Use placeholders.

_ in place of a type makes an existential type just as in place of expression makes a function literal. Each _ is a new existential type.

```
Iterator[_ <: Component]
Iterator[T] for Some {type T <: Component}
```

It must be done using Java notation and compiled with `javac`.

By using existential types, a fully supported part of the language in practice used only for Java compatibility.

```
Iterator[T] for Some {type T}
Iterator[T] for Some {type T <: Component}
```

Scala can check program soundness even though types and values in forSome clause are unknown.

e.g., `val contents = (new JavaClass).contents` is fine even if `JavaClass.contents` is `Collection<?>`.

A combination of static and instance methods.

They use a class called `ObjectName$` with one instance stored as `MODULE$` field.

If it's standalone, it will create `ObjectName` class with static forwarder methods for each singleton method.

It forwards them on to Java bytecode.

The Java bytecode verifier does not check the declarations, only `javac` does in source code.

It indicates the presence of a static initializer block.

- When passing existential type to a method, move type parameters from forSome clause to type parameter of the method. Then in the method body it has a name.
- Instead of returning existential type from method, return an object that has abstract members for each part of forSome clause.

Only when `@throws` annotation is used for Java compatibility.

```
@throws(classOf[IOException])
```

It's generally not practical.

If the scala trait has only abstract methods, however, it will be translated directly to a Java interface and can be implemented in Java.

There is no way to name the existential type.

e.g., how would one parameterize `Set.empty[??]` if you had taken advantage of Scala's ability to ignore most existential types from Java?

```
type forSome {declarations}
```
where type is some arbitrary Scala type and declarations is a list of abstract vals and types.