***Core Spring***

* Spring does a lot of things. But underneath all of the fantastic functionality it adds to enterprise development, its primary features are dependency injection (DI) and aspect-oriented programming (AOP).
* Spring is an open source framework, originally created by Rod Johnson.
* Spring was created to address the complexity of enterprise application development and makes it possible to use simple JavaBeans to achieve things that were previously only possible with EJB.
* Any java application can benefit from spring in terms of simplicity, testability, and loose coupling.
* **Spring simplifies Java development.**
* Spring uses four key strategies:
  + Lightweight and minimally invasive development with POJOs or Java Beans
  + Loose coupling through DI and interface orientation
  + Declarative programming through aspects and common conventions
  + Eliminating boilerplate code with aspects and templates.
* Spring avoids (as much as possible) littering your application code with its API.
* Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class. Instead, the classes in a Spring-bases application often have no indication that they’re being used by Spring. At worst a class may be annotated with one of Springs’s annotation, but it’s otherwise a POJO.

***Injecting Dependencies***

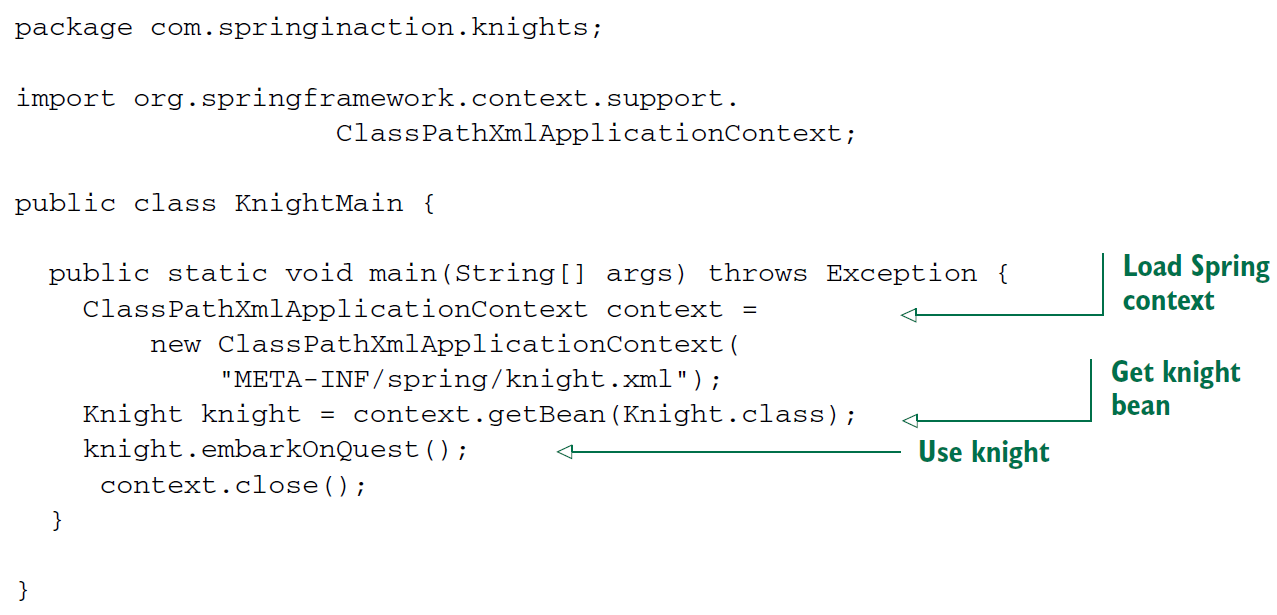
* The phrase *dependency injection* may sound intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, DI isn’t nearly as complex as it sounds. By applying DI in your projects, you’ll find that your code will become significantly simpler, easier to understand, and easier to test.

**How DI works**

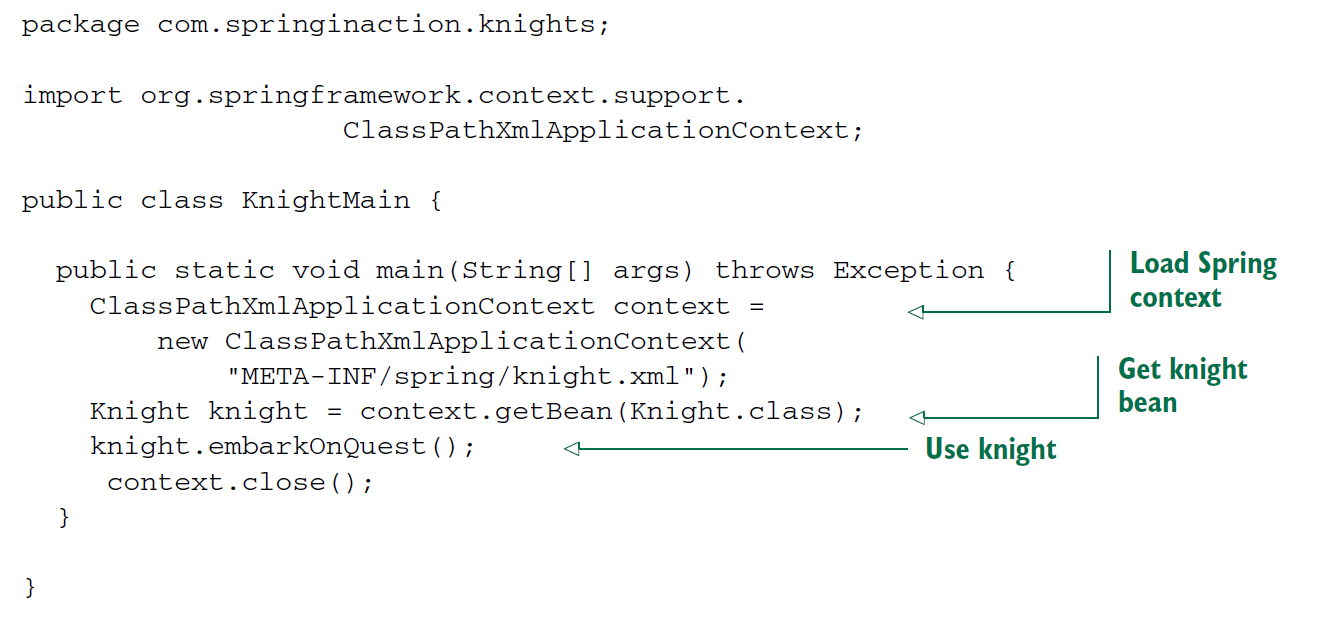
* Any nontrivial application (pretty much anything more complex than a Hello World example) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code.
* Coupling object is a two-headed beast. On the one hand, tightly coupled code is difficult to test, difficult to reuse, and difficult to understand, and it typically exhibits buggy behavior (fixing one bug results in the creation of one or more new bugs)
* On the other hand, a certain amount of coupling is necessary – Completely uncoupled code doesn’t do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but should be carefully managed.
* With **DI,** objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren’t expected to create or obtain their dependencies.
* Dependencies are injected into the objects that need them.
* Key benefit of DI- loose coupling. If an object only knows about its dependencies by their interface (not by their implementation or how they’re instantiated), then the dependency can be swapped out with a different implementation without the depending object knowing the difference.

**Wiring**

* The act of creating associations between application components is commonly referred to as *wiring.*
* In spring, there are many ways to wire components together, but a common approach has always been via XML.
* Application context loads bean definitions and wires them together. The Spring application context is fully responsible for the creation of and wiring of the objects that make up the application.
* Spring comes with several implementations of its application context, each primarily differing only in how it loads its configuration.
* When the beans in knights.xml are declared in an XML file, an appropriate choice for application context might be *ClassPathXmlApplicationContext.* This Spring context implementation loads the Spring context from one or more XML files located in the application’s classpath. The main() method in the following listing uses *ClassPahtXmlApplicationContext* to load knights.xml and to get a reference to the *Knight* object.

****

* Whether you use XML-based or Java-based configuration, the benefits of DI are the same. Although BraveKnight depends on a Quest, it doesn’t know what type of Quest it will be given or where that Quest will come from. Like Wise, SlayDragonQuest depends on a PrintStream, but it isn’t coded with knowledge of how that PrintStream comes to be. **Only Spring, thorough its configuration, knows how all the pieces come together.**  This makes it possible to change those dependencies with no changes to the depending classes.
* This example has shown approach to wiring beans in Spring. Don’t concern yourself too much with the details right now. We’ll dig more in Spring configuration later.
* We will look at ways that beans cab be wired in Spring, including a way to let Spring automatically discover beans and create the relationships between them.

****

* For Java-based configuration, Spring offers *AnnotationConfigApplicationContext.*
* Here the main() creates the Spring application context based on the knights.xml file. Then it uses the application context as factory to retrieve the bean whose ID is *knight.* With a reference to Knight object, it calls the embarkOnQuest() method to have the knight embark on the Quest he was given. Note that this class knows nothing about which type of Quest your hero has. For that matterm it’s blissfully unaeare of the fact that it’s dealing with BraveKnight. Only the knight.xml file knows for sure what the implementations are.