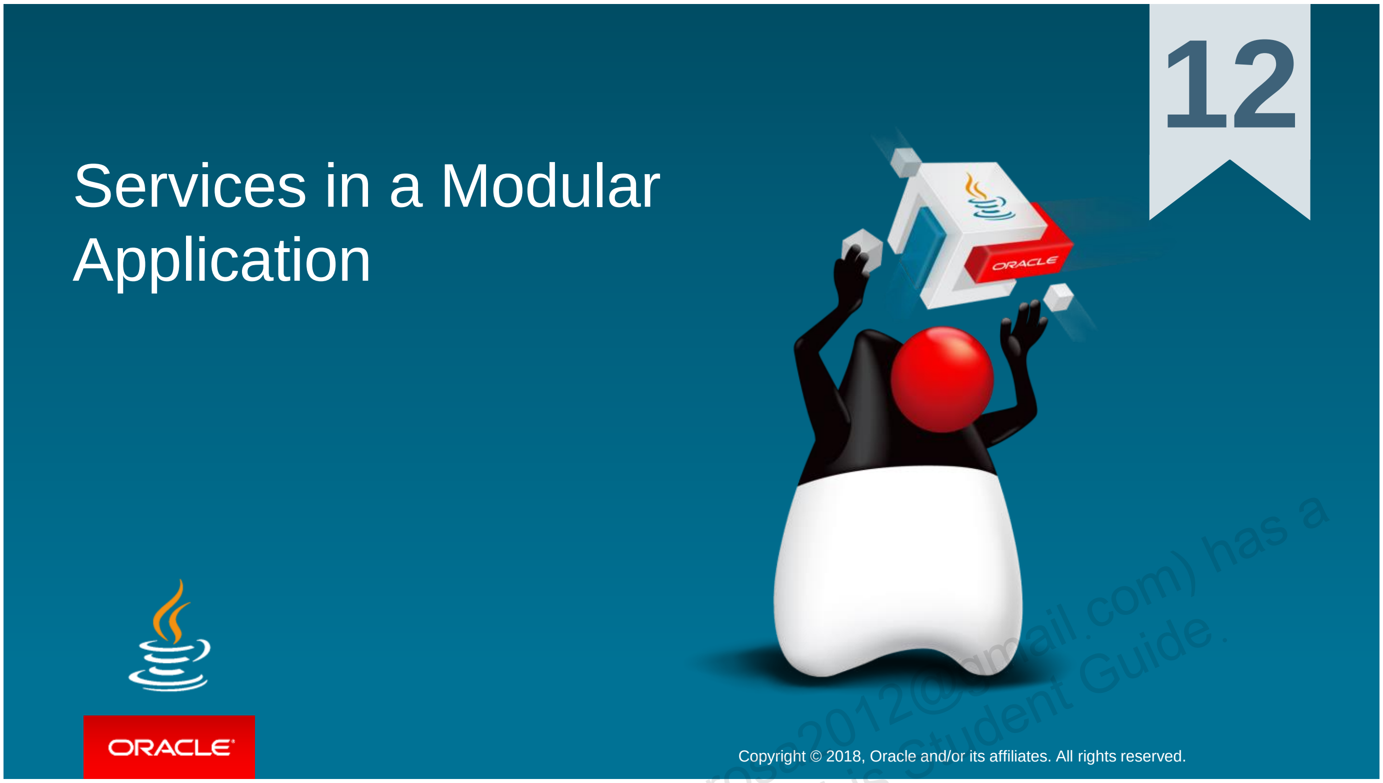
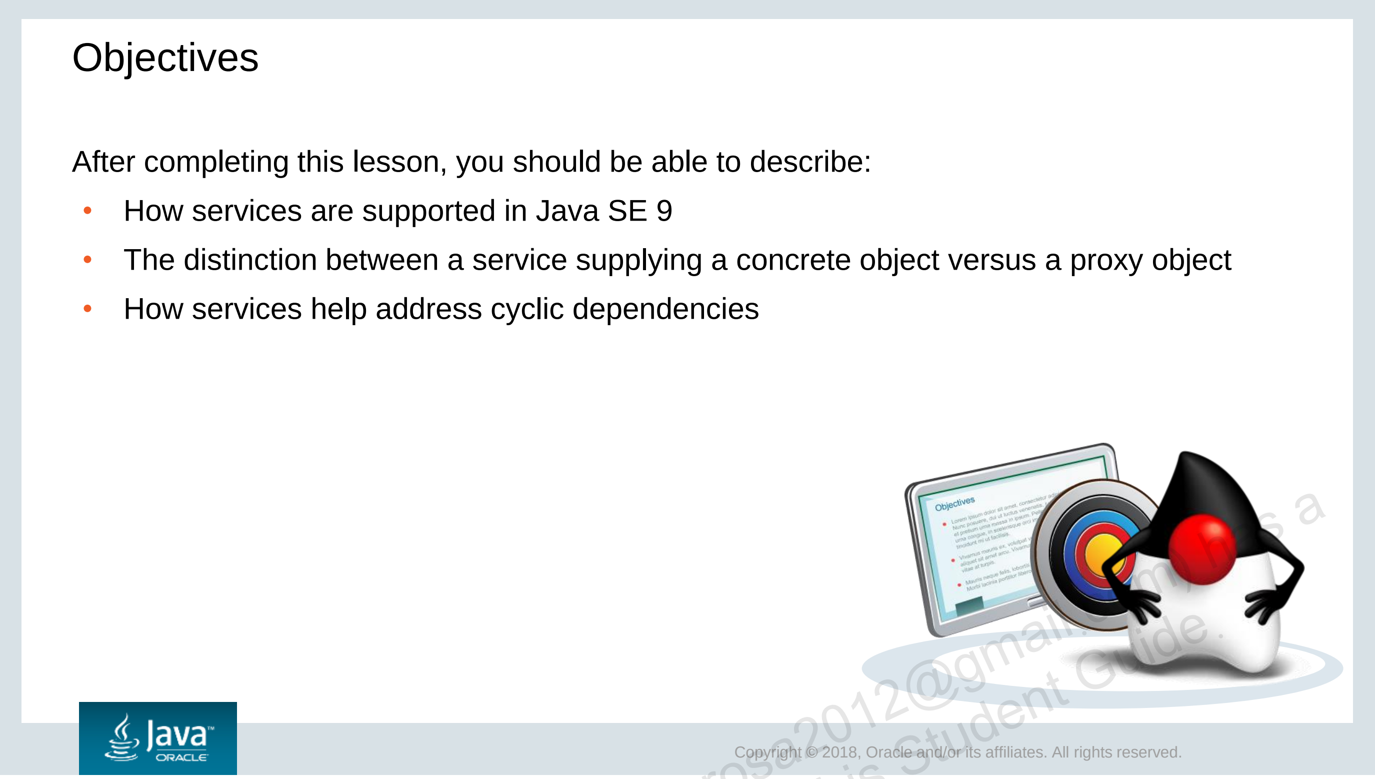


12

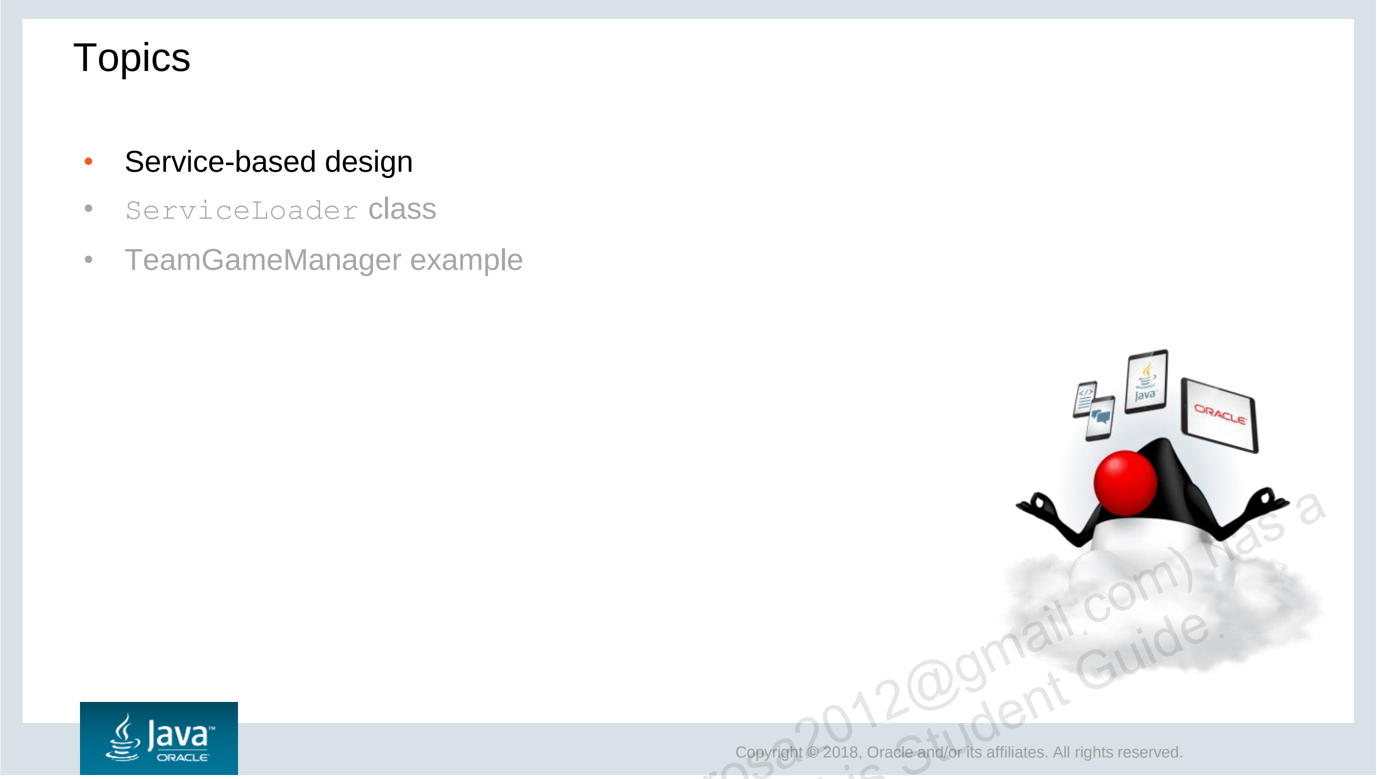
Services in a Modular Application



Objectives

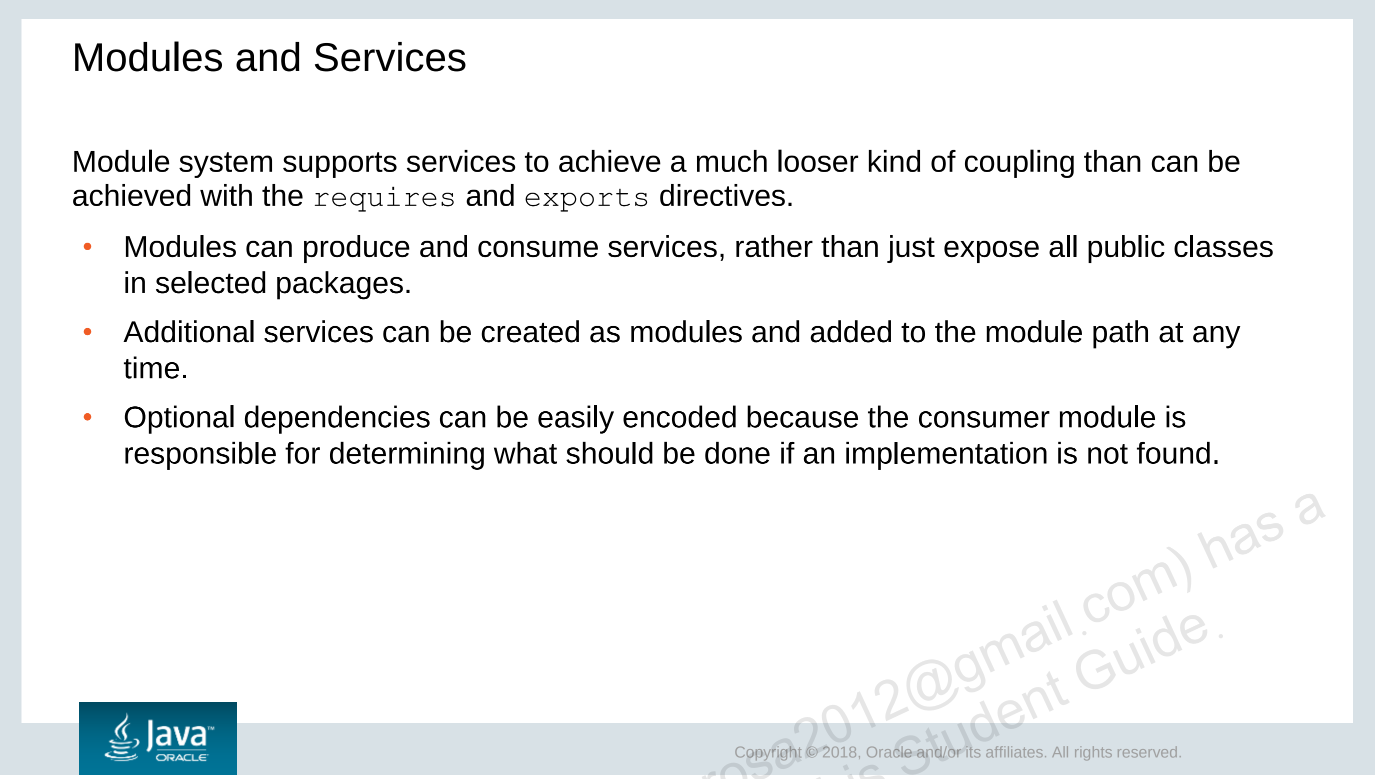
After completing this lesson, you should be able to describe:

- How services are supported in Java SE 9
- The distinction between a service supplying a concrete object versus a proxy object
- How services help address cyclic dependencies



Topics

- Service-based design
- `ServiceLoader` class
- TeamGameManager example

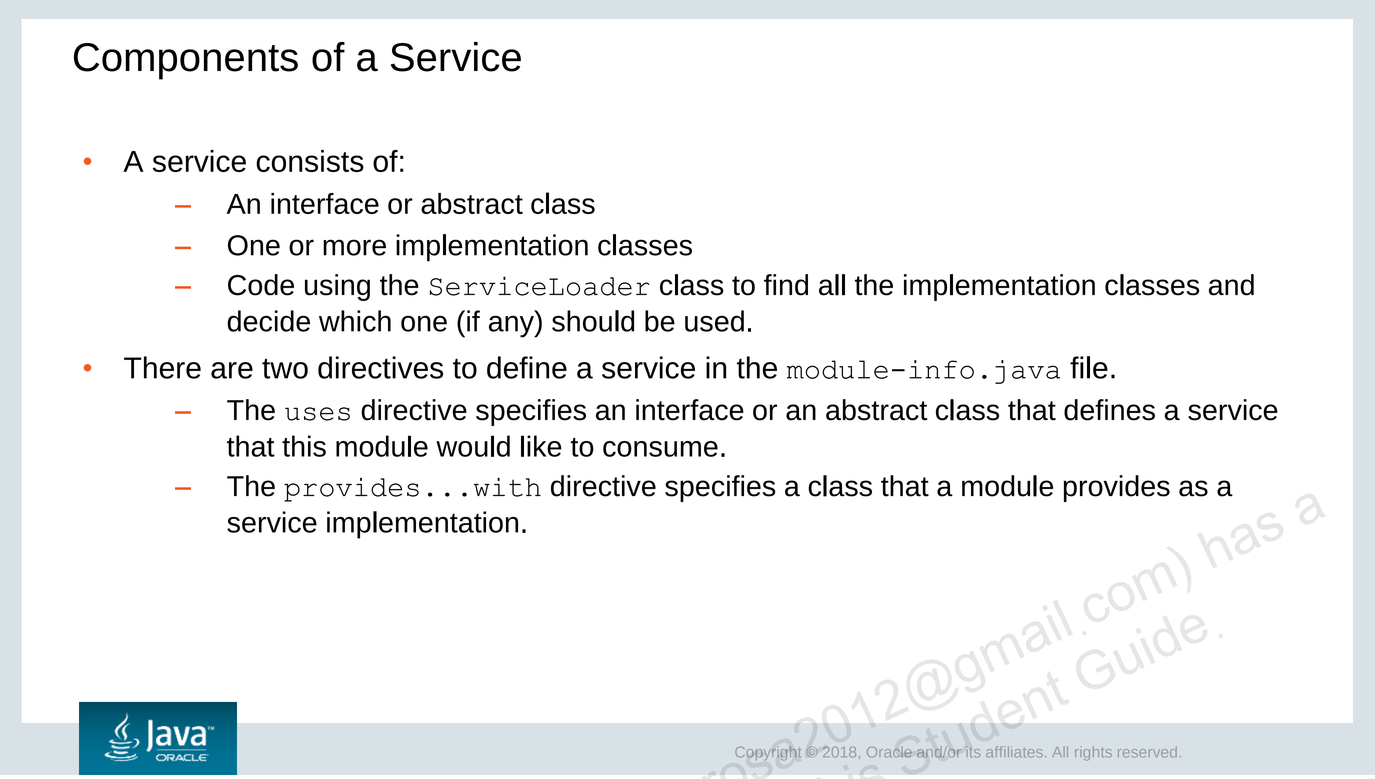


Modules and Services

Module system supports services to achieve a much looser kind of coupling than can be achieved with the `requires` and `exports` directives.

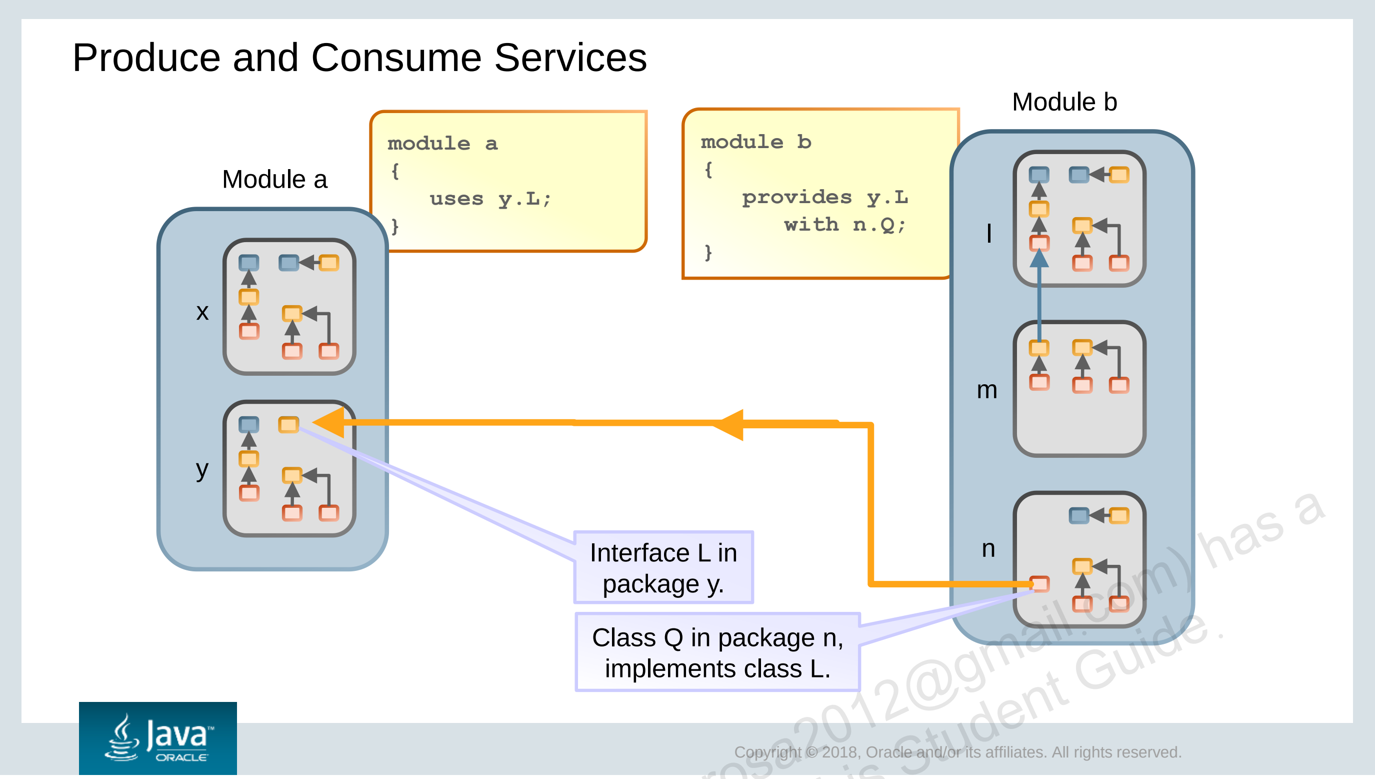
- Modules can produce and consume services, rather than just expose all public classes in selected packages.
- Additional services can be created as modules and added to the module path at any time.
- Optional dependencies can be easily encoded because the consumer module is responsible for determining what should be done if an implementation is not found.

In Java, one should program to an interface, not an implementation. In JDK 9, the module system introduces “services” to make this concept explicit; modules declare which interfaces they program to, and the module system automatically discovers implementations.



Components of a Service

- A service consists of:
  - An interface or abstract class
  - One or more implementation classes
  - Code using the `ServiceLoader` class to find all the implementation classes and decide which one (if any) should be used.
- There are two directives to define a service in the `module-info.java` file.
  - The `uses` directive specifies an interface or an abstract class that defines a service that this module would like to consume.
  - The `provides...with` directive specifies a class that a module provides as a service implementation.



Produce and Consume Services

```
module a
{
   uses y.L;
}
```

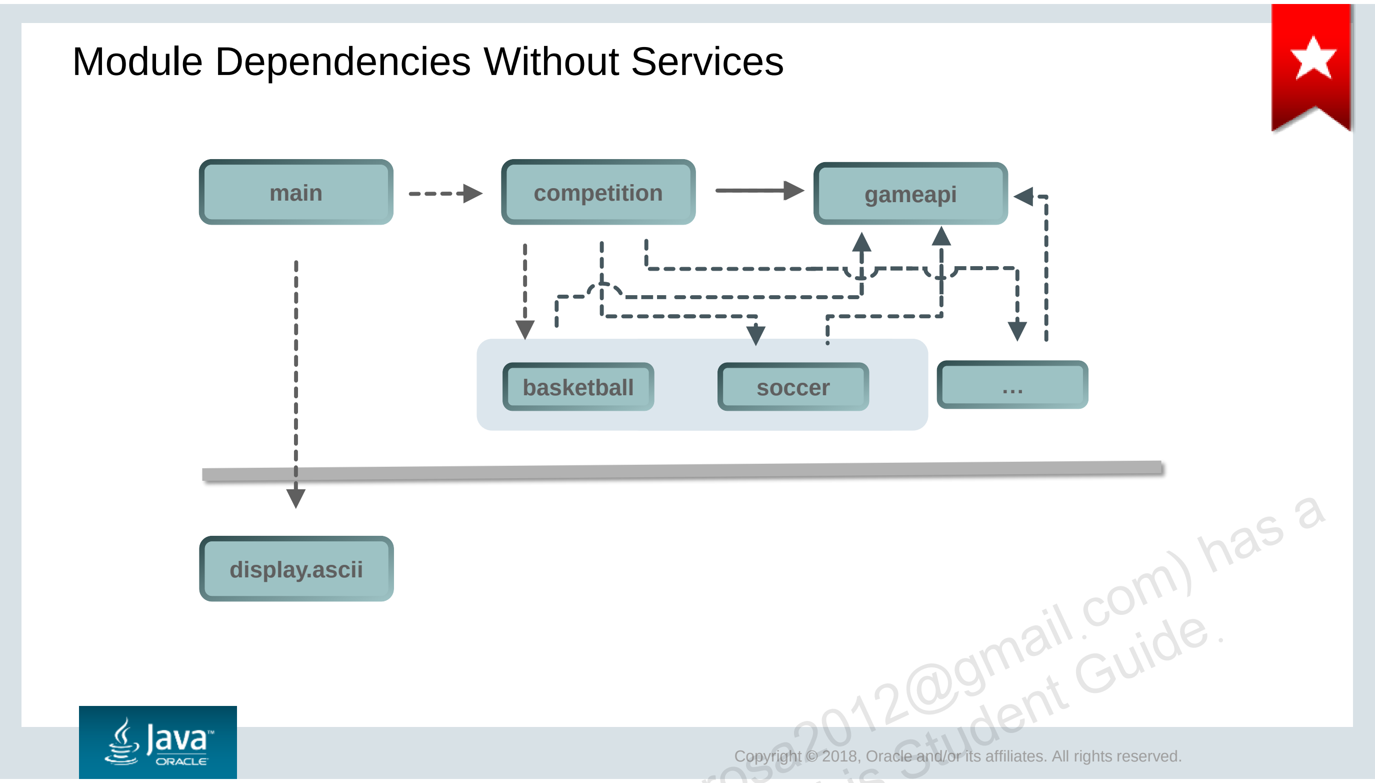
```
module b
{
   provides y.L
      with n.Q;
}
```

Interface L in package y.

Class Q in package n, implements class L.

A uses module directive specifies a service used by this module—making module a a service consumer. A service is an object of a class that implements the interface or extends the abstract class specified in the uses directive.

A provides…with module directive specifies that a module provides a service implementation—making the module a service provider. The provides part of the directive specifies an interface or abstract class listed in a module's uses directive and the with part of the directive specifies the name of the service provider class that implements the interface or extends the abstract class.



Module Dependencies Without Services

main

competition

gameapi

basketball

soccer

...

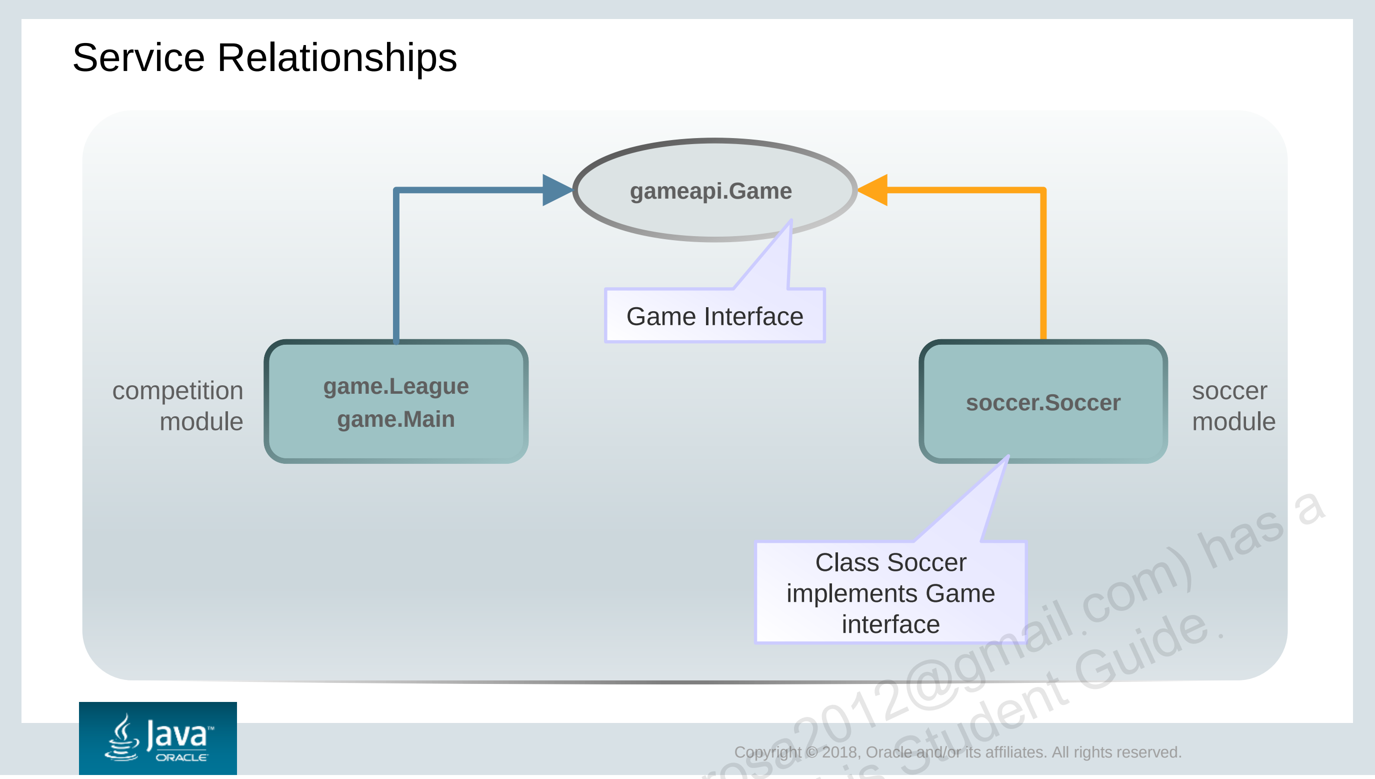
display.ascii

The graphic in the slide represents an application that runs a league and provides for future extension in the form of new games to be supported. Currently basketball and soccer are supported. To provide support for another game, perform the following steps:

- Create a new module, for example, the `lacrosse` module, with a dependency on `gameapi`.
- In the lacrosse module, implement the interfaces that are required by the `competition` module to run a league of lacrosse games.
- Add functionality to the `lacrosse` module to represent the game of lacrosse.
- Use the `exports` keyword to ensure that the necessary implementations are available to the `competition` module.
- Rewrite the `Factory` class in `competition` so that it can create the needed objects for the game of lacrosse.

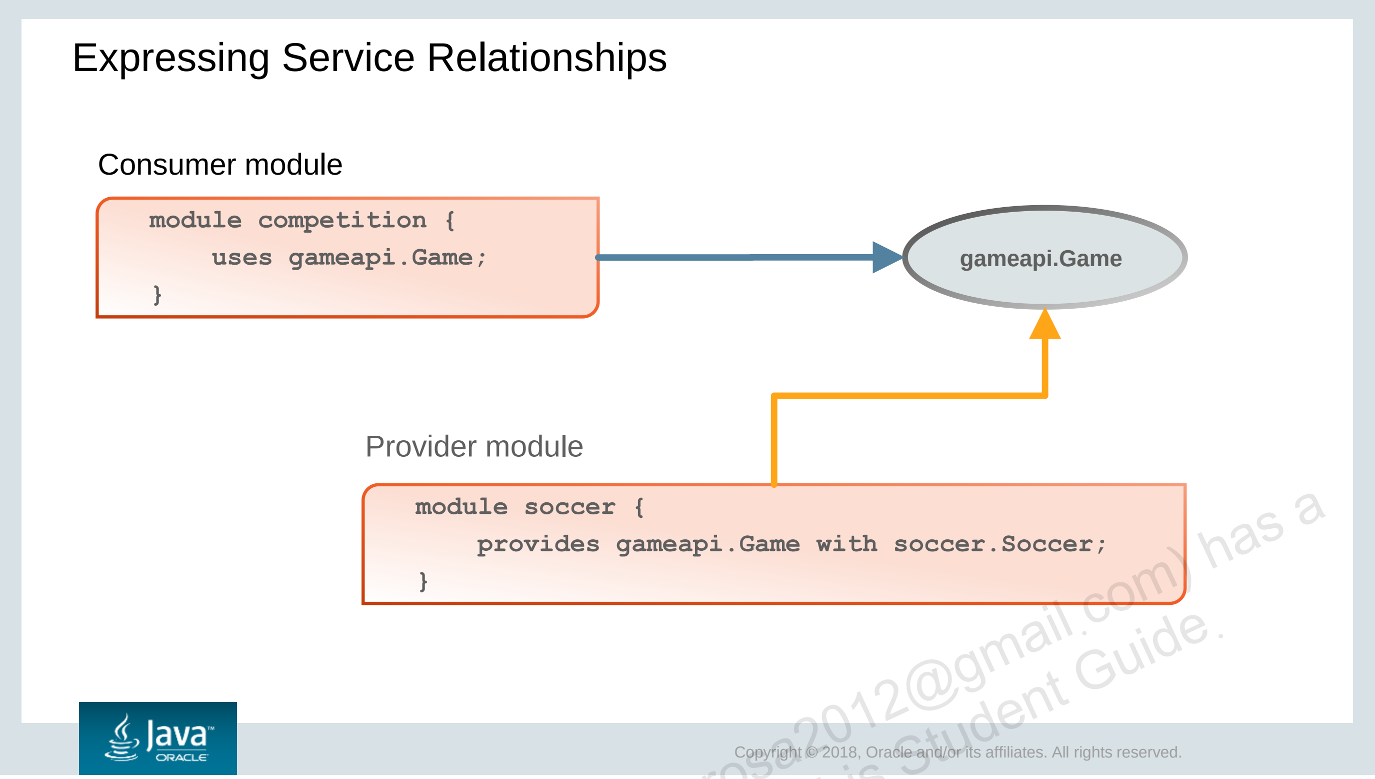
There are a lot of dependencies! It's not enough to create the new module; existing modules also need to be modified to accommodate the new module. Also, notice that if you decide to no longer support a particular game (for example, basketball), it's not enough to simply not include it in the module path. You will also need to modify the `Factory` class in the `competition` module.

This is a very tightly coupled architecture.



Service Relationships

Here’s how it works. A module requests an implementation of an interface, which is provided by another module (or, in some cases, even by the same module). In the example shown in the slide, the `Game` interface implemented by the `Soccer` class can be implemented by any number of other classes, like, for example, the `Basketball` class.



Expressing Service Relationships

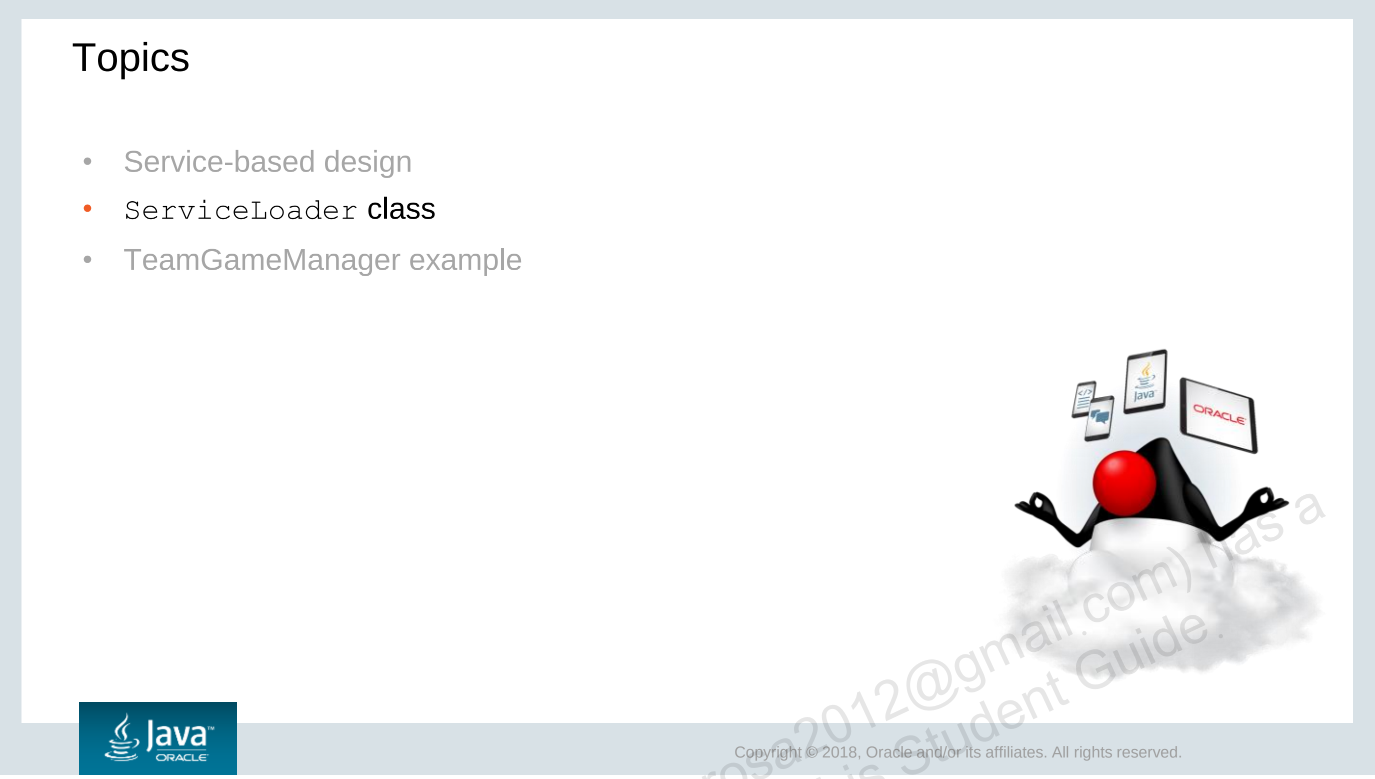
Consumer module

```
module competition {
    uses gameapi.Game;
}
```

Provider module

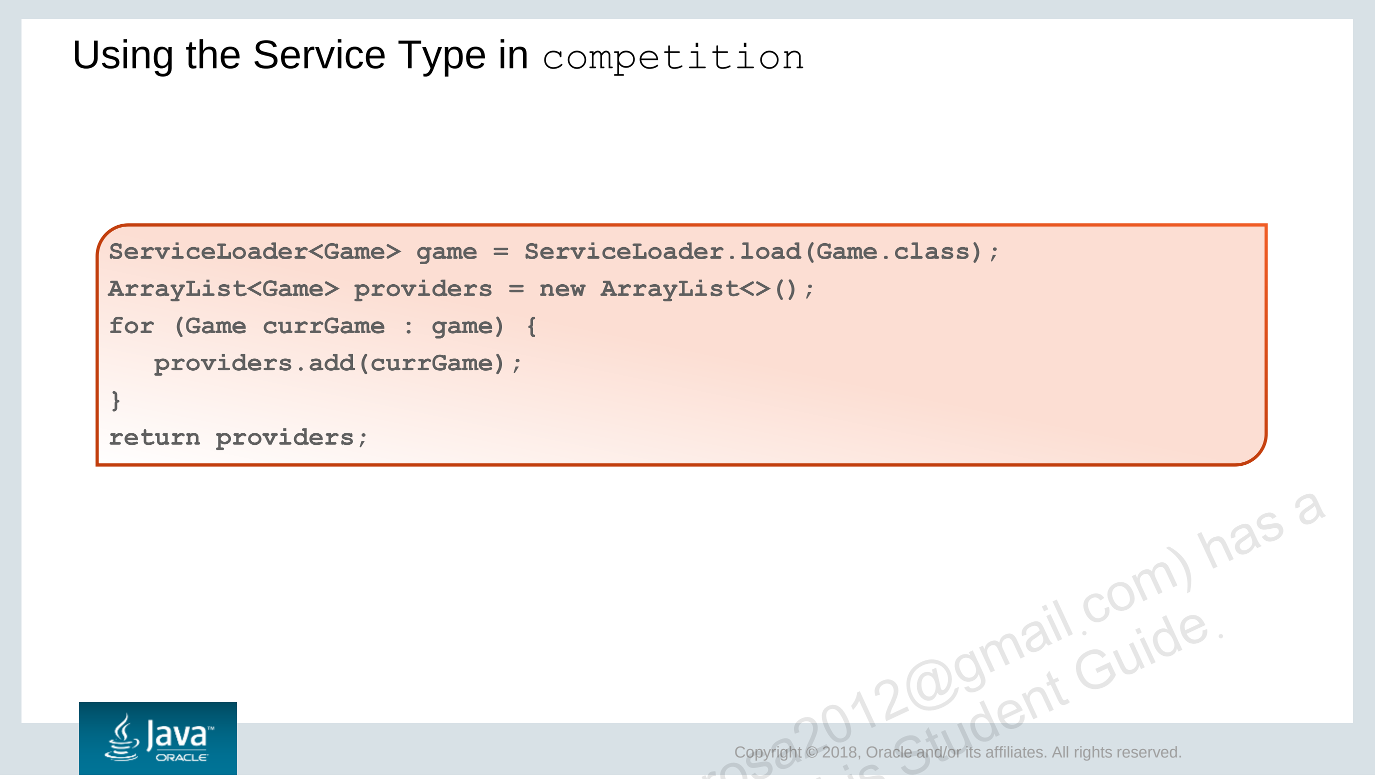
```
module soccer {
    provides gameapi.Game with soccer.Soccer;
}
```

A module requesting the implementation of an interface uses the `uses` directive in its `module-info.java` file to register this. This module is called the consumer module. A module implementing this interface uses the `provides` directive to declare this.



Topics

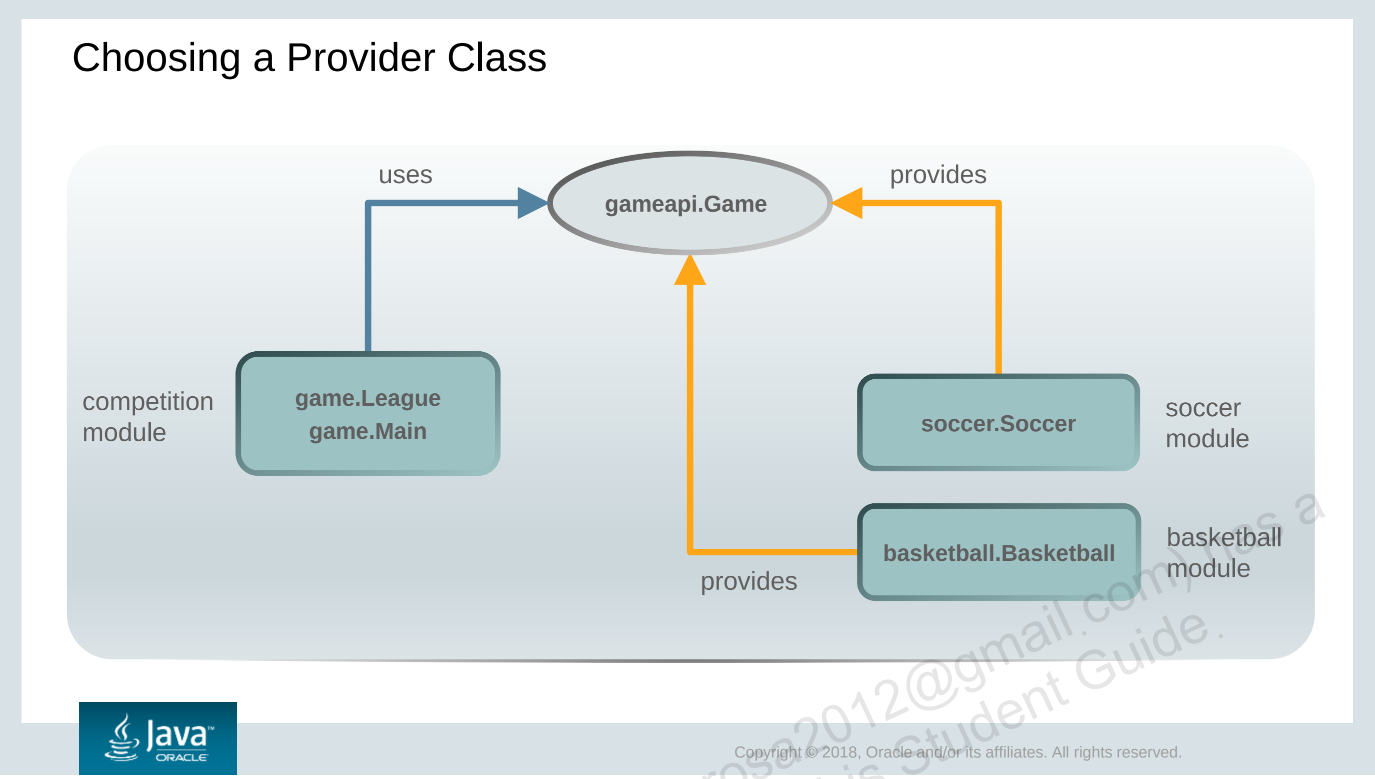
- Service-based design
- `ServiceLoader` class
- TeamGameManager example



Using the Service Type in competition

```
ServiceLoader<Game> game = ServiceLoader.load(Game.class);
ArrayList<Game> providers = new ArrayList<>();
for (Game currGame : game) {
   providers.add(currGame);
}
return providers;
```

The slide shows the basic code for getting the providers (the implementations).
The `ServiceLoader` class is used to get the providers. Its key method is static and called `load`. This method creates a new `ServiceLoader` for a given service type. `ServiceLoader` implements `Iterable`, and so you can use the `for` clause as shown to load and instantiate each of the available providers.



Choosing a Provider Class

uses

gameapi.Game

provides

competition module

game.League
game.Main

soccer.Soccer

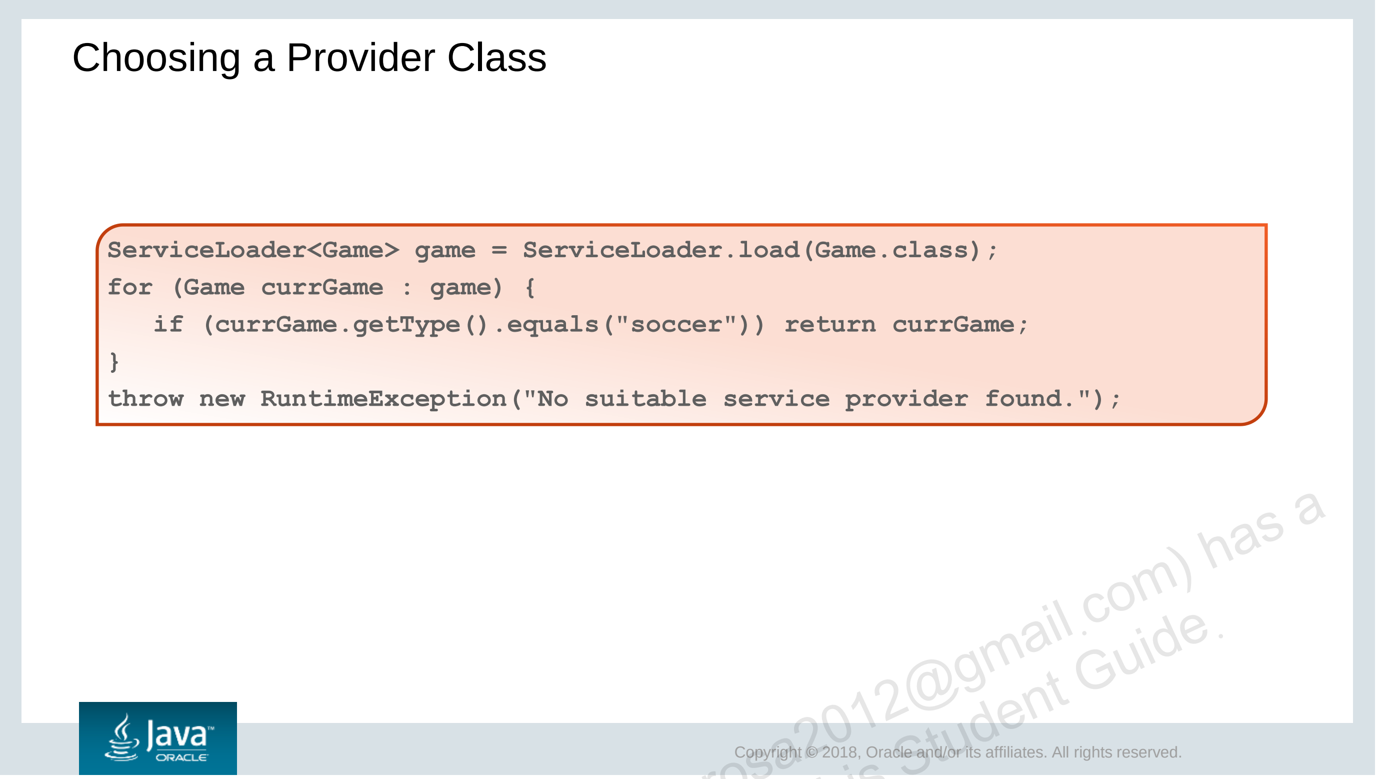
soccer module

basketball.Basketball

basketball module

provides

The module system finds all the implementations of the interface in the module path and, using the `ServiceLoader` class, makes them available to the `consumer` module. The provider modules do not know of each other, and the selection of which implementation is the best one given the need must be determined by the `consumer` module. But the design of the implementation may help with this. For example, the Game interface may declare a `getType` method that must be implemented by the `Soccer`, `Basketball`, or other class that implements it. This `getType` method can then be used to determine if the providers support the type of game desired.



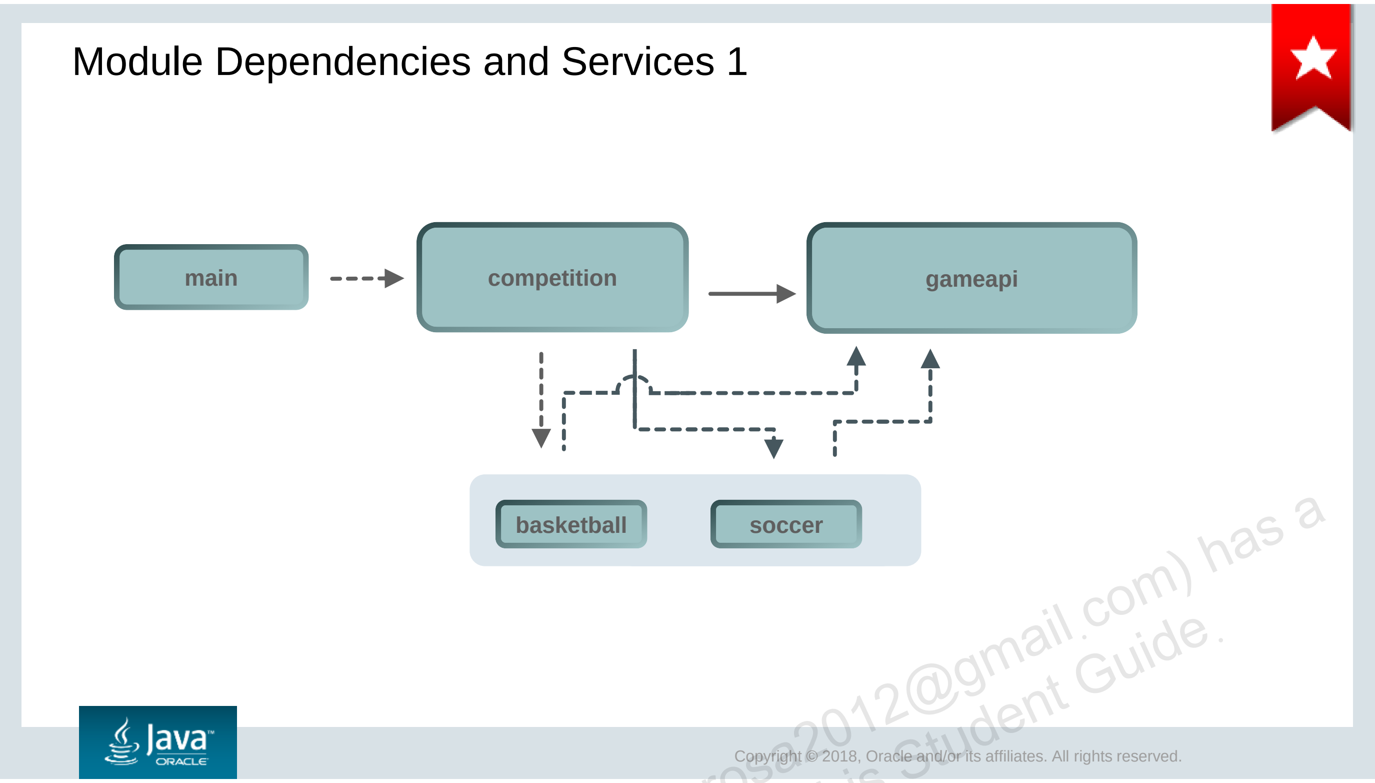
Choosing a Provider Class

```
ServiceLoader<Game> game = ServiceLoader.load(Game.class);
for (Game currGame : game) {
   if (currGame.getType().equals("soccer")) return currGame;
}
throw new RuntimeException("No suitable service provider found.");
```

The code in the slide uses the `getType` method to return a provider that identifies itself as “soccer.” More than one provider could do this, so there could be multiple soccer providers, in which case if the consumer does not simply wish to consume the first one returned, it must specify and test other criteria to make the decision.

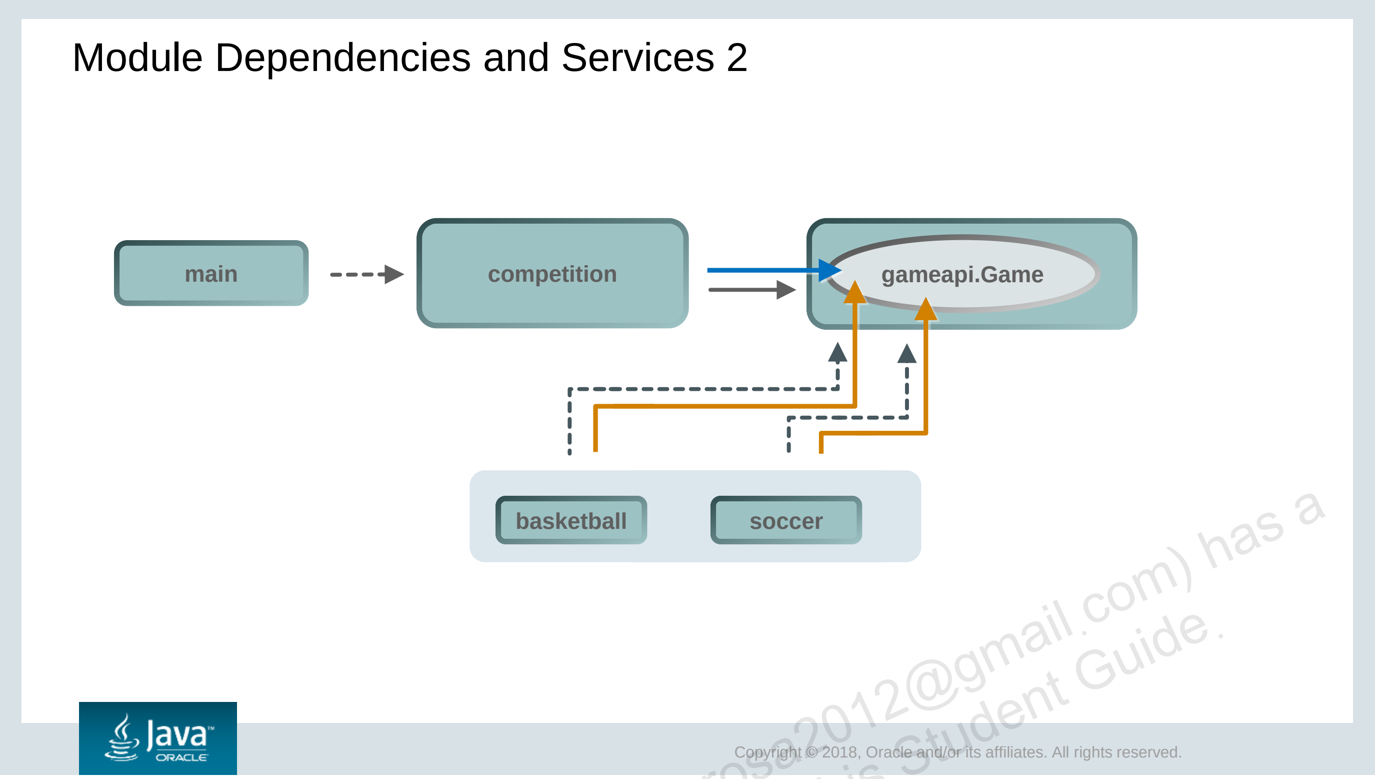
Note that the Game interface will have specified the `getType` method for this purpose, and any implementing class must implement it.

In the example in the slide, if the `for` loop completes without a provider being found, a `RuntimeException` will be thrown. The decision to do this is again entirely up to the consumer. In some cases, a default provider may be loaded at this point, or in other cases, services might be used to model optional dependencies, so no provider would need to be loaded.



Module Dependencies and Services 1

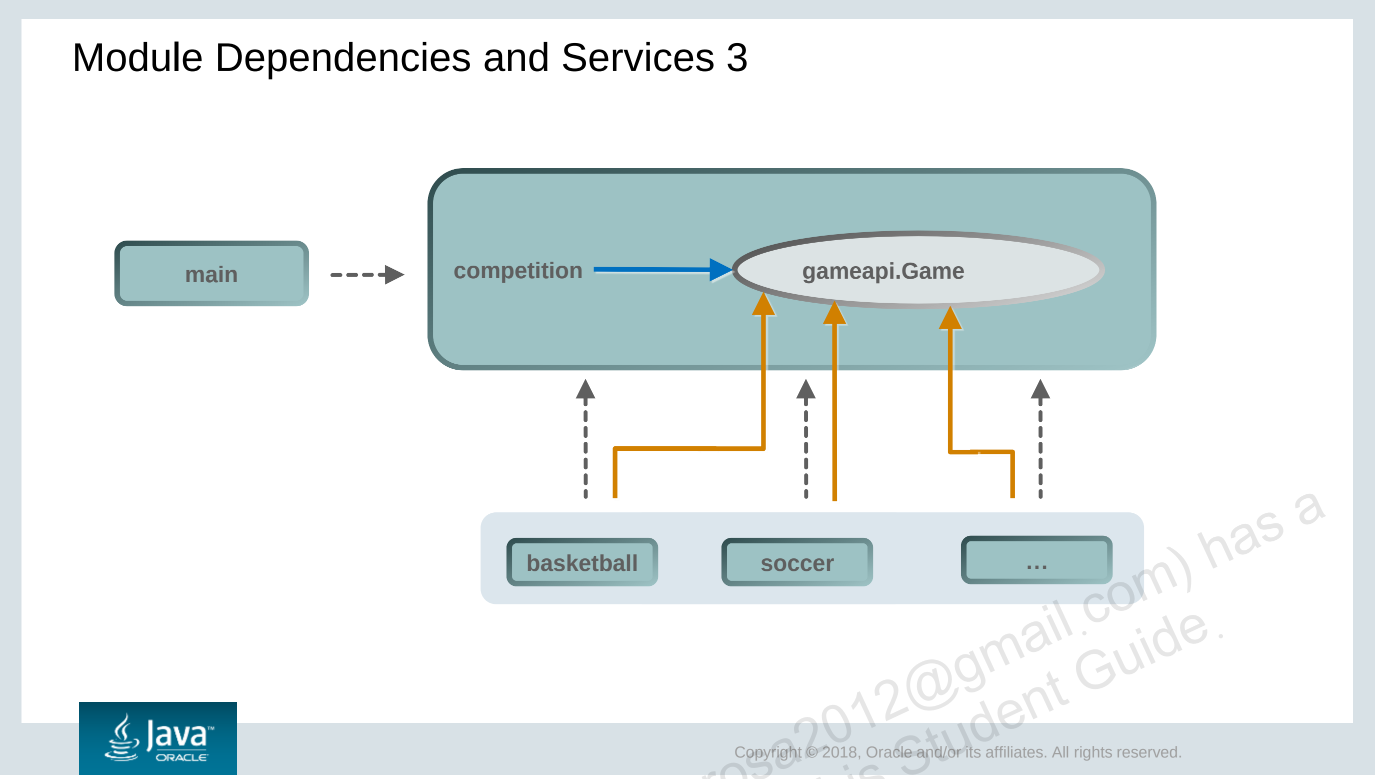
This slide and the following two slides show how services can replace hard-coded dependencies, thus changing the architecture from tightly coupled to loosely coupled. The graphic on this slide shows the application using hard-coded dependencies where the interfaces implemented in the soccer and basketball modules must be in a module other than the competition module so that a cyclic dependency does not occur.



Module Dependencies and Services 2

The graphic in the slide shows basketball and soccer functionality being provided as services. The following are some important points to note:

- The `competition` module does not declare a dependency on any provider, only on the module that contains the Game interface for which it requests an implementation.
- Each provider does have a dependency on the module that contains the Game interface because that is the interface it implements.



Module Dependencies and Services 3

main

competition

gameapi.Game

basketball

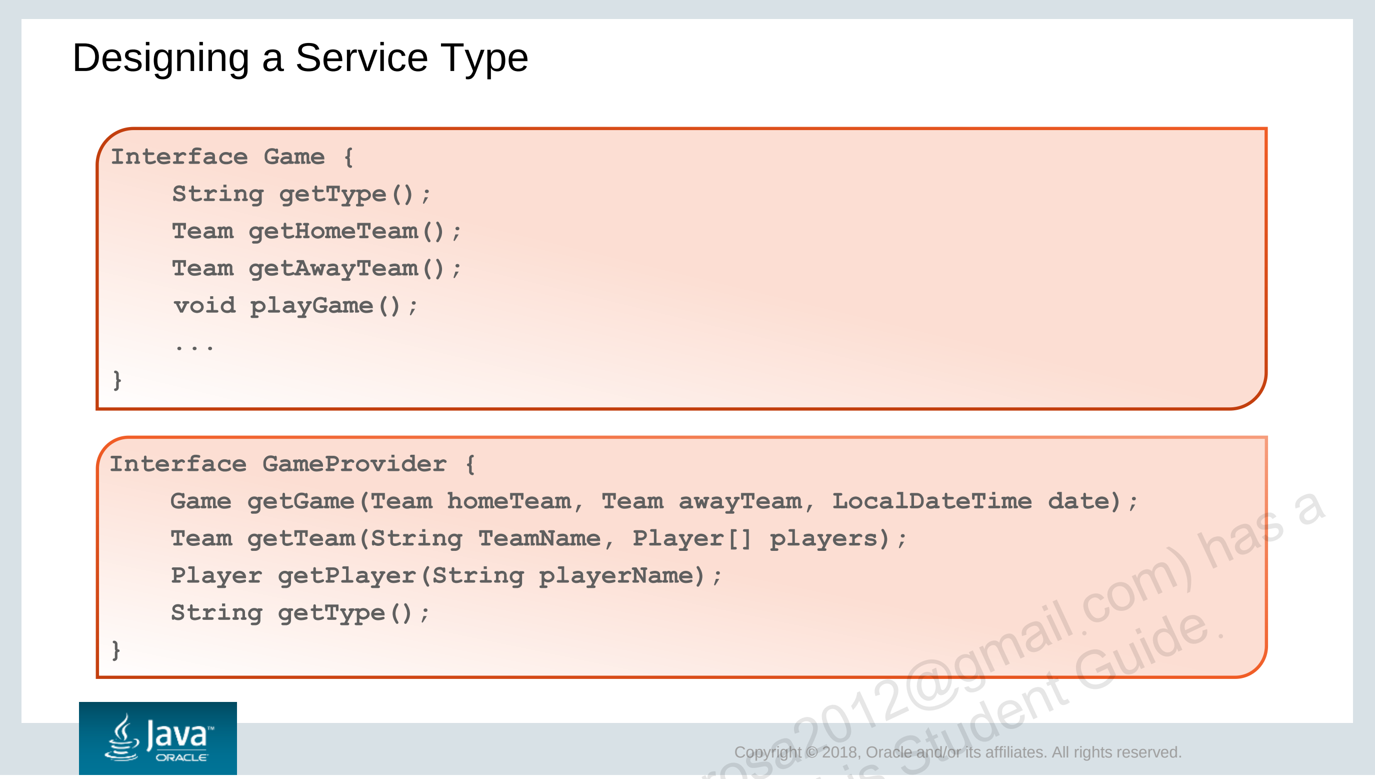
soccer

...

The graphic in this slide shows how using services can help to address cyclic dependencies.

Notice that the `Game` interface has been moved back into the competition. A cyclic dependency is when module `A` requires module `B` and vice versa. It is not a cyclic dependency if module `A` uses an implementation that lives in module `B` while module `B` requires module `A`. Therefore, the interfaces can be in any module, and in this simple example application, it makes sense to put the Game interface in the competition module.

Services are easily extensible as more provider modules can be added at any time, and any or all service modules can be removed without `competition` being modified.



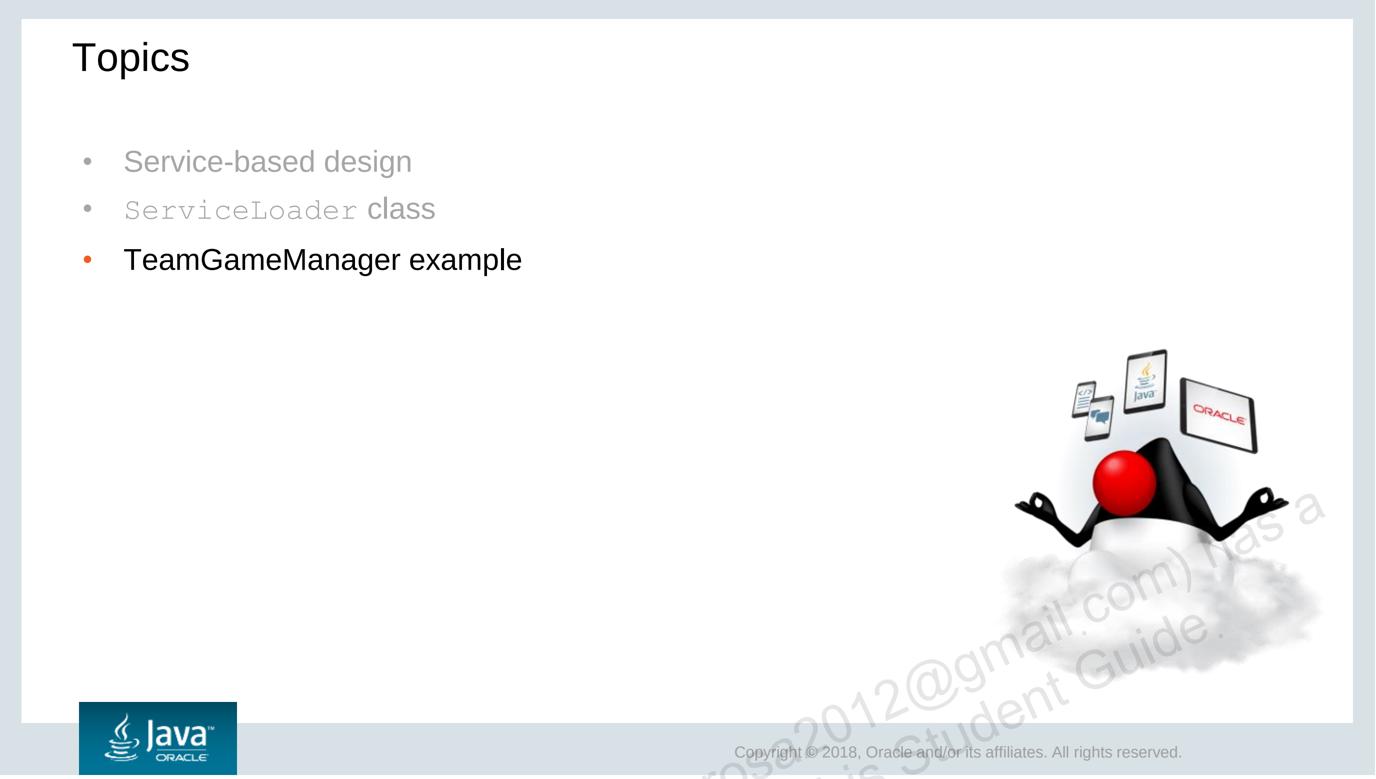
Designing a Service Type

```
Interface Game {
    String getType();
    Team getHomeTeam();
    Team getAwayTeam();
    void playGame();
    ...
}
```

```
Interface GameProvider {
    Game getGame(Team homeTeam, Team awayTeam, LocalDateTime date);
    Team getTeam(String TeamName, Player[] players);
    Player getPlayer(String playerName);
    String getType();
}
```

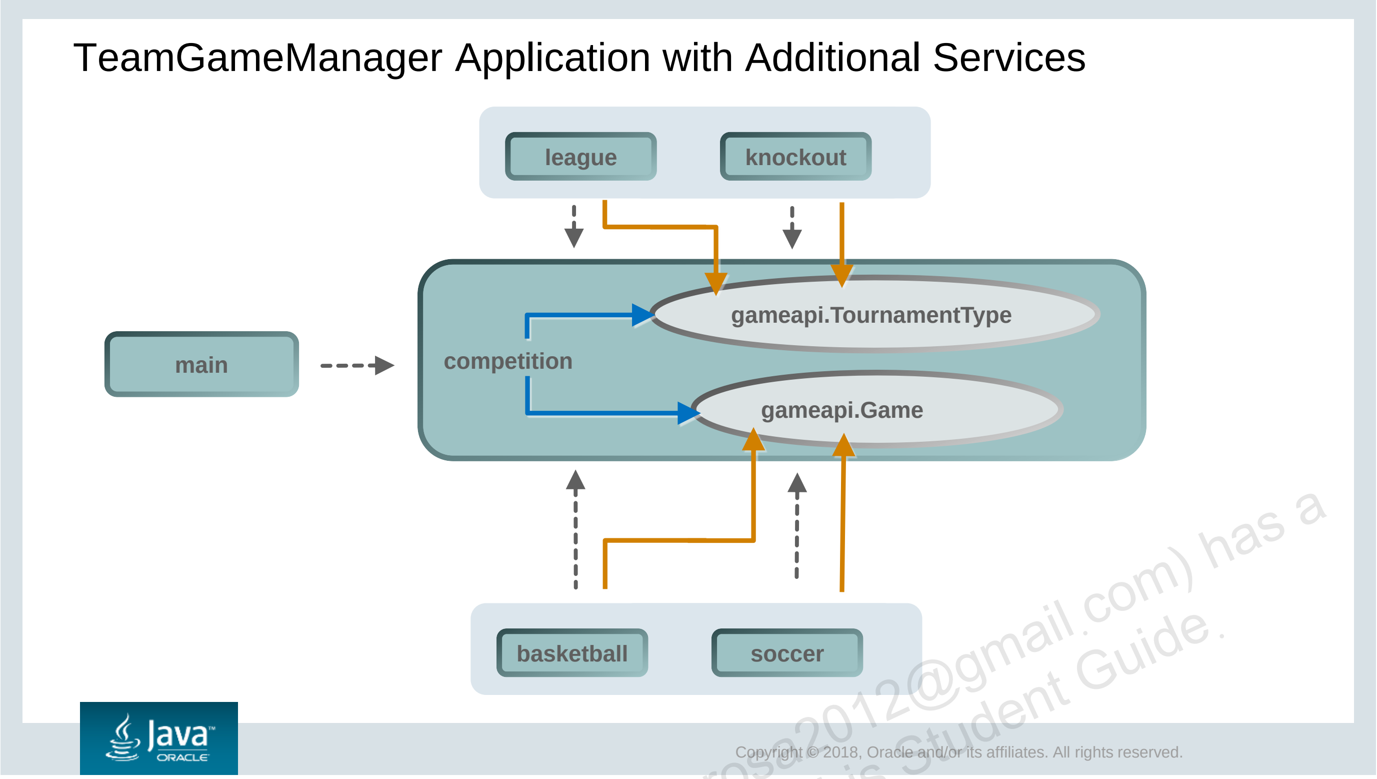
The service type itself can either return the concrete class to be used in the code or a proxy that will itself return a concrete class. In the examples shown in the slide, the implementation of the Game interface might be the actual class to be used in the code of the competition module. However, the Game implementation needs a Team type, and the Team implementation in turn needs a Player type, so implementations of all three types, Game, Team, and Player, would need to be available as providers. Also, each implementation class would need a no-argument constructor and probably also another method to populate the data in the class, given that the constructor is no-argument.

Another approach is to use a proxy provider. In this case, the proxy is not the actual class to be used in the competition module. Instead, it provides the classes that are to be used. This means that only one provider is required for the three classes that make up a type of game (Game, Team, Player), and the implementations of these three classes do not need a no-argument constructor. Instead, the necessary parameters can be passed directly to the constructor.



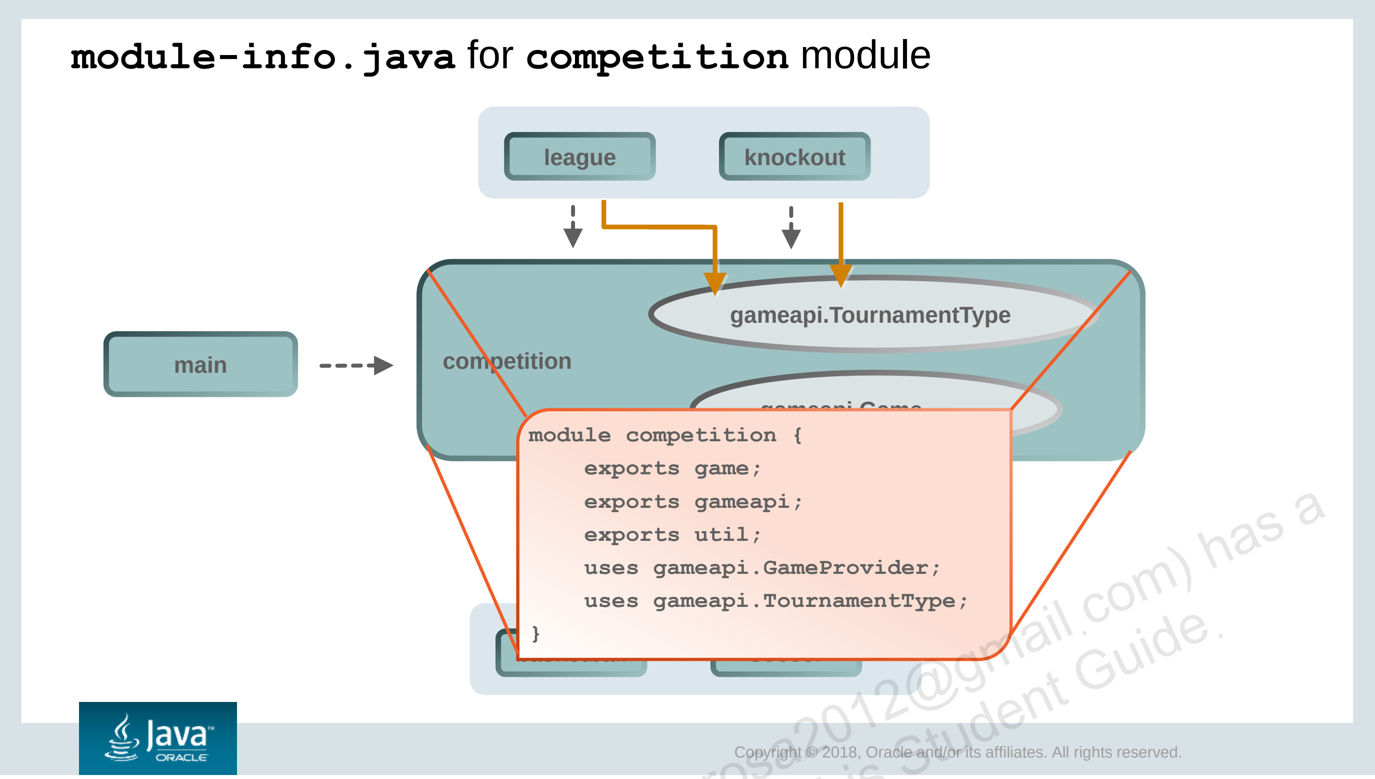
Topics

- Service-based design
- `ServiceLoader` class
- TeamGameManager example



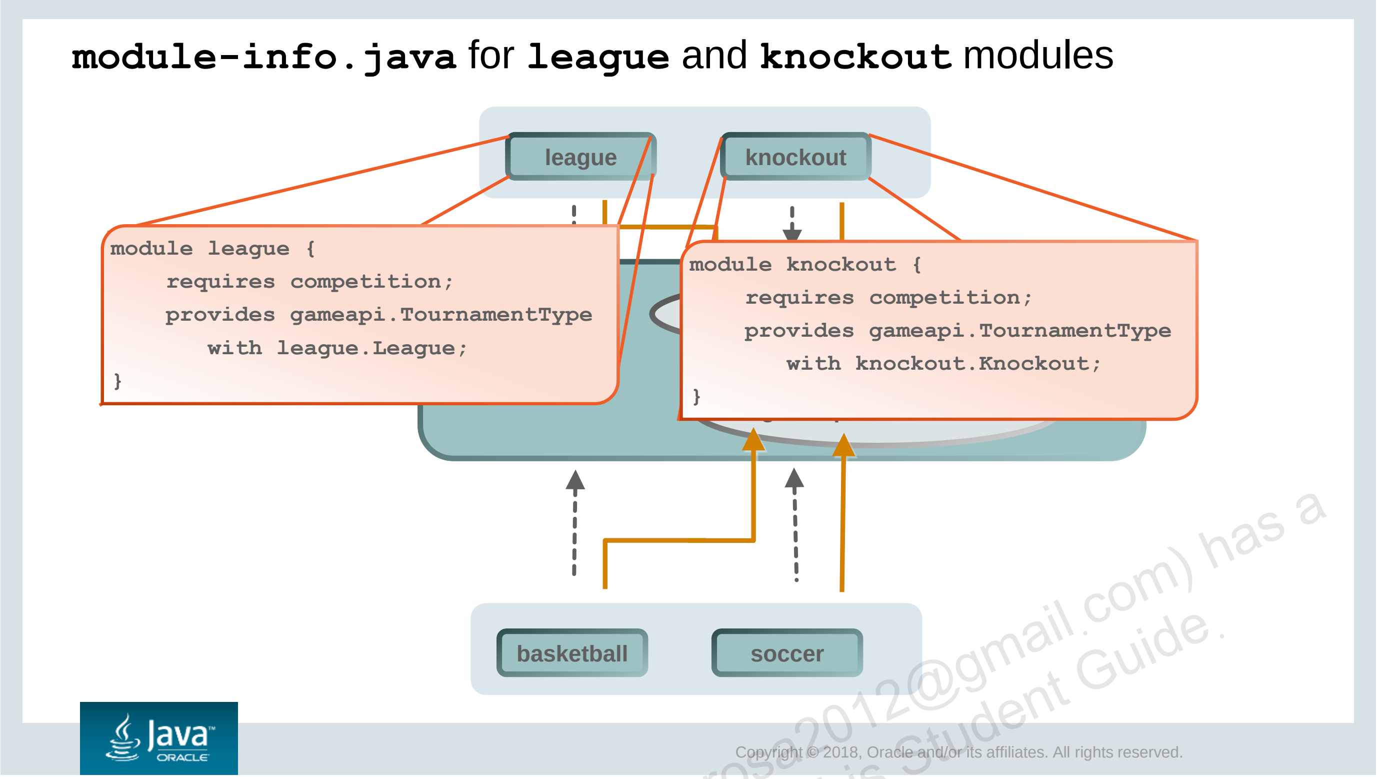
TeamGameManager Application with Additional Services

The slide shows a dependency diagram for the TeamGameManager application. The application has also been extended so that the type of competition is now also provided as a service. The two examples in the slide are league and knockout, but there are many other types possible.



module-info.java for **competition** module

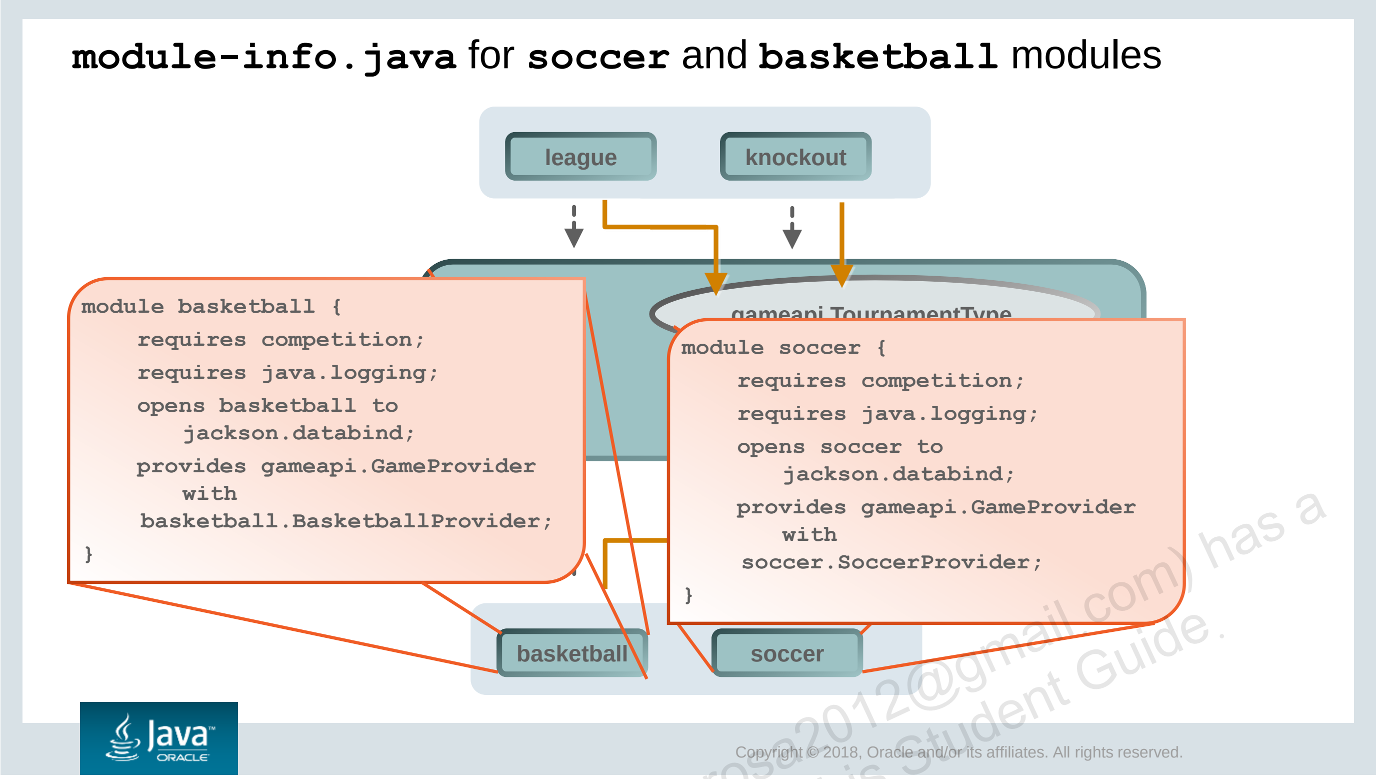
```
module competition {
    exports game;
    exports gameapi;
    exports util;
    uses gameapi.GameProvider;
    uses gameapi.TournamentType;
}
```



module-info.java for league and knockout modules

```
module league {
    requires competition;
    provides gameapi.TournamentType
        with league.League;
}
```

```
module knockout {
    requires competition;
    provides gameapi.TournamentType
        with knockout.Knockout;
}
```



`module-info.java` for `soccer` and `basketball` modules

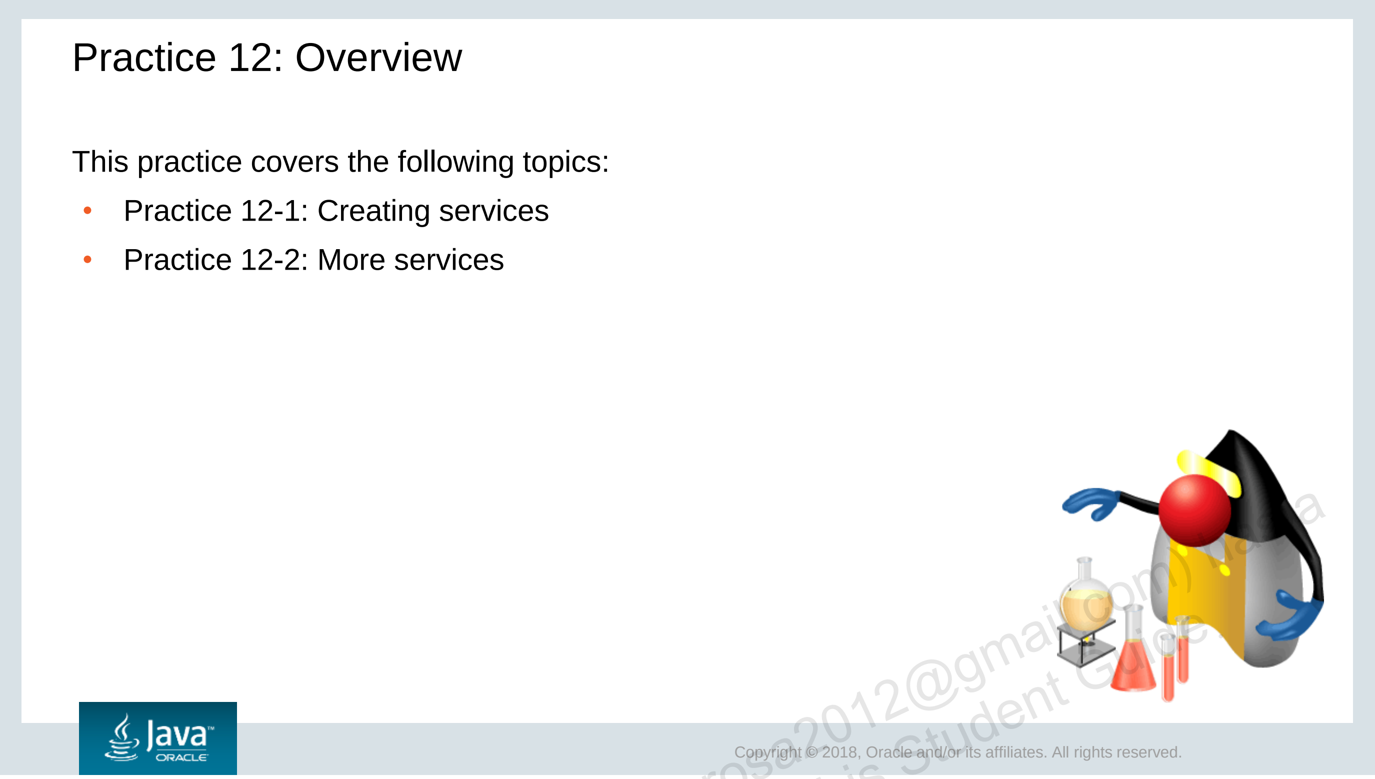
```
module basketball {
    requires competition;
    requires java.logging;
    opens basketball to
        jackson.databind;
    provides gameapi.GameProvider
        with
     basketball.BasketballProvider;
}
```

```
module soccer {
    requires competition;
    requires java.logging;
    opens soccer to
        jackson.databind;
    provides gameapi.GameProvider
        with
     soccer.SoccerProvider;
}
```

Summary

In this lesson, you should have learned to describe:

- How services are supported in Java SE 9
- The distinction between a service supplying a concrete object versus a proxy object
- How services can help address cyclic dependencies



Practice 12: Overview

This practice covers the following topics:

- Practice 12-1: Creating services
- Practice 12-2: More services

Java™
ORACLE®