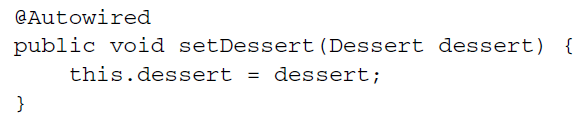
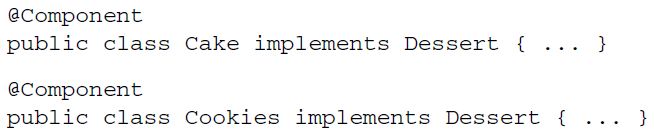
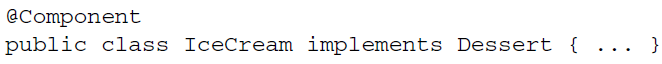
***Addressing ambiguity in autowiring***

* Autowiring is a huge help because it reduces the amount of explicit configuration necessary to assemble application components.
* But autowiring only works when exactly one bean matches the desired result. When there’s more than one matching bean, the ambiguity prevents Spring from autowiring the property, constructor argument, or method parameter.
* To illustrate autowiring ambiguity, suppse you’ve annotated the following *set-Dessert()* method with *@Autowired:*

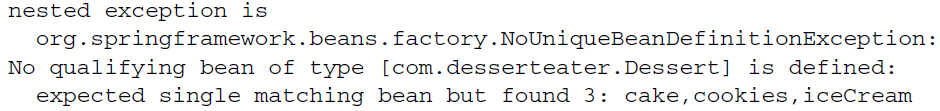


* In this example, *Dessert* is an interface and is implemented by three classes: *Cakes, Cookies,* and *IceCream:*





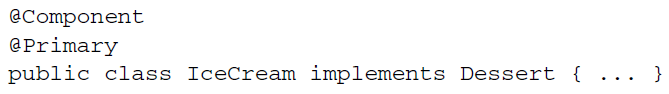
* Because all three implementation are annotated by *@Component,* they’re all picked up during component-scanning and created as beans in the Spring application context. Then, when Spring tries to autowire the *Dessert* parameter in *setDessert(),* it doesn’t have a single, unambiguous choice. Spring has no option but to fail and throw an exception. To be precise, Spring throws a NoUniqueBeanDefinitionException:



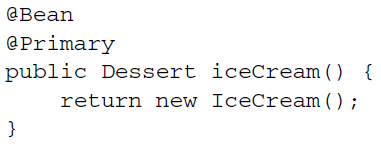
* In reality, autowiring ambiguity is more rare than you’d expect. Even though such ambiguity is a real problem, more often than not there’s only one implementation of a given type, and autowiring works perfectly.
* For those times when ambiguity does happen, however, Spring offers a couple of options. You can declare one of the candidate beans as the primary choice, or you can use qualifiers to help Spring narrow its choices to a single candidate.

***Designating a primary bean***

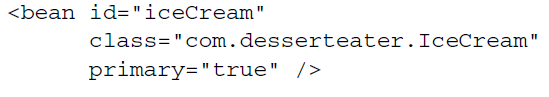
* When declaring beans, you can avoid autowiring ambiguity by designating one of the candidate beans as a primary bean. In the event of any ambiguity, Spring will choose the primary bean over any other candidate beans.
* Let’s say that ice cream is your favorite dessert. You can express that favorite choice in Spring using the *@Primary* annotation.
* *@Primary* can be used either alongside *@Component* for beans that are component-scanned or alongside *@Bean* for beans declared in Java Configuration.
* For example, here’s how you might declare the *@Component-*annotated *IceCream* bean as the primary choice:



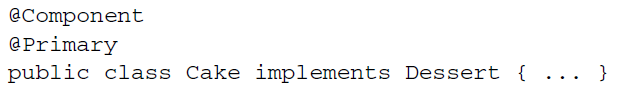
* Or, if you’re declaring the *IceCream* bean explicitly in Java Configuration, the *@Bean* method might look like this:



* If you’re configuring your beans in XML, you’re not left out. The *<bean>* element has a primary attribute to specify a primary bean:



* No matter how you designate a primary bean, the effect is the same. You’re telling Spring that it should choose the primary bean in the case of ambiguity.
* This works well right up to the point where you designate two or more primary beans. For example, suppose the *Cake* class looks like this:

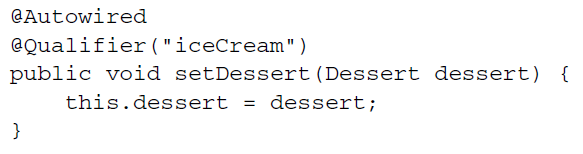


* Now there are two primary *Dessert* beans: *Cake* and *IceCream.* This poses a new ambiguity issue. Just as Spring couldn’t choose among multiple candidate beans, it can’t choose among multiple primary beans. Clearly, when more than one bean is designated as primary, there are no primary candidates.

***Qualifying autowired beans***

The limitation of primary beans is that *@Primary* doesn’t limit the choices to a single unambiguous option. It only designates a preferred option. When there’s more than one primary, there’s not much else you can do to narrow the choices further.

* In Contrast, Spring’s qualifiers apply a narrowing operation to all candidate beans, ultimately arriving at the single bean that means the prescribed qualifications.
* If ambiguity still exists after applying all qualifiers, you can always apply more qualifiers to narrow the choices further.
* The *@Qualifier* annotation is the main way to work with qualifiers. It can be applied alongside *@Autowired* or *@Inject* at the point of injection at the point of injection to specify which bean you want to be injected. For example, let’s say you want to ensure that the *IceCream* bean is injected into *setDessert()*:



* This is a prime example of qualifiers in their simplest form. The parameter given to *@Qualifier* is the ID of the bean that you to inject. All *@Component-*annotated classes will be created as beans whose ID is the uncapitalized class name. Therefore, *@Qualifiers(“iceCream”)* refers to the bean created when component-scanning created an instance of the *IceCream* class.
* Actually, there’s a bit more to the story than that. To be more precise, *@Qualifier(“iceCream”)* refers to the bean that has the *String* “iceCream” as a qualifier. For lack of having specified any other qualifiers, all beans are given a default qualifier that’s the same as their bean ID. Therefore, the *setDessert()* method will be injected with the bean that has “iceCream” as a qualifier. That just happens to be the bean whose ID is *iceCream*, created when the *IceCream* class was component-scanned.