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:

```
<bean id="iceCream"
      class="com.desserteater.IceCream"
      primary="true" />
```

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:

```
@Component
@Primary
public class Cake implements Dessert { ... }
```

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.

For a more powerful ambiguity-busting mechanism, let's look at qualifiers.

3.3.2 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 meets 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 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()`:

```
@Autowired
@Qualifier("iceCream")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

This is a prime example of qualifiers in their simplest form. The parameter given to `@Qualifier` is the ID of the bean that you want to inject. All `@Component`-annotated classes will be created as beans whose ID is the uncapitalized class name. Therefore, `@Qualifier("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.

Basing qualification on the default bean ID qualifier is simple but can pose some problems. What do you suppose would happen if you refactored the `IceCream` class, renaming it `Gelato`? In that case, the bean's ID and default qualifier would be `gelato`, which doesn't match the qualifier on `setDessert()`. Autowiring would fail.

The problem is that you specified a qualifier on `setDessert()` that is tightly coupled to the class name of the bean being injected. Any change to that class name will render the qualifier ineffective.

CREATING CUSTOM QUALIFIERS

Instead of relying on the bean ID as the qualifier, you can assign your own qualifier to a bean. All you need to do is place the `@Qualifier` annotation on the bean declaration. For example, it can be applied alongside `@Component` like this:

```
@Component
@Qualifier("cold")
public class IceCream implements Dessert { ... }
```

In this case, a qualifier of `cold` is assigned to the `IceCream` bean. Because it's not coupled to the class name, you can refactor the name of the `IceCream` class all you want without worrying about breaking autowiring. It will work as long as you refer to the `cold` qualifier at the injection point:

```
@Autowired
@Qualifier("cold")
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

It's worth noting that `@Qualifier` can also be used alongside the `@Bean` annotation when explicitly defining beans with Java configuration:

```
@Bean
@Qualifier("cold")
public Dessert iceCream() {
    return new IceCream();
}
```

When defining custom `@Qualifier` values, it's a good practice to use a trait or descriptive term for the bean, rather than using an arbitrary name. In this case, I've described the `IceCream` bean as a "cold" bean. At the injection point, it reads as "give me the cold dessert," which happens to describe `IceCream`. Similarly, I might describe `Cake` as "soft" and `Cookies` as "crispy."