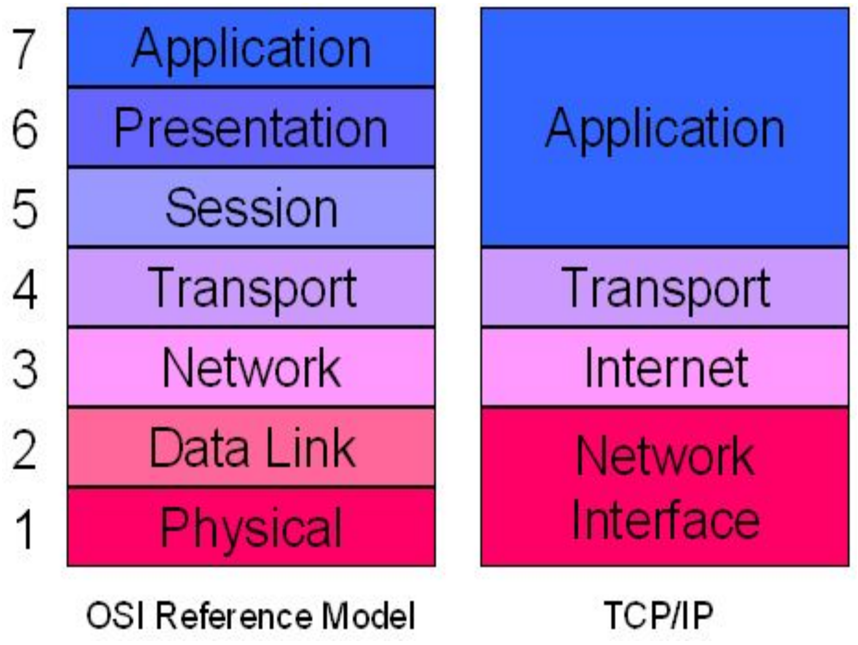
- Should we always count on more hardware then?
  - Wirth's law / Page's law / Bill's law: software is getting slower faster than hardware getting faster
  - Whatever Andrew gives, Bill takes it away

Monolithic Architecture


Source: http://odino.org/on-monoliths-service-oriented-architectures-and-microservices/

- Functionally distinguishable aspects (UI, business logic, and access control) are not architecturally separate components
- Pros and cons?

Layered software architecture: Break the system into *layers*



- Pros
  - Clarity: Easy to understand and communicate
  - Minimize dependencies
    - Easy to isolate and troubleshoot
  - Easy to substitute
- Cons
  - Cascading changes
  - May harm performance, if not done right

Three principal layers

- Presentation
  - Display of information, user interaction
  - GUI (Graphical), CLI (Command Line), VUI (Voice)
- Domain (AKA: Service, Business Logic)
  - Where the real computing is: validation, calculation, dispatching, etc
- Data source
  - Communication with database, messaging, and external services

Where to run the layers?

- Presentation
  - Desktop client (rich client)
  - Browser/HTML client
    - Part of the presentation can live on server
    - Is it always a thin?
  - Mobile client
- Business Logic
  - Client or server (pros and cons?)
- Data source
  - Almost always on the server

Service Oriented Architecture (SOA)

- Style of software design where services are provided to other components as applications, delivering the services through a communication protocol over a network
- A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently
    - Self-contained
    - Logically represents a business activity with a specified outcome
    - Black box for its consumers, but with a well defined interface
    - May consist of other underlying services

Ref: https://en.wikipedia.org/wiki/Service-oriented_architecture

Building blocks in SOA

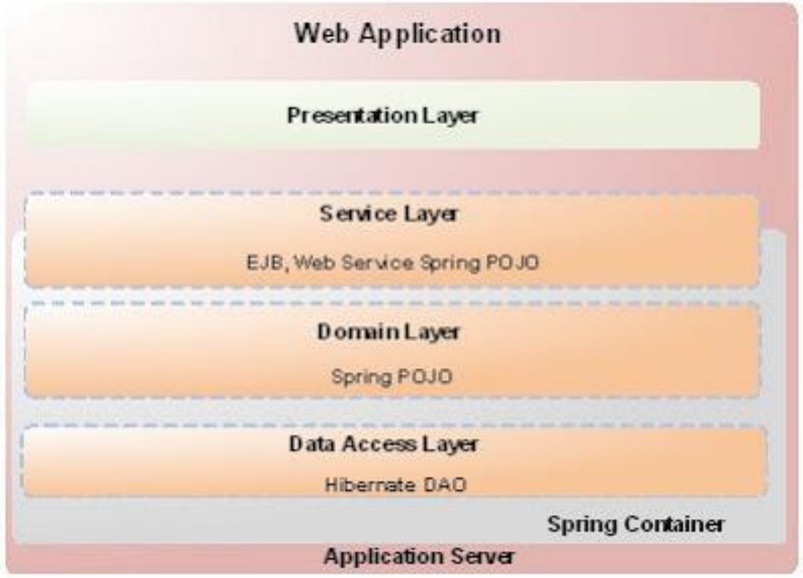
- Service providers
  - Standardized service contract
  - Reference autonomy
  - Location transparency
  - Longevity / high availability
  - Statelessness
- Service brokers, registries, repositories
- Service requesters/consumers

Pros and Cons of SOA

- Benefits
  - Promotes decoupling
  - Promotes fast and independent development
  - Higher level of sharing
  - Easier for isolation and testing at service level
- Criticism
  - Harder for integration tests
  - Hard to manage metadata consistency
  - Hard to manage version consistency

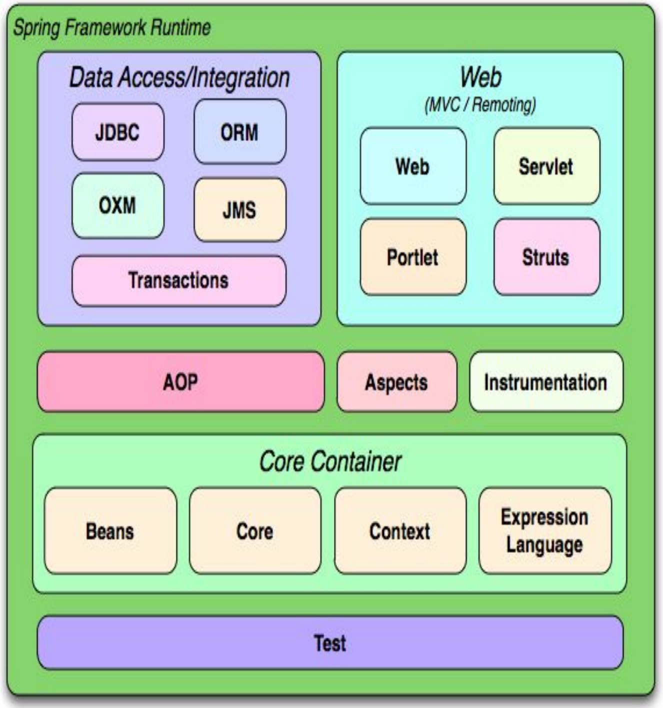
Spring: Popular framework to simplify Java application development

- Open source application framework
- Comprehensive and lightweight
- Alternative to, replacement for, or addition to EJB



What does Spring provide? Almost everything

- Inversion of Control (IoC) container
  - configuration of application components and lifecycle mgmt
- Aspect Oriented Programming (AOP)
- Data access
  - JDBC, ORM integration
- Transaction management
- Security
- Model-View-Controller (MVC)
  - Separate representation from presentation



Why frameworks like Spring?

- No-intrusive framework
  - Minimal changes to run with or without Spring (**POJO**s allowed)
  - Minimal lock-in: easy migration
- Promotes good programming practices
  - Program to interfaces, not implementations
  - Convention over configuration
- Lightweight, flexible,and allows pick-and-choose
- Does not reinvent the wheel
  - No NIH (Not-In-House) syndrome
  - Welcomes existing solutions and eases the integration

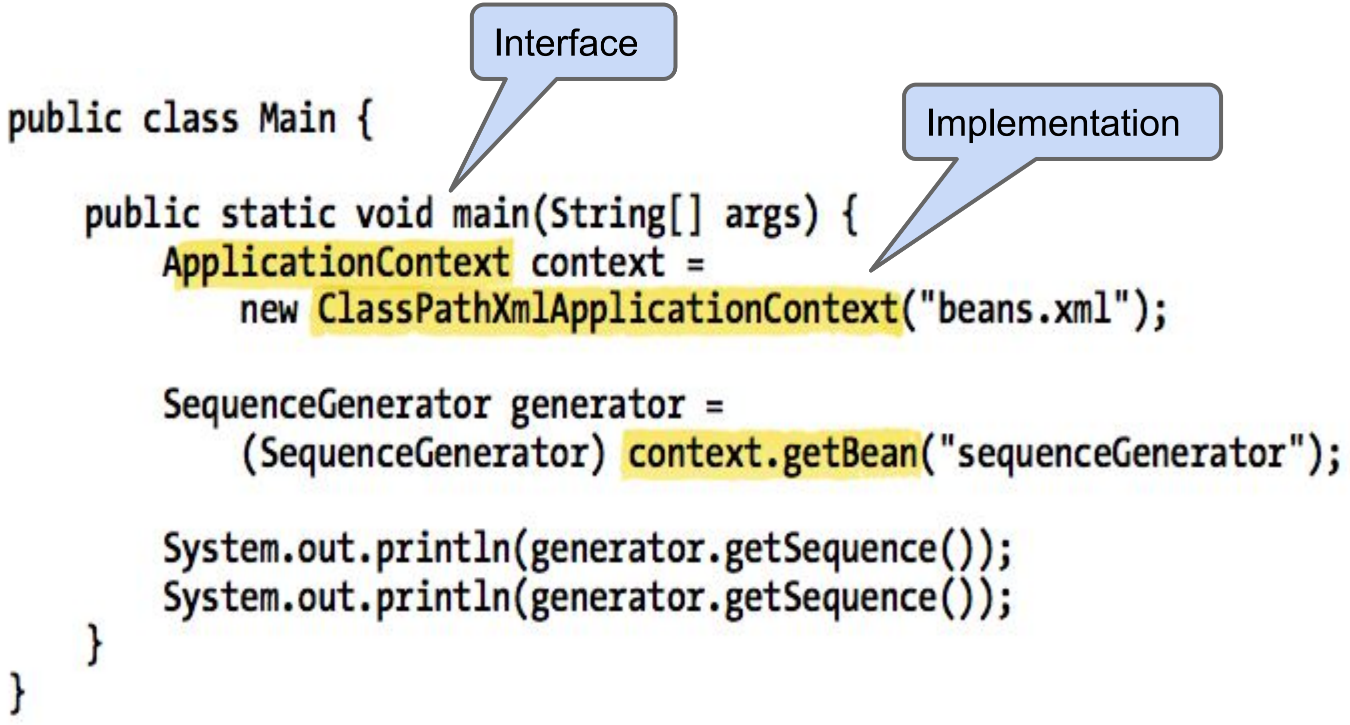
Inversion of Control: Don’t call us, we will call you

- What is IoC?
  - Traditional procedural programming: custom code calls reusable code (libraries)
  - IoC: reusable code calls custom code
    - Not really new: callbacks, event handlers
- Why IoC
  - Decouple execution flow from task implementation
  - Focus on module implementation based on contracts
  - Promotes module replaceability

Dependency Injection

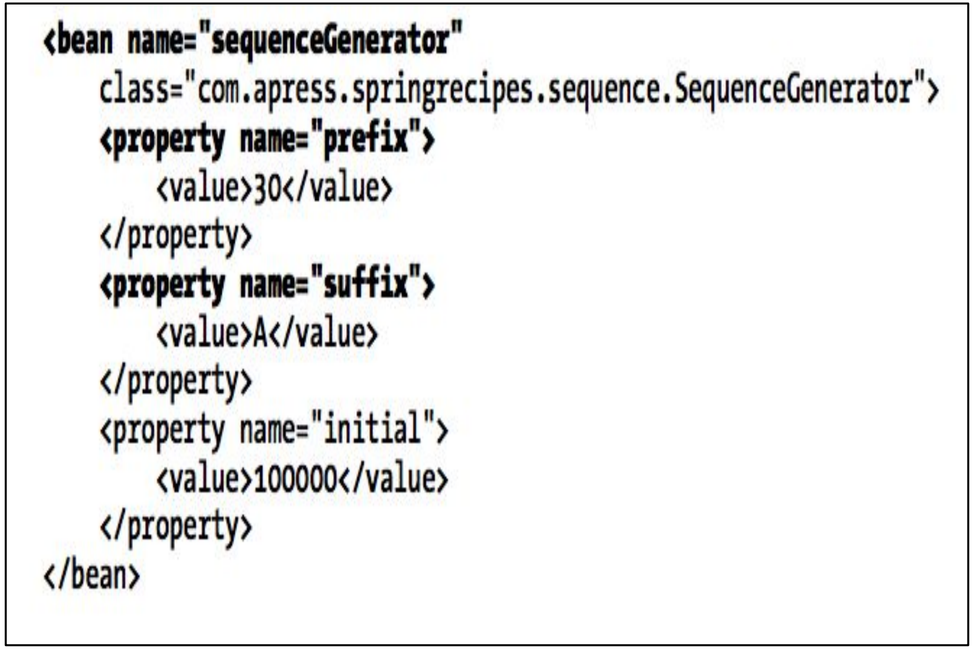
- Passing of a dependency (a service) to a dependent object (a client) - an implementation of IoC
- How to make it happen
  - Client object depending on the service
  - Interface the client uses to communicate with the service
  - Implementation of a service object
  - Injector object (aka injector, container, etc), responsible for injecting the service into the client
    - In the case of Spring, it’s Spring’s *application context*

Create application context and retrieve beans

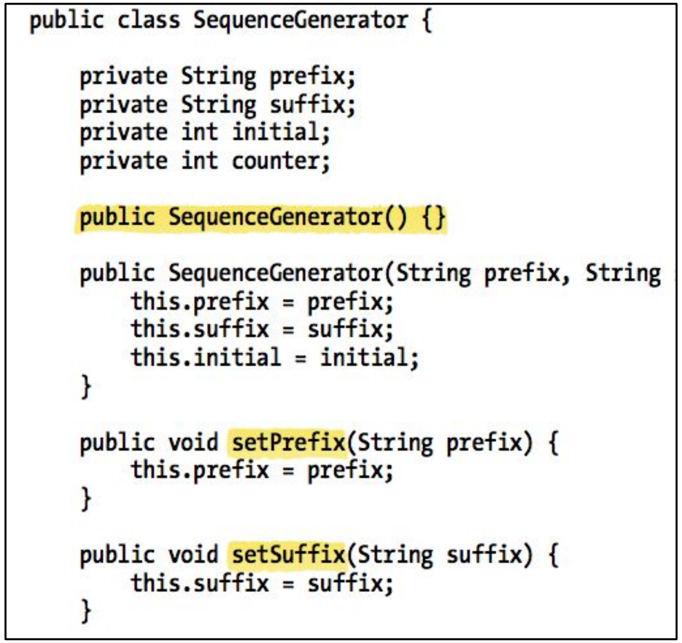


```
public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");

        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

Create beans through XML configuration



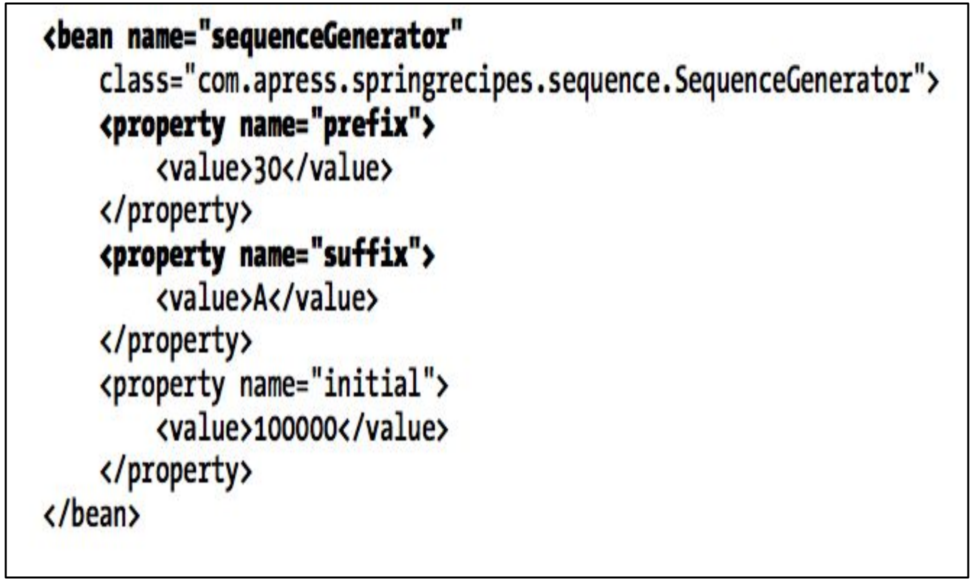
```xml
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
        <value>30</value>
    </property>
    <property name="suffix">
        <value>A</value>
    </property>
    <property name="initial">
        <value>100000</value>
    </property>
</bean>
```



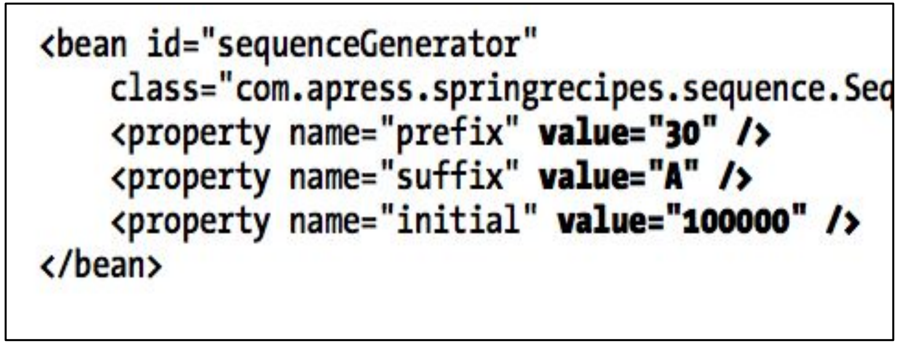
```java
public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, String
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
```

Setter injection

Shortcut for defining bean properties

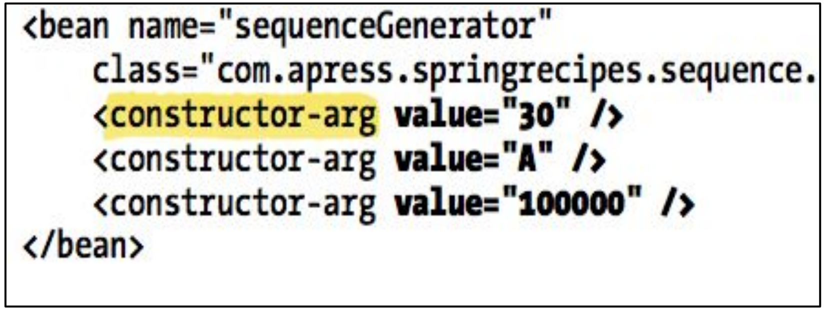


```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
        <value>30</value>
    </property>
    <property name="suffix">
        <value>A</value>
    </property>
    <property name="initial">
        <value>100000</value>
    </property>
</bean>
```

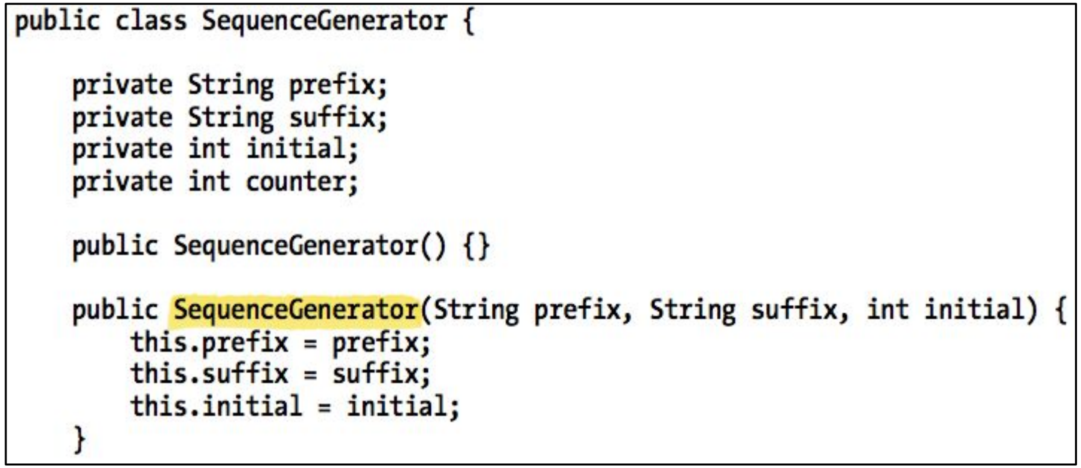


```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.Seq
    <property name="prefix" value="30" />
    <property name="suffix" value="A" />
    <property name="initial" value="100000" />
</bean>
```

Constructor injection

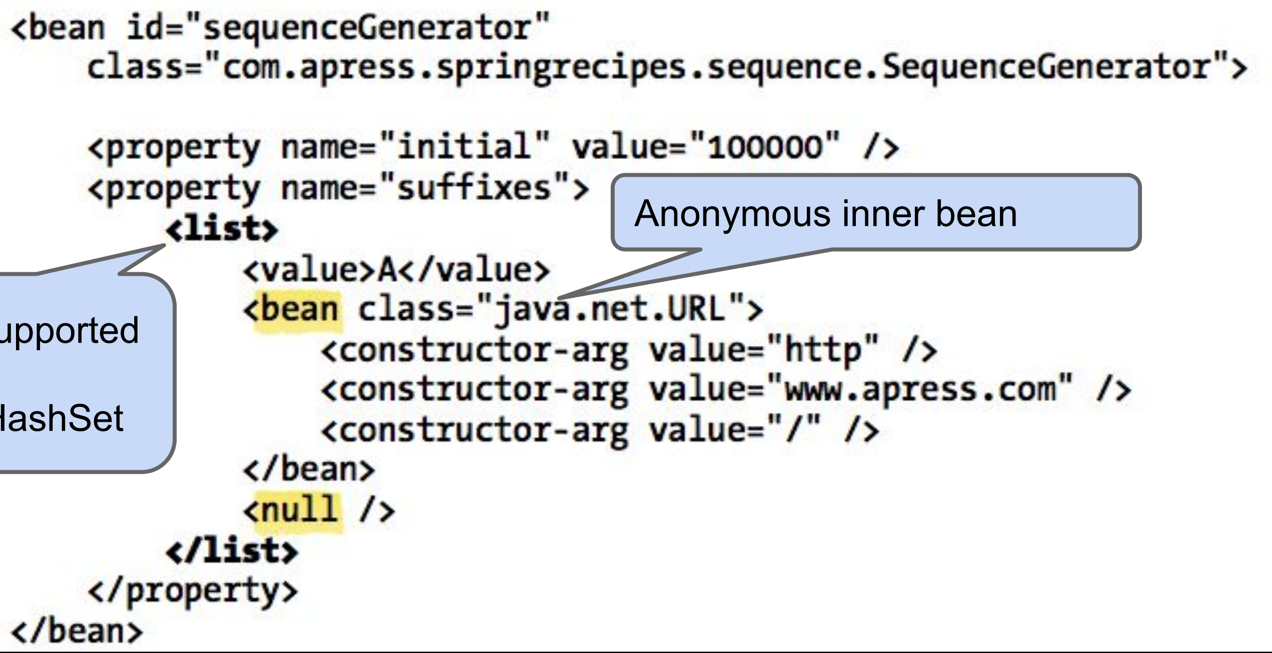


```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <constructor-arg value="100000" />
</bean>
```



```
public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }
```

Inject collection of objects



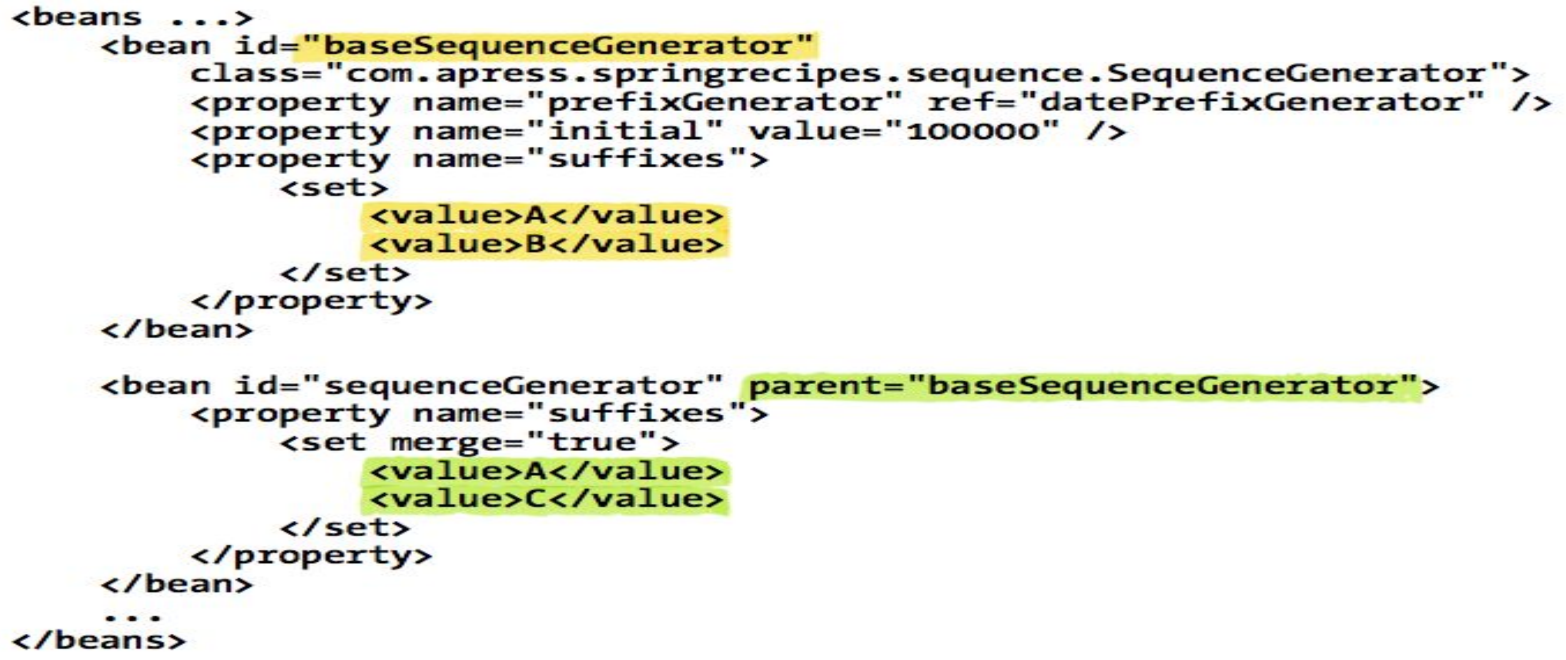
```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">

    <property name="initial" value="100000" />
    <property name="suffixes">
        <list>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </list>
    </property>
</bean>
```

Anonymous inner bean

Set is supported too with LinkedHashSet

Names vs ID: Both need to be unique, but neither required. Names can be multiple

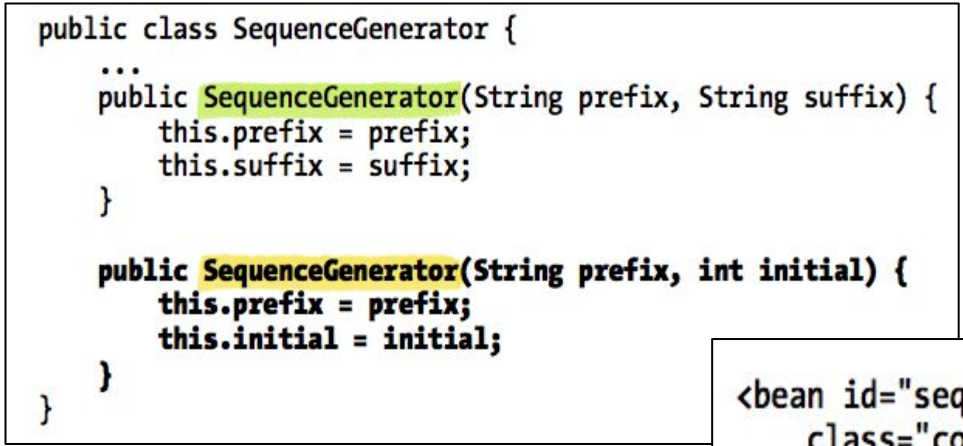
Parent/Child beans and property merging



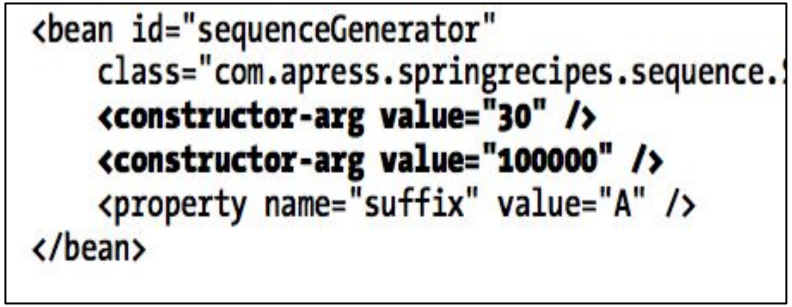
```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefixGenerator" ref="datePrefixGenerator" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <set>
                <value>A</value>
                <value>B</value>
            </set>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <set merge="true">
                <value>A</value>
                <value>C</value>
            </set>
        </property>
    </bean>
    ...
</beans>
```

What properties does the second bean have?

Resolving constructor ambiguity by type specification

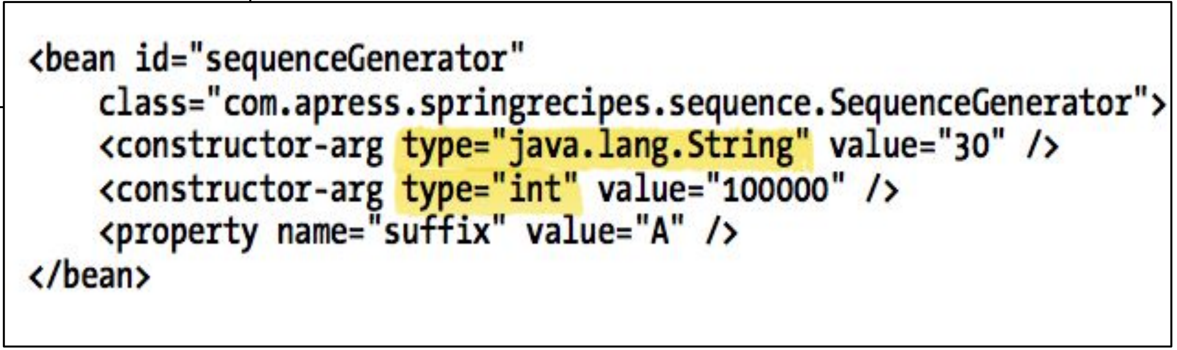


```java
public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
}
```



```xml
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.
    <constructor-arg value="30" />
    <constructor-arg value="100000" />
    <property name="suffix" value="A" />
</bean>
```

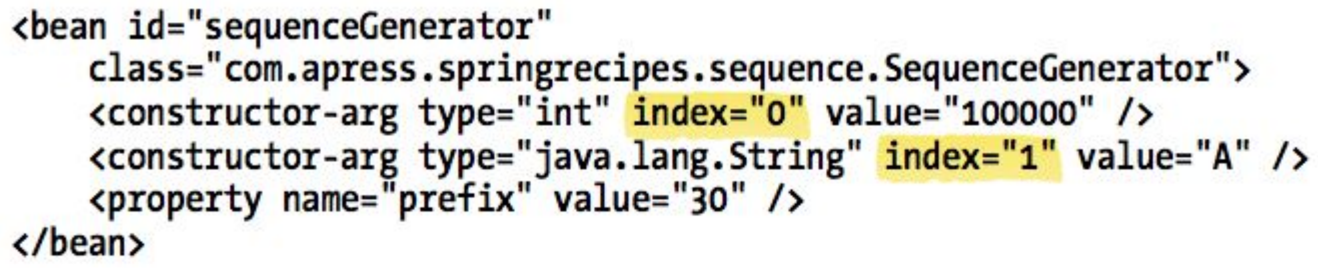
First Match



```xml
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="java.lang.String" value="30" />
    <constructor-arg type="int" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Explicit type specification helps!

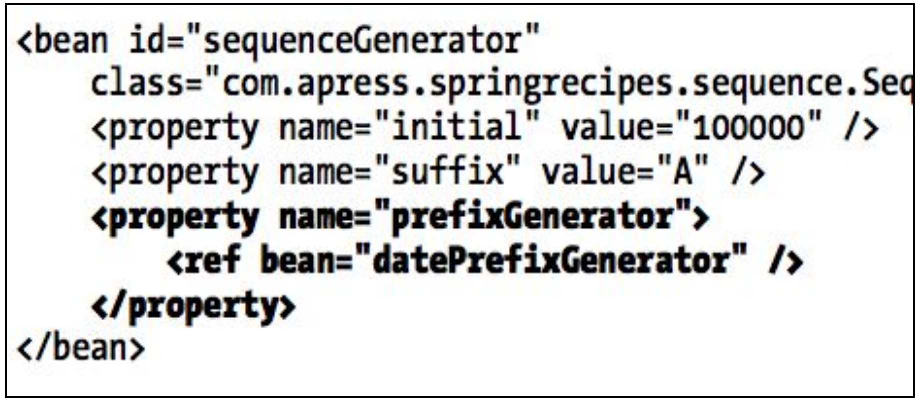
Constructor argument index can be used to de-ambiguate as well



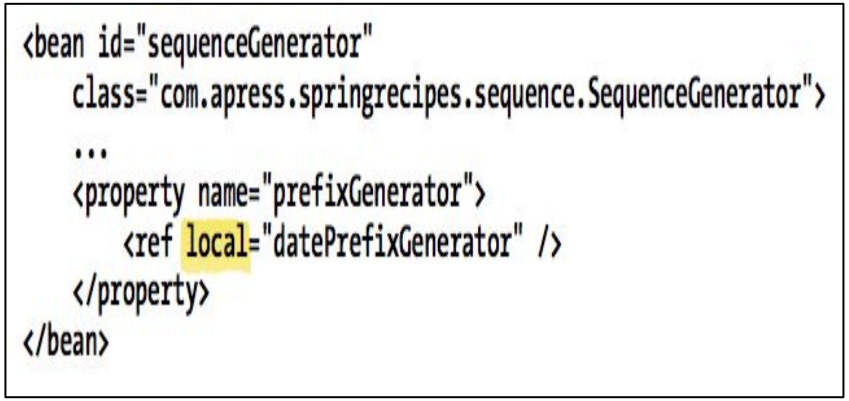
```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" index="0" value="100000" />
    <constructor-arg type="java.lang.String" index="1" value="A" />
    <property name="prefix" value="30" />
</bean>
```

- When is the index specification necessary?
- Do we a need to specify indexes for setter injections?

Reference beans



```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.Seq
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
        <ref bean="datePrefixGenerator" />
    </property>
</bean>
```



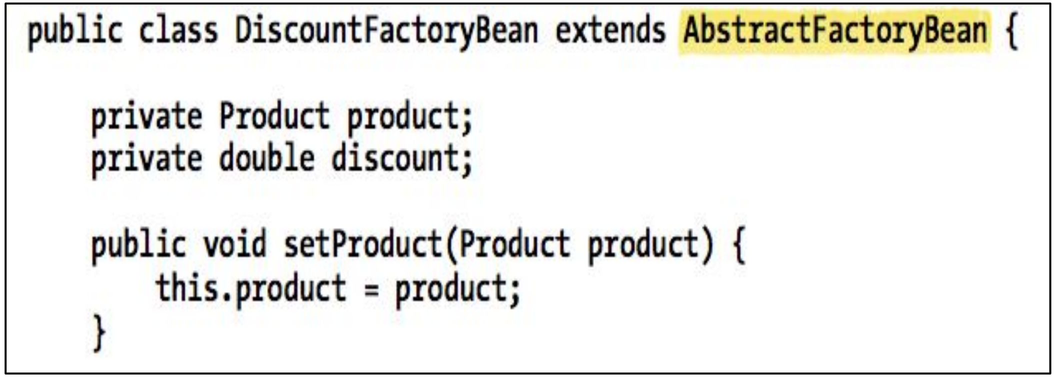
```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator">
        <ref local="datePrefixGenerator" />
    </property>
</bean>
```

Deprecated in Spring 4.0

What’s special about local?

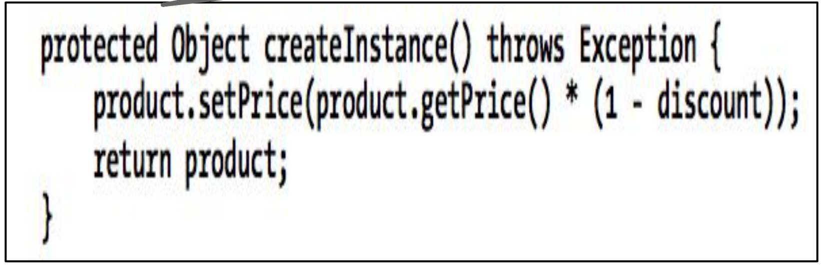
Create beans with factory beans

- Factory beans
  - Implementations of FactoryBean interface
  - Mostly used to implement framework facilities
    - E.g., LocalSessionFactoryBean for Hibernate session
    - Rarely need to implement your own



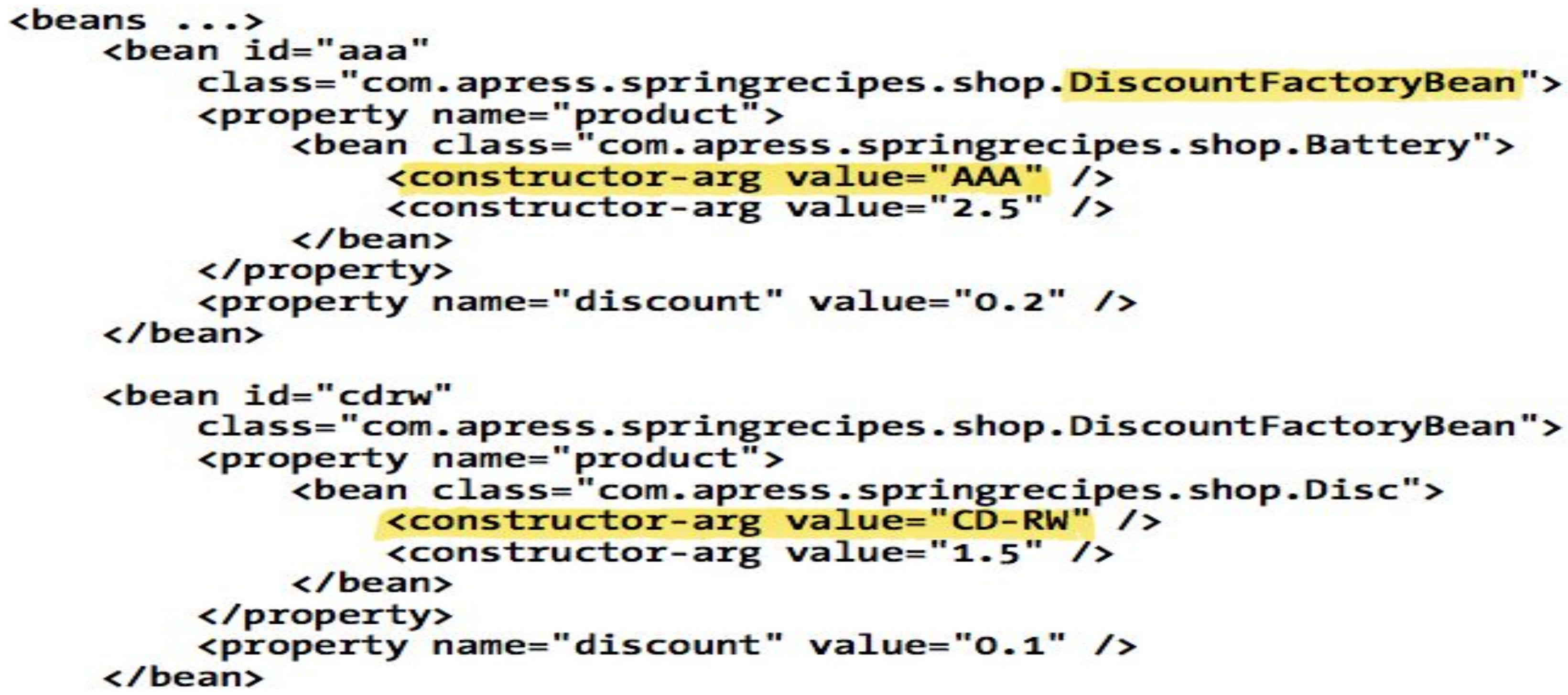
```
public class DiscountFactoryBean extends AbstractFactoryBean {

    private Product product;
    private double discount;

    public void setProduct(Product product) {
        this.product = product;
    }
```

Why is this *protected*?



```
protected Object createInstance() throws Exception {
    product.setPrice(product.getPrice() * (1 - discount));
    return product;
}
```

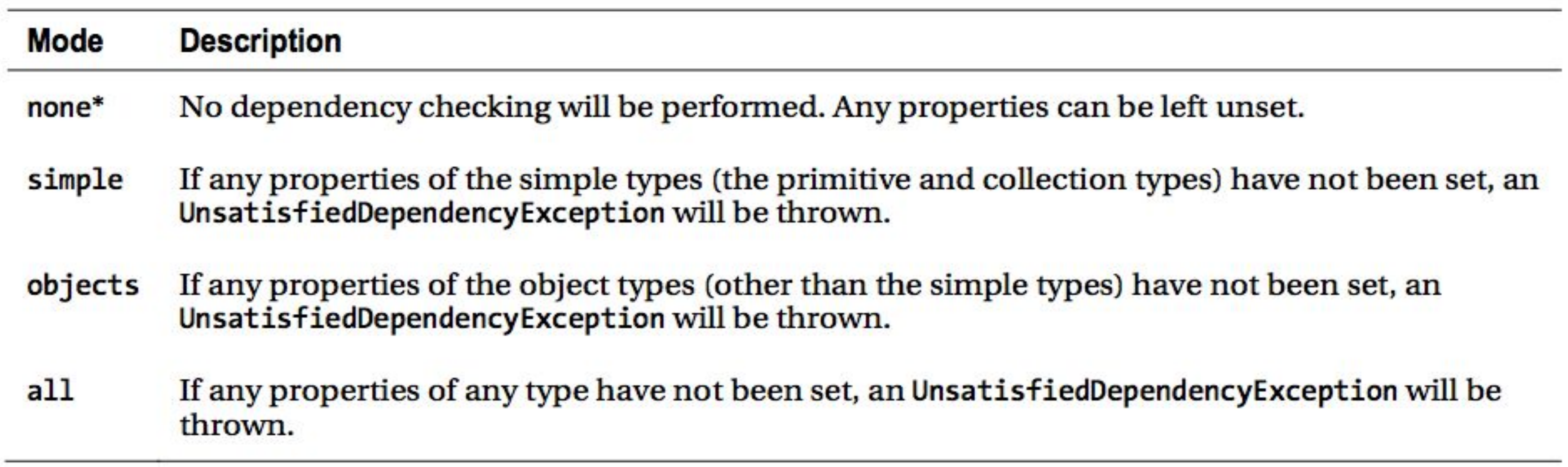
Factory bean in action



```
<beans ...>
    <bean id="aaa"
        class="com.apress.springrecipes.shop.DiscountFactoryBean">
        <property name="product">
            <bean class="com.apress.springrecipes.shop.Battery">
                <constructor-arg value="AAA" />
                <constructor-arg value="2.5" />
            </bean>
        </property>
        <property name="discount" value="0.2" />
    </bean>

    <bean id="cdrw"
        class="com.apress.springrecipes.shop.DiscountFactoryBean">
        <property name="product">
            <bean class="com.apress.springrecipes.shop.Disc">
                <constructor-arg value="CD-RW" />
                <constructor-arg value="1.5" />
            </bean>
        </property>
        <property name="discount" value="0.1" />
    </bean>
```

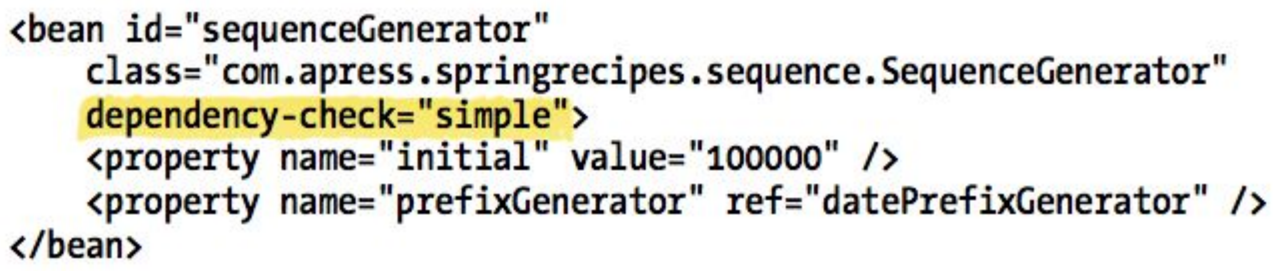
Check properties with dependency checking

- Shortcoming of setter injection
  - Hard to make sure a property is injected
- Spring allows checking by property types



Mode	Description
`none*`	No dependency checking will be performed. Any properties can be left unset.
`simple`	If any properties of the simple types (the primitive and collection types) have not been set, an `UnsatisfiedDependencyException` will be thrown.
`objects`	If any properties of the object types (other than the simple types) have not been set, an `UnsatisfiedDependencyException` will be thrown.
`all`	If any properties of any type have not been set, an `UnsatisfiedDependencyException` will be thrown.

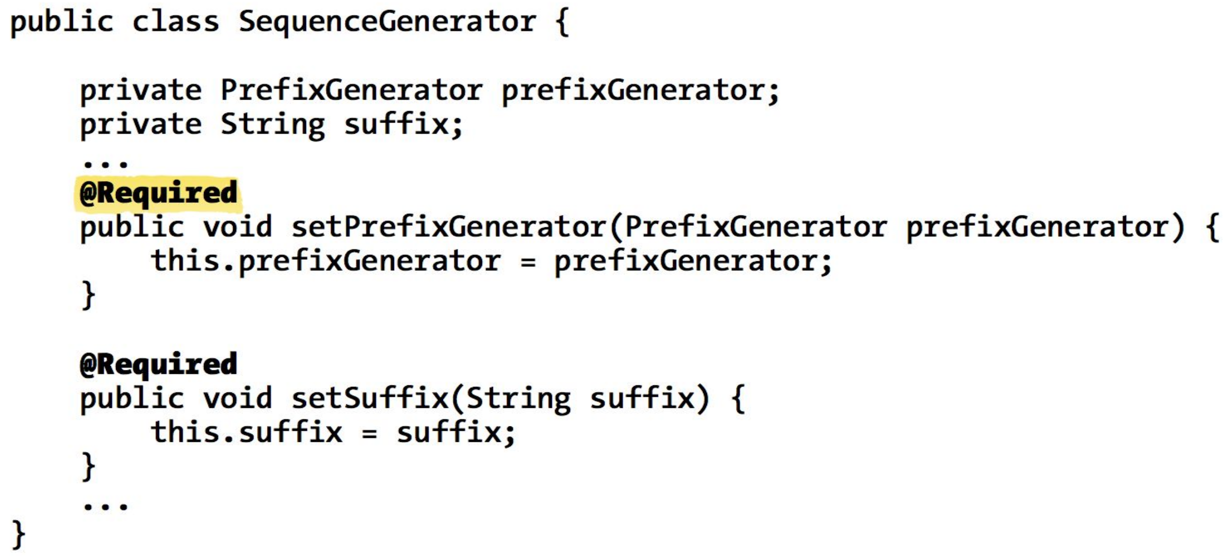
Type checking in action



```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="simple">
    <property name="initial" value="100000" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Useful? Probably, but not flexible enough...

@Required: Flexible dependency checking



```
public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Required
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    @Required
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    ...
}
```

- Explicitly create a RequiredAnnotationBeanPostProcessor bean, or
- Include <context:annotation-config>

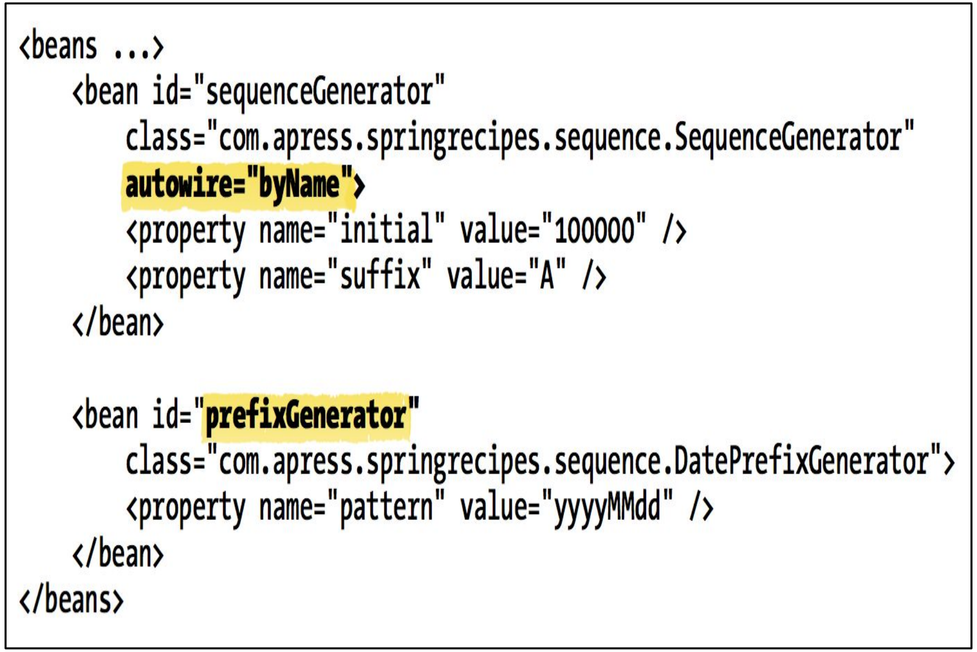
Auto-wiring by type

- Don’t have to explicitly specify all properties

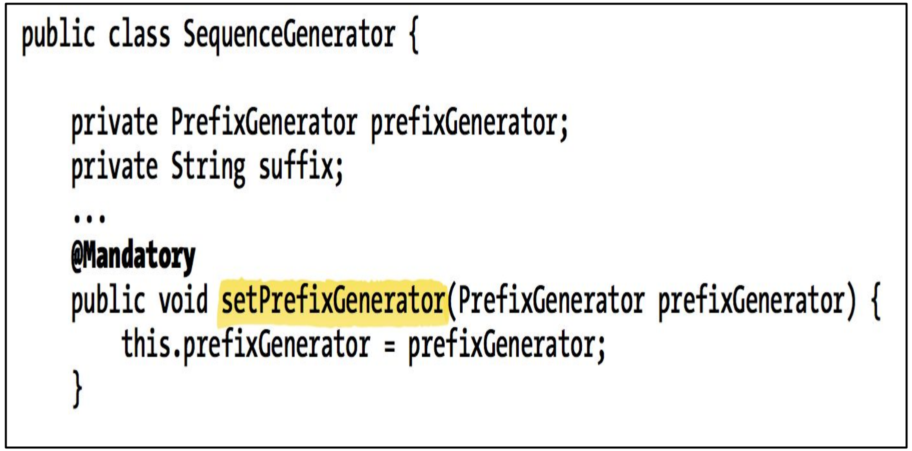
```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
    <bean id="yearPrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyy" />
    </bean>
</beans>
```

Avoid ambiguity!

Auto-wiring by name



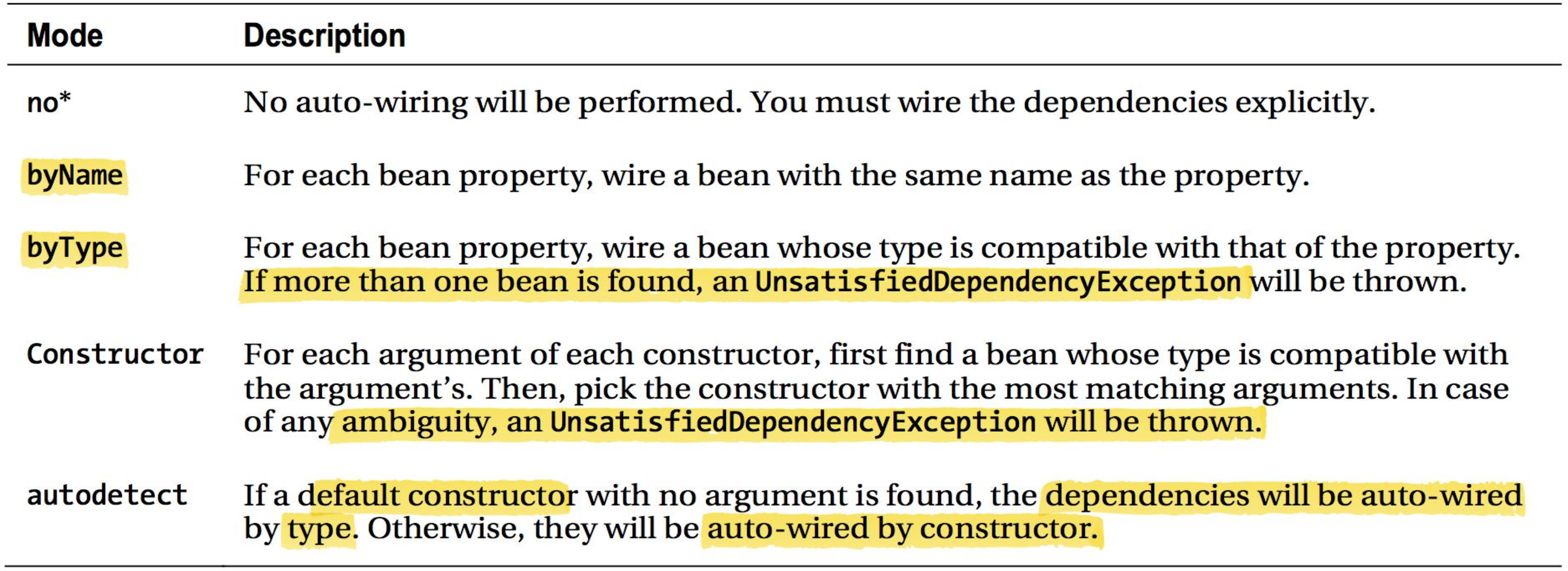
```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byName">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="prefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```



```
public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Mandatory
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
```

Have to get the name exactly right!

Complete auto-wiring options with XML



Mode	Description
no*	No auto-wiring will be performed. You must wire the dependencies explicitly.
byName	For each bean property, wire a bean with the same name as the property.
byType	For each bean property, wire a bean whose type is compatible with that of the property. If more than one bean is found, an UnsatisfiedDependencyException will be thrown.
Constructor	For each argument of each constructor, first find a bean whose type is compatible with the argument's. Then, pick the constructor with the most matching arguments. In case of any ambiguity, an UnsatisfiedDependencyException will be thrown.
autodetect	If a default constructor with no argument is found, the dependencies will be auto-wired by type. Otherwise, they will be auto-wired by constructor.

- Caveats: Lots of gory details!
- Avoid ambiguity, or avoid auto-wiring

Auto-Wiring Beans with @Autowired

```java
public class SequenceGenerator {
    ...
    @Autowired
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

```xml
<beans ...>
    ...
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

How does auto-wiring work: component scanning

```
@Component
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
    ...
}
```

Would it be too much scanning?

```
<beans ...>
    <context:component-scan base-package="com.apress.springrecipes.sequence">
        <context:include-filter type="regex"
            expression="com\.apress\.springrecipes\.sequence\..*Dao.*" />
        <context:include-filter type="regex"
            expression="com\.apress\.springrecipes\.sequence\..*Service.*" />
```

Narrow it down as much as possible

XML or annotation based injection?

- Convention over configuration
- Capability to adjust the wiring/specification without code change?
- Centralized view of components?
- Performance issues for auto-wiring

Summary

- What is software architecture?
- How to measure the performance of applications?
- Why do we layer enterprise applications?
- What is SOA and what are the pros and cons?
- What does Spring provide and why is it popular?
- What are IoC and DI?
- How do we configure and wire up beans in Spring?

Thanks!

Note: the content of the slides is heavily based on the book Spring Recipes: A Problem-Solution Approach