ObjectTeams/Java Language Definition

– version 1.2 built on 2009-06-23 –

Stephan Herrmann, Christine Hundt, Marco Mosconi

<www.objectteams.org>

Contents

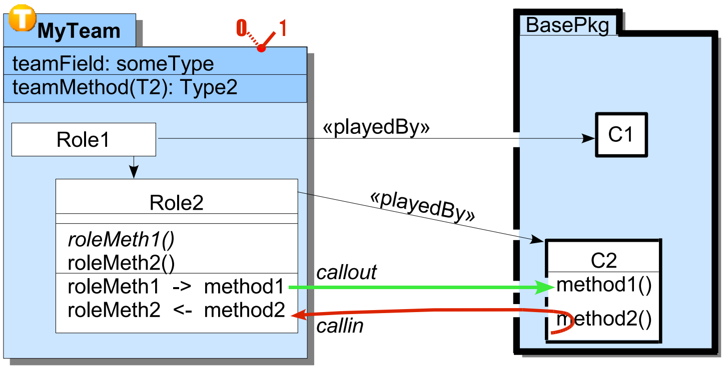


Figure 1: Essential concepts of ObjectTeams/Java

Introduction

ObjectTeams/Java (OT/J) is an extension to the Java programming language, realizing the concepts of the programming model *Object Teams*. OT/J introduces aspect-oriented and collaboration/role-based concepts which are smoothly integrated with the object-oriented concepts like inheritance and polymorphism.

Object Teams promotes the notion of collaborations as modules for interacting roles. It does so by introducing two new kinds of modules: *teams* (§ 1.1.) as higher-order modules for contained *roles* (§ 1.2.) (see `MyTeam` and its roles `Role1` and `Role2` in Fig. 1).

A `playedBy` relationship (§ 2.1.) binds a role class to a base class, which will be reflected by run-time links between pairs of role and base instances. The main purpose of creating a role-to-base connection is to create a channel for specific communication, which is established by two kinds of method bindings.

A `callin` method binding (§ 4.) intercepts the control flow at a method of the base entity and redirects it to a role method. Looking at Fig. 1, calls to `method2` will be intercepted, causing an invocation of `roleMeth2`. In order to ensure that the effect is purely additive, one of the modifiers `before` or `after` can be used. When specifying a `replace` callin binding, this has approximately the same effect as overriding a method in the context of inheritance. Using the concept of callin bindings, roles unify instance-based inheritance with method call interception as it is used in aspect-oriented programming.

Conversely, a `callout` method binding (§ 3.) simply forwards a method (`roleMeth1`) from a role instance to its *player*, the base instance (`method1`). Callout bindings per se are rather unspectacular, but a callout binding may support *decapsulation*, a term we coined for the inverse of encapsulation. This means that a callout binding can – within limits – access protected or even private members of the corresponding base class (note the “holes” in the border of the base package and its class C2). Decapsulation is also supported for the `playedBy` clause at class level, so that a role may be attached to an otherwise inaccessible class (like a package-visible class or a private inner class).

Several means exist to control when a callin binding is actually effective, i.e, whether or not the base control flow should be intercepted. The most elegant technique is to *activate* or deactivate a given team instance (see § 5.), which has the effect that all callin bindings of all contained roles are enabled or disabled in one step. In Fig. 1, activation is symbolized by the switch at the top of `MyTeam`.

▷ About this Document § 0.

Levels of this document

Terms, concepts Each chapter of this document starts with a short synopsis of concepts covered by the chapter (like this).

Definition The actual definition is given in small numbered paragraphs.

Examples Examples and accompanying explanations will be interspersed into the definition.

▷ Purpose(s) of this document § 0.1.

This document defines the ObjectTeams/Java programming language (OT/J). The main goals were to create a precise and complete reference for this language. Didactical considerations had lower priorities, which means that this document is not designed as an introductory tutorial. Still, we advise programmers learning the OT/J language, to consult this document whenever a compiler error message is not perfectly clear to them (see § 0.3.).

▷ Text structure § 0.2.

Each chapter of this document starts with a short synopsis of concepts covered by the chapter (see above).

→ Syntax § A

(a) Paragraphs

The actual definition is structured into small paragraphs for easy referral.

(b) Syntax Links

Links to the syntax precede definitions whenever new syntax is introduced.

Interspersed you will find some example program listings. Examples are typeset in a box:

```
1 public team class MyTeamA {
2    ...
3 }
```

Explanations for examples look like the following:

> **Effects:**
>
> - Lines 1-3 show a minimal OT/J program, which should not cause any headache.

Examples are given for illustration only.

Additional paragraphs like "Language implementation", or "open issues" provide some background information which is not necessary for understanding the definition, which might however, help to understand why things are the way they are and what other things might be added to the language in the future.

§ 0.3. Compiler messages

Error messages given by the Object Teams compiler refer to this definition whenever appropriate. This way it should be easy to find out, why the compiler rejected your program. Please make sure you are using a language definition whose version matches the version of your compiler.

§ 0.4. Versions

The structure of this document has changed between versions 0.6.1 and 0.7 of this document. This change reflects the transition from our first compiler for OT/J (called `otc`) and the `OTDT` (Object Teams Development Tooling) plugin for Eclipse.

Starting with the OTDT v0.7.x, the major and minor number of the tool correspond to the major and minor version number of the OTJLD (this document), ie., the OTDT v1.0.x implements the language as defined in the OTJLD v1.0.

Changes between the current and previous versions are listed in appendix § B..

§ 0.5. Publishing

The sources of this language definition are maintained in a target-independent XML format. Three different versions are generated from these sources, using XSLT:

- Online version[1] (XHTML)
- Tooling version (XHTML – directly accessible from inside the OTDT)
- Print version (LaTeX/PDF)

[1]`http://www.objectteams.org/def/`

▷ Teams and Roles § 1.

Fundamental concepts of Teams

Teams and Roles Classes that are defined with the modifier `team` are called team classes, or **teams** for short.
Direct inner classes of a team are called role classes, or **roles** for short.

Role inheritance Inheritance between teams introduces a special inheritance relationship between their contained roles. The rules of this **implicit inheritance** are given below (§ 1.3.1.).

Externalized role Roles are generally confined to the context of their enclosing team instance. Subject to specific restrictions, a role *may* be passed outside its team using the concept of externalized roles (§ 1.2.2.).

▷ Team classes § 1.1.

→ Syntax § A.1.1

A class declared with the modifier `team` is a *team class* (or team for short).

```
public team class MyTeamA {
  ...
}
```

Teams are meant as containers for *roles*, which are defined in the following paragraphs.

```
public team class MyTeamA {
  public class MyRole
    ...
  }
}
```

Teams introduce a new variant of inheritance for contained role classes (see § 1.3.1. below). Other properties of teams, which are defined in later sections, are:

- Team activation (§ 5.)
- Abstractness and instantiation (§ 2.5.)
- Declared lifting in team methods (§ 2.3.2.)
- Reflective functions defined in `org.objectteams.Team` (§ 6.1.)

Apart from these differences, team classes are regular Java classes with methods and fields, whose instances are regular Java objects.

▷ Role classes and objects § 1.2.

Each direct inner class of a team is a role class. Just like inner classes, each instance of a role class has an implicit reference to its enclosing team instance. This reference is immutable.

Within the implementation of a role it can be accessed by qualifying the identifier `this` with the name of the team class, as in:

```
public team class MyTeamA {
  public class MyRole {
    public void print() { System.out.println("Team: "+ MyTeamA.this); }
  }
}
```

Creation of role instances is further restricted as defined in § 2.4.. Teams can also define role interfaces just like role classes. With respect to role specific properties a role interface is treated like a fully abstract class.

§ 1.2.1. Modifiers for roles

Member classes of a team cannot be `static`. Also the use of access modifiers for roles is restricted and modifiers have different (stronger) semantics than for regular classes (see below). With respect to accessibility a team acts mainly like a package regarding its roles.

(a) Role class protection

A role class must have exactly one of the access modifiers `public` or `protected`. This rule does not affect the class modifiers `abstract`, `final` and `strictfp`.

(b) protected role classes

A `protected` role can only be accessed from within the enclosing team or any of its sub-teams. The actual border of encapsulation is the enclosing team *instance*. The rules for protected roles are given in § 1.2.3. below.

(c) public role classes

Only `public` roles can ever be accessed outside their enclosing team. Accessing a role outside the enclosing team instance is governed by the rules of **externalized roles**, to be defined next (§ 1.2.2.).

(d) abstract role classes

A role class has to be marked **abstract** if any of its methods is not effective.
The *methods of a role class* comprise direct methods and methods acquired by inheritance. In addition to regular inheritance a role class may acquire methods also via implicit inheritance (§ 1.3.1.).
A method may become *effective* by either:

- implementation (i.e., a regular method body), or
- a callout binding (see § 3.).

§ 2.5. discusses under which circumstances abstract roles force the enclosing team to be abstract, too.

(e) Role features

Access modifiers for members of roles have some special interpretation:

1. A private member is also visible in any implicit sub role (see implicit inheritance § 1.3.1.(c)).
   In contrast to inner classes in Java, private members of a role are not visible to the enclosing team.
2. The default visibility of role members restricts access to the current class and its sub-classes (explicit and implicit).
3. `protected` role members can only be accessed from the enclosing team or via callin (§ 4.).
4. `public` role members grant unrestricted access.

Additionally, a role always has access to all the features that its enclosing team has access to.
Only `public` members can ever be accessed via an externalized role (§ 1.2.2.).

(f) Static role methods

In contrast to inner classes in pure Java, a role class may indeed define static methods. A static role method requires no role instance *but* it still requires a team instance in scope. Static role methods can be called:

- from the enclosing team,
- via callin (see § 4.7.).

Within a static role method the syntax `MyTeam.this` is available for accessing the enclosing team instance.

(g) No static initializers

A static field of a role class must not have a non-constant initialization expression. Static initialization blocks are already prohibited for inner classes by Java (see JLS §8.1.2[2]).

<u>*Note:*</u>
Static initialization generally provides a means for performing initialization code prior to instantiation, i.e., at class-loading time. Before any role can be created already two levels of initialization are performed: (1) The (outer most) enclosing team class performs static initializations when it is loaded. (2) Any enclosing team executes its constructor when it is instantiated. It should be possible to allocate any early initialization to either of these two phases instead of using static role initializers.

▷ Externalized roles § 1.2.2.

→ Syntax § A.9.2

Normally, a team encapsulates its role against unwanted access from the outside. If roles are visible outside their enclosing team instance we speak of **externalized roles**.

Externalized roles are subject to specific typing rules in order to ensure, that role instances from different team instances cannot be mixed in inconsistent ways. In the presence of implicit inheritance (§ 1.3.1.) inconsistencies could otherwise occur, which lead to typing errors that could only be detected at run-time. Externalized roles use the theory of "virtual classes"

[2] `http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#262890`

[1] (see p. 18), or more specifically "family polymorphism" [2] (see p. 18), in order to achieve the desired type safety. These theories use special forms of *dependent types*. Externalized roles have *types that depend on a team instance*.

§ 1.2.3. deduces even stronger forms of encapsulation from the rules about externalized roles.

(a) Visibility

Only instances of a `public` role class can ever be externalized.

(b) Declaration with anchored type

Outside a team role types are legal only if denoted relative to an existing team instance (further on called "anchored types"). The syntax is:

```
final MyTeam myTeam = expression;
RoleClass<@myTeam> role = expression;
```

The syntax `Type<@anchor>` is a special case of a parameterized type, more specifically a value dependent type (§ 9.). The type argument (i.e., the expression after the at-sign) can be a simple name or a path. It must refer to an instance of a team class. The role type is said to be *anchored* to this team instance.
The type-part of this syntax (in front of the angle brackets) must be the simple name of a role type directly contained in the given team (including roles that are acquired by implicit inheritance).

<u>Note:</u>
Previous versions of the OTJLD used a different syntax for anchored types, where the role type was prefixed with the anchor expression, separated by a dot (`anchor.Type`, see § A.6.). A compiler may still support that path syntax but it should be flagged as being deprecated.

(c) Immutable anchor

Anchoring the type of an externalized role to a team instance requires the team to be referenced by a variable which is marked `final` (i.e., immutable). The type anchor can be a path `v.f1.f2...` where `v` is any final variable and `f1` ... are final fields.

(d) Implicit type anchors

The current team instance can be used as a default anchor for role types:

1. In non-static team level methods role types are by default interpreted as anchored to `this` (referring to the team instance). I.e., the following two declarations express the same:

```
public RoleX getRoleX (RoleY r) { stmts }
public RoleX<@this> getRoleX (RoleY<@this> r) { stmts }
```

2. In analogy, *role methods* use the enclosing team instance as the default anchor for any role types.

Note, that `this` and `Outer.this` are always `final`.
The compiler uses the pseudo identifier **`tthis`** to denote such implicit type anchors in error messages.

(e) Conformance

Conformance between two types `RoleX<@teamA>` and `RoleY<@teamB>` not only requires the role types to be compatible, but also the team instances to be provably *the same object.* The compiler must be able to statically analyze anchor identity.

(f) Substitutions for type anchors

Only two substitutions are considered for determining team identity:

1. For type checking the application of team methods, `this` is **substituted** by the actual call target. For role methods a reference of the form *`Outer`*`.this` is substituted by the enclosing instance of the call target.
2. Assignments from a `final` identifier to another `final` identifier are transitively followed, i.e., if `t1, t2` are final, after an assignment `t1=t2` the types `R<@t1>` and `R<@t2>` are considered identical. Otherwise `R<@t1>` and `R<@t2>` are incommensurable.
   Attaching an actual parameter to a formal parameter in a method call is also considered as an assignment with respect to this rule.

(g) Legal contexts

Anchored types for externalized roles may be used in the following contexts:

1. Declaration of an attribute
2. Declaration of a local variable
3. Declaration of a parameter or result type of a method or constructor
4. In the `playedBy` clause of a role class (see § 2.1.).

It is not legal to inherit from an anchored type, since this would require membership of the referenced team instance, which can only be achieved by class nesting.

Note:
Item 4. — within the given restriction — admits the case where the same class is a role of one team and the base class for the role of another team. Another form of nesting is defined in § 1.5..

(h) Externalized creation

A role can be created as externalized using either of these equivalent forms:

```
outer.new Role()
new Role<@outer>()
```

This requires the enclosing instance `outer` to be declared `final`. The expression has the type `Role<@outer>` following the rules of externalized roles.
The type `Role` in this expression must be a simple (unqualified) name.

(i) No import

It is neither useful nor legal to import a role type.

Rationale:
Importing a type allows to use the unqualified name in situations that would

otherwise require to use the fully qualified name, i.e., the type prefixed with its containing package and enclosing class. Roles, however are contained in a team instance. Outside their team, role types can only be accessed using an anchored type which uses a team instance to qualify the role type. Relative to this team anchor, roles are always denoted using their simple name, which makes importing roles useless.

A static import for a constant declared in a role is, however, legal.

Listing 1: Example code (Externalized Roles):

```
1 team class FlightBonus extends Bonus {
2   public class Subscriber {
3     void clearCredits() { ... }
4   }
5   void unsubscribe(Subscriber subscr) { ... }
6 }
```

```
7  class ClearAction extends Action {
8    final FlightBonus context;
9    Subscriber<@context> subscriber;
10   ClearAction (final FlightBonus bonus, Subscriber<@bonus> subscr) {
11     context = bonus; // unique assignment to 'context'
12     subscriber = subscr;
13   }
14   void actionPerformed () {
15     subscriber.clearCredits();
16   }
17   protected void finalize () {
18     context.unsubscribe(subscriber);
19   }
20 }
```

Effects:

- Lines 1-6 show a terse extract of a published example [NODe02][3]. Here passengers can be subscribers in a flight bonus program.
- Lines 7-20 show a sub-class of `Action` which is used to associate the action of resetting a subscriber's credits to a button or similar element in an application's GUI.
- Attribute `context` (line 8) and parameter `bonus` (line 10) serve as anchor for the type of externalized roles.
- Attribute `subscriber` (line 9) and parameter `subscr` (line 10) store a Subscriber role outside the FlightBonus team.
- In order to type-check the assignment in line 12, the compiler has to ensure that the types of LHS and RHS are anchored to the same team instance. This can be verified by checking that both anchors are indeed `final` and prior to the role assignment a team assignment has taken place (line 11).
  Note, that the Java rules for **definite assignments** to final variables ensure that exactly one assignment to a variable occurs prior to its use as type anchor. No further checks are needed.

- It is now legal to store this role reference and use it at some later point in time, e.g., for invoking method `clearCredits` (line 15). This method call is also an example for implicit team activation (§ 5.3.(b)).
- Line 18 demonstrates how an externalized role can be passed to a team level method. The signature of `unsubscribe` is for this call expanded to

  void unsubscribe(Subscriber<@context> subscr)

  (by substituting the call target `context` for `this`). This proves identical types for actual and formal parameters.

▷ Protected roles § 1.2.3.

Roles can only be `public` or `protected`. A `protected` role is encapsulated by its enclosing team instance. This is enforced by these rules:

(a) Importing role classes

This rule is superseded by § 1.2.2.(i)

(b) Qualified role types

The name of a `protected` role class may never be used qualified, neither prefixed by its *enclosing type* nor parameterized by a *variable as type anchor* (cf. § 1.2.2.(a)).

(c) Mixing qualified and unqualified types

An externalized role type is never compatible to an unqualified role type, except for the substitutions in § 1.2.2.(f), where an explicit anchor can be matched with the implicit anchor `this`.

Rules (a) and (b) ensure that the name of a protected role class cannot be used outside the lexical scope of its enclosing team. Rule (c) ensures that team methods containing unqualified role types in their signature cannot be invoked on a team other than the current team. Accordingly, for role methods the team context must be the enclosing team instance.

(d) Levels of encapsulation

Since protected role types can not be used for externalization, instances of these types are already quite effectively encapsulated by their enclosing team. Based on this concept, encapsulation for protected roles can be made even stricter by the rules of *role confinement*. On the contrary, even protected roles can be externalized as *opaque roles* which still expose (almost) no information. Confinement and opaque roles are subject of § 7..

▷ Type tests and casts § 1.2.4.

In accordance with § 1.2.2.(e), in ObjectTeams/Java the `instanceof` operator and type casts have extended semantics for roles.

(a) instanceof

For role types the `instanceof` operator yields true only if both components of the type match: the dynamic role type must be compatible to the given static type, and also type anchors must be the same instance.

(b) Casting

Casts may also fail if the casted expression is anchored to a different team instance than the cast type. Such failure is signaled by a `org.objectteams.RoleCastException`.

(c) Class literal

A class literal of form `R.class` is dynamically bound to the class `R` visible in the current instance context. Using a class literal for a role outside its enclosing team instance (see § 1.2.2.) requires the following syntax (i.e., the parametric syntax for externalized roles is not available here):
teamAnchor.RoleClass`.class`

§ 1.2.5. File structure

Just like regular inner classes, role classes may be inlined in the source code of the enclosing team. As an alternative style it is possible to store role classes in separate **role files** according to the following rules:

(a) Role directory

In the directory of the team class a new directory is created which has the same name as the team without the `.java` suffix.

(b) Role files

Role classes are stored in this directory (a). The file names are derived from the role class name extended by `.java`.
A role file must contain exactly one top-level type.

(c) package statement

A role class in a role file declares as its package the fully qualified name of the enclosing team class. The package statement of a role file must use the `team` modifier as its first token.

(d) Reference to role file

A team should mention in its javadoc comment each role class which is stored externally using a `@role` tag.

(e) Legal types in role files

The type in a role file must not be an `enum`.

(f) Imports in role files

A role file may have imports of its own. Within the role definition these imports are visible *in addition* to all imports of the enclosing team. Only base imports (see § 2.1.2.(d)) *must* be defined in the team.

Semantically, there is no difference between inlined role classes and those stored in separate role files.

<u>*Note:*</u>
Current Java compilers disallow a type to have the same fully qualified name as a package. However, the JLS does not seem to make a statement in this respect. In OT/J, a package and a type are interpreted as being the same team, if both have the same fully qualified name and both have the `team` modifier.

[in file `org/objectteams/examples/MyTeamA.java` :]

Listing 2: Role file example:

```
package org.objectteams.examples;
/**
 * @author Stephan Herrmann
 * @date 20.02.2007
 * @file MyTeamA.java
 * @role MyRole
 */
public team class MyTeamA {
   ...
}
```

[in file `org/objectteams/examples/MyTeamA/MyRole.java`:]

```
team package org.objectteams.examples.MyTeamA;
public class MyRole {
   ...
}
```

▷ Acquisition and implicit inheritance of role classes § 1.3.

The predefined class `org.objectteams.Team` is the root of all team classes. It is implicitly taken as the super-class of any team class that has no explicit `extends` clause. If an explicit `extends` clause is present in a team, the super-class must again be a team class. Conversely, any subclass of a team (including `org.objectteams.Team`) must again be a team. Interface implementation is not affected by this rule.

Infrastructure provided by class `org.objectteams.Team` is presented in § 6..

▷ Acquisition and implicit inheritance of role classes § 1.3.1.

A team acquires all roles from its super-team. This relation is similar to inheritance of inner classes, but with a few decisive differences as defined next. Two implementation options are mentioned below (see p. 17), which can be used to realize the special semantics of role acquisition (virtual classes and copy inheritance).

Listing 3: Implicit role inheritance

```
1 public team class S {
2     protected class R0 {...}
3     protected class R1 extends R0 {
4         boolean ok;
5         R2 m() {...}
6         void n(R2 r) {...}
7     }
8     protected class R2 {...}
9 }
```

```
10 public team class T extends S {
11     @Override protected class R1 {
12         R2 m() {
13             if(ok) { return tsuper.m(); }
14             else { return null; }
15         }
16         void doIt() {
17             n(m());
18         }
19     }
20 }
```

(a) Role class acquisition

A team `T` which extends a super-team `S` has one role class `T.R` corresponding to each role `S.R` of the super-team. The new type `T.R` **overrides** `R` for the context of `T` and its roles. Acquisition of role classes can either be direct (see (b) below), or it may involve overriding and implicit inheritance ((c) below).

In the above example (Listing 3) the team `S` operates on types `S.R0`, `S.R1` and `S.R2`, while `T` operates on types `T.R0`, `T.R1` and `T.R2`.
(Type references like "`S.R0`" are actually illegal in source code (§ 1.2.3.(b)). Here they are used for explanatory purposes only)

(b) Direct role acquisition

Within a sub-team `T` each role `S.R` of its super-team `S` is available by the simple name `R` without further declaration.

The role `R2` in Listing 3 can be used in the sub-team `T` (line 12), because this role type is defined in the super class of the enclosing team.

(c) Overriding and implicit inheritance

If a team contains a role class definition by the same name as a role defined in its super-team, the new role class overrides the corresponding role from the super-team and **implicitly inherits** all of its features. Such relation is established only by name correspondence.
A role that overrides an inherited role should be marked with an `@Override` annotation. A compiler should optionally flag a missing `@Override` annotation with a warning. Conversely, it is an error if a role is marked with an `@Override` annotation but does not actually override an inherited role.

It is an error to override a role class with an interface or vice versa. A final role cannot be overridden.
Unlike regular inheritance, **constructors** are also inherited along implicit inheritance, and can be overridden just like normal methods.

> In Listing 3 `R1` in `T` implicitly inherits all features of `R1` in `S`. This is, because its enclosing team `T` extends the team `S` (line 10) and the role definition uses the same name `R1` (line 11). Hence the attribute `ok` is available in the method `m()` in `T.R1` (line 13). `T.R1` also overrides `S.R1` which is marked by the `@Override` annotation in line 11.

(d) Lack of subtyping

Direct acquisition of roles from a super-team and implicit inheritance do not establish a **subtype** relation. A role of a given team is never conform (i.e., substitutable) to any role of any *other* team. `S.R` and `T.R` are always incommensurable.
Note, that this rule is a direct consequence of § 1.2.2.(e).

(e) Dynamic binding of types

Overriding an acquired role by a new role class has the following implication: If an expression or declaration, which is evaluated on behalf of an instance of team `T` or one of its contained roles, refers to a role `R`, `R` will always resolve to `T.R` even if `R` was introduced in a super-team of `T` and even if the specific line of code was inherited from a super-team or one of its roles. Only the dynamic type of the enclosing team-instance is used to determine the correct role class (see below for an example).
A special case of dynamically binding role types relates to so-called class literals (see JLS §15.8.2[4]). Role class literals are covered in § 6.1.(c).
The above is strictly needed only for cases involving implicit inheritance. It may, however, help intuition, to also consider the directly acquired role `T.R` in (b) to override the given role `S.R`.

> In line 17 of Listing 3 the implicitly inherited method `n` is called with the result of an invocation of `m`. Although `n` was defined in `S` (thus with argument type `S.R2, see line 6`) in the context of `T` it expects an argument of `T.R2`. This is correctly provided by the invocation of `m` in the context of `T`.

(f) tsuper

→ Syntax § A.5.4 (TSuperCall)
Super calls along implicit inheritance use the new keyword **tsuper**. While `super` is still available along regular inheritance, a call `tsuper.m()` selects the version of `m` of the corresponding role acquired from the super-team.
See § 2.4.2. for `tsuper` in the context of role constructors.
`tsuper` can only be used to invoke a corresponding version of the enclosing method or constructor, i.e., an expression `tsuper.m()` may only occur within the method `m` with both methods having the same signature (see § 2.3.2.(b) for an exception, where both methods have slightly different signatures).

[4] `http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#251530`

> In Listing 3 the role `R1` in team `T` overrides the implicitly inherited method `m()` from `S`. **`tsuper`**`.m()` calls the overridden method `m()` from `S.R1` (line 13).

(g) Implicitly inheriting super-types

If a role class has an explicit super class (using `extends`) this relation is inherited along implicit inheritance.

> In Listing 3 the role `R1` in `T` has `T.R0` as its implicitly inherited super class, because the corresponding role in the super-team **`extends`** `R0` (line 3).

Overriding an implicitly inherited super class is governed by § 1.3.2.(b), below.
The list of implemented interfaces is merged along implicit inheritance.

(h) Preserving visibility

A role class must provide at least as much access as the implicit super role, or a compile-time error occurs (this is in analogy to JLS §8.4.6.3[5]). Access rights of methods overridden by implicit inheritance follow the same rules as for normal overriding.

(i) Dynamic binding of constructors

When creating a role instance using `new` not only the type to instantiate is bound dynamically (cf. § 1.3.1.(e)), but also the constructor to invoke is dynamically bound in accordance to the concrete type.
Within role constructors all `this(..)` and `super(..)` calls are bound statically with respect to explicit inheritance and dynamically with respect to implicit inheritance. This means the target role name is determined statically, but using that name the suitable role type is determined using dynamic binding.
See also § 2.5.(a) on using constructors of abstract role classes.

(j) Overriding and compatibility

The rules of JLS §8.4.6[6] also apply to methods *and constructors* inherited via implicit inheritance.

(k) Covariant return types

Given a team `T1` with two roles `R1` and `R2` where `R2` explicitly inherits from `R1`, both roles defining a method `m` returning some type `A`. Given also a sub-team of `T1`, `T2`, where `T2.R1` overrides `m` with a covariant return type `B` (sub-type of `A`):

[5] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#227965
[6] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#228745

```
public team class T1 {
   protected abstract class R1 {
      abstract A m();
   }
   protected class R2 extends R1 {
      A m() { return new A(); }
   }
}
public team class T2 extends T1 {
   protected class R1 {
      @Override B m() { return new B(); } // this declaration renders class T2.R2 illegal
   }
}
```

In this situation role `T2.R2` will be illegal unless also overriding `m` with a return type that is at least `B`. Note, that the actual error occurs at the implicitly inherited method `T2.R2.m` which is not visible in the source code, even `T2.R2` need not be mentioned explicitly in the source code. A compiler should flag this as an imcompatibility at the team level, because a team must specialize inherited roles in a consistent way.

Listing 4: Example code (Teams and Roles):

```
public team class MyTeamA {
  protected class MyRole {
    String name;
    public MyRole (String n) { name = n; }
    public void print() { System.out.println("id="+name); }
  }
  protected MyRole getRole() { return new MyRole("Joe"); }
}
```

```
public team class MySubTeam extends MyTeamA {
  protected class MyRole {
    int age;
    public void setAge(int a) { age = a; }
    public void print() {
      tsuper.print();
      System.out.println("age="+age);
    }
  }
  public void doit() {
    MyRole r = getRole();
    r.setAge(27);
    r.print();
  }
}
...
MySubTeam myTeam = new MySubTeam();
myTeam.doit();
```

Listing 5: Program output

```
id=Joe
age=27
```

Effects:

- According to § 1.3., `MyTeamA` inherits from `Team` (line 1).
- An implicit role inheritance is created for `MySubTeam.MyRole` (§ 1.3.1.(c); line 11).
  If we visualize this special inheritance using a fictitious keyword `overrides` the compiler would see a declaration:

  ```
  protected class MyRole overrides MyTeamA.MyRole { ... }
  ```

- Invoking `getRole()` on `myTeam` (line 27, 20) creates an instance of `MySubTeam.MyRole` because the acquired role `MyTeamA.MyRole` is overridden by `MySubTeam.MyRole` following the rules of implicit inheritance (cf. § 1.3.1.(e)).
- Overriding of role methods and access to inherited features works as usual.
- As an example for § 1.3.1.(f) see the call `tsuper.print()` (line 15), which selects the implementation of `MyTeamA.MyRole.print`.

§ 1.3.2. Regular role inheritance

In addition to implicit inheritance, roles may also inherit using the standard Java keyword `extends`. These restrictions apply:

(a) Super-class restrictions

If the super-class of a role is again a role it must be a direct role of an enclosing team This rule is simply enforced by disallowing type anchors in the `extends` clause (see § 1.2.2.(g)). As an effect, the super-class may never be more deeply nested than the sub-class.

(b) Inheriting and overriding the extends clause

If a role overrides another role by implicit inheritance, it may change the inherited `extends` clause (see § 1.3.1.(g) above) only if the new super-class is a sub-class of the class in the overridden extends clause. I.e., an implicit sub-role may *specialize* the extends clause of its implicit super-role.

(c) Constructors and overridden 'extends'

Each constructor of a role class that overrides the extends clause of its implicit super-role must invoke a constructor of this newly introduced explicit super-class. Thus it may not use a `tsuper` constructor (see § 2.4.2.).

(d) Adding implemented interfaces

`implements` declarations are additive, i.e., an implicit sub-role may add more interfaces but has to implement all interfaces of its implicit super-role, too.

(e) Visibility of inherited methods

When a role inherits non-public methods from a regular class (as its super class), these methods are considered as private for the role, i.e., they can only be accessed in an unqualified method call `m()` using the implicit receiver `this`.

▷ Name clashes § 1.4.

ObjectTeams/Java restricts Java with respect to handling of conflicting names.

(a) Names of role classes

A role class may not have the same name as a method or field of its enclosing team. A role class may not shadow another class that is visible in the scope of the enclosing team.

(b) Names of role methods and fields

Along implicit inheritance, the names of methods or fields may not hide, shadow or obscure any previously visible name.
(see JLS §8.3[7], §8.4.6.2[8], §8.5[9], §9.3[10], §9.5[11] (hiding), §6.3.1[12] (shadowing), §6.3.2[13] (obscuring).

▷ Team and role nesting § 1.5.

Multi-level nesting of classes is restricted only by the following rules.

[7] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#40898
[8] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#227928
[9] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#246026
[10] http://java.sun.com/docs/books/jls/second_edition/html/interfaces.doc.html#78642
[11] http://java.sun.com/docs/books/jls/second_edition/html/interfaces.doc.html#252566
[12] http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#34133
[13] http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#104058

Listing 6: Example code (Nesting):

```
1  public team class SuperOuter {
2    public team class RoleAndTeam {
3      protected class InnerRole {
4        Runnable foo() { return null; }
5      }
6    }
7    public team class RoleAndTeamSub extends RoleAndTeam {
8      protected class InnerRole {
9        Runnable foo() { throw new RuntimeException(); }
10     }
11   }
12 }
13 public team class OuterTeam extends SuperOuter {
14   public team class RoleAndTeam {
15     protected class InnerRole {
16       Runnable foo() {
17         class Local {};
18         return new Runnable() { // anonymous class definition
19           public void run() {}
20         };
21       }
22       // class IllegalMember {}
23     }
24   }
25   public team class RoleAndTeamSub {
26     protected class InnerRole {
27       Runnable foo() {
28         RoleAndTeamSub.tsuper.foo();
29         return OuterTeam.tsuper.foo();
30       };
31     }
32   }
33 }
```

(a) Nested teams

If a role class is also marked using the `team` modifier, it may contain roles at the next level of nesting.

- In the above example (Listing 6) class `RoleAndTeam` starting in line 14 is a role of `OuterTeam` and at the same time a team containing a further role `InnerRole`

Such a hybrid role-and-team has all properties of both kinds of classes.

(b) Nested classes of roles

A regular role class (ie., not marked as `team`, see above) may contain local types (see JLS §14.3[14] - in the example: class `Local`), anonymous types (JLS §15.9.5[15] - in the example: class defined in lines 18-20) but no member types (JLS §8.5[16] - in the example: illegal class `IllegalMember`).
The effect is, that nested types of a regular role cannot be used outside the scope of their enclosing role.

[14] http://java.sun.com/docs/books/jls/second_edition/html/statements.doc.html#247766
[15] http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#252986
[16] http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#246026

(c) Prohibition of cycles

A nested team may not extend its own enclosing team.

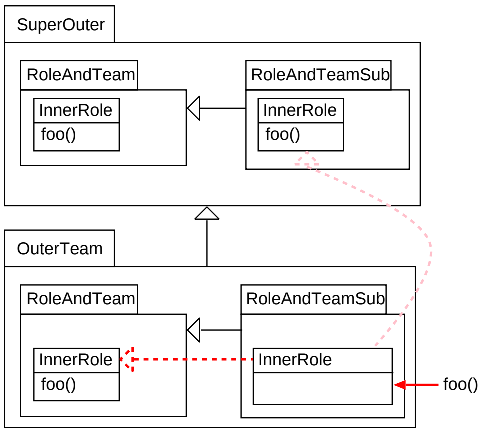
(d) Prohibition of name clashes

A nested team may inherit roles from multiple sources: its explicit super team and any of its implicit super classes (roles) from different levels of nesting. If from different sources a team inherits two or more roles of the same name that are not related by implicit inheritance, this is an illegal name clash.

(e) Precedence among different supers

If a role inherits the same feature from several super roles (super and tsuper), an implicitly inherited version always overrides any explicitly inherited feature.
Also implicit inheritance alone may produce several candidate methods inherited by a role class. This is a result of team-nesting where each level of nesting may add one more tsuper role. In that case inner team inheritance has precedence over outer team inheritance.

In the above example (Listing 6) role `OuterTeam.RoleAndTeamSub.InnerRole` has two direct tsuper roles: `OuterTeam.RoleAndTeam.InnerRole` and `SuperOuter.RoleAndTeamSub.InnerRole`. Without the method `foo` defined in lines 27-30, the enclosing class `OuterTeam.RoleAndTeamSub.InnerRole` would inherit the method `foo` defined in line 16, because the inner inheritance between `RoleAndTeamSub` and `RoleAndTeam` binds stronger than the outer inheritance between `OuterTeam` and `SuperOuter`.



(f) Qualified tsuper

A role in a nested team may qualify the keyword `tsuper` (see § 1.3.1.(f) above) by a type name in order to select among different method version inherited from different implicit super classes. A term `OuterTeam.tsuper` evaluates to the super-class, say `SuperOuter`, of an enclosing team "`OuterTeam`". A method call `OuterTeam.tsuper.m()` evaluates to the method version within `SuperOuter` that best corresponds to the current method containing the tsuper-call.

- In the above example (Listing 6) line 28 is identical to an unqualified tsuper-call
- Line 29 selects a corresponding method from the context of `SuperOuter` resolving to `SuperOuter.RoleAndTeamSub.InnerRole.foo()`

Language implementation:

Role acquisition and implicit inheritance can be implemented in at least two ways.

Virtual classes: Each role class is an overridable feature of its enclosing team. Role classes are resolved by dynamic binding with respect to the enclosing team instance. This implementation requires multiple-inheritance in order to also allow regular inheritance between roles of the same team. `super` and `tsuper` select parent versions of a method along the two dimensions of inheritance.

Copy inheritance: Role acquisition from a super-team has the effect of copying a role definition `T.R` yielding a new role `Tsub.R`. All role applications `Rx` in the role copy refer to `Tsub.Rx`. Implicit role inheritance extends a role copy in-place. Only the `tsuper` construct allows to access the previous version of a method (i.e. before in-place overriding).

References:

[1] Ole Lehrmann Madsen and Birger Møller-Pedersen. *Virtual classes: A powerful mechanism in object-oriented programming*. In Proceedings OOPSLA 89, ACM SIGPLAN Notices, volume 24, 10, pages 397-406, October 1989.

[2] Erik Ernst. *Family Polymorphism.* In Proceedings ECOOP 2001, LNCS 2072, pages 303-326, Springer, 2001.

▷ Role Binding § 2.

Roles and base classes

playedBy relation A role can be bound to a class outside the team by a `playedBy` relation, which declares that each role instances is associated to a base instances.

Base class The class to which a role is bound (using `playedBy`) is called its **base class**. Role instances may inherit and override features from their base instance, which is declared using **callout** (§ 3.) and **callin** (§ 4.) method bindings.

Bound role Each role class that declares a `playedBy` relation is called a **bound role**. The term bound role may also be used for the instances of such a class.

Lifting / lowering Translations between a role and its base are called **lifting** (base to role) (§ 2.3.) and **lowering** (role to base) (§ 2.2.).

Translation polymorphism Conformance between a role and a base is governed by **translation polymorphism**, which refers to a substitutability that is achieved using either lifting or lowering.

Declared lifting Generally, lifting happens implicitly at data flows between a role object and its base. Team level methods provide additional data flows, where lifting may be declared explicitly.

▷ playedBy relation § 2.1.

→ Syntax § A.1.1

(a) Role-base binding

Roles are bound to a base class by the `playedBy` keyword.

```
public team class MyTeamA {
  public class MyRole playedBy MyBase {
    ...
  }
}
```

(b) Inheritance

The `playedBy` relation is inherited along explicit and implicit (§ 1.3.1.(c)) role inheritance.

(c) Covariant refinement

An *explicit* sub-role (sub-class using `extends`) can refine the `playedBy` relation to a more specific base class (this is the basis for smart lifting (§ 2.3.3.)).
If a role class inherits several `playedBy` relations from its super-class and its super-interfaces, there must be a most specific base-class among these relations, which is conform to all other base-classes. This most specific base-class is the base-class of the current role.

(d) No-variance

An *implicit* sub-role (according to § 1.3.1.(c)) may only add a `playedBy` relation but never change an existing one.
Note however, that implicit inheritance may implicitly specialize an existing `playedBy` relation (this advanced situation is illustrated in § 2.7.(d)).

(e) Use of playedBy bindings

The `playedBy` relation by itself has no effect on the behavior of role and base objects. It is, however, the precondition for translation polymorphism (lowering: § 2.2. and lifting: § 2.3.) and for method bindings (callout: § 3. and callin: § 4.).

(f) Effect on garbage collection

A role and its base object form one conceptual entity. The garbage collector will see a role and its base object as linked in a bidirectional manner. As a result, a role cannot be garbage collected if its base is still reachable and vice versa.
Internally a team manages its roles and corresponding bases using weak references. When using one of the `getAllRoles(..)` methods (see § 6.1.(a)), the result may be non-deterministic because these internal structures may hold weak references to objects that will be collected by the next run of the garbage collector. We advise clients of `getAllRoles(..)` to call `System.gc()` prior to calling `getAllRoles(..)` in order to ensure deterministic results.

§ 2.1.1. Binding interfaces

Role base bindings may involve classes and/or interfaces. An interface defined as a member of a team is a role interface and may therefore have a `playedBy` clause. Also the type mentioned after the `playedBy` keyword may be an interface.

Implementation limitation:
The language implementation as of OTDT version 1.0.X cannot yet bind a role class to a base interface, but this restriction will go in the future.

§ 2.1.2. Legal base classes

Generally, the base class mentioned after `playedBy` must be visible in the enclosing scope (see below (§ 2.1.2.(c)) for an exception). Normally, this scope is defined just by the imports of the enclosing team. For role files (§ 1.2.5.(b)) also additional imports in the role file are considered.
§ 2.1.2.(d) below defines how imports can be constrained so that certain types can be used as base types, only.

(a) No role of the same team

The base class of any role class must not be a role of the same team.
It is also not allowed to declare a role class of the same name as a base class bound to this or another role of the enclosing team, if that base class is given with its simple name and resolved using a regular import. Put differently, a base class mentioned after `playedBy` may not be *shadowed* by any role class of the enclosing team.
Base imports as defined below (§ 2.1.2.(d)) relax this rule by allowing to import a

class as a base class only. In that case no shadowing occurs since the scopes for base classes and roles are disjoint.

(b) No cycles

The base class mentioned after `playedBy` may not be an enclosing type (at any depth) of the role class being defined.
This rule prohibits the creation of cycles where the base instance of a given role `R` contains roles of the same type `R`.
More generally any sequence of classes C_1, C_2, .. C_n were each C_{i+1} is either a member or the base class of C_i and C_n = C_1 is forbidden.
Conversely, it is also prohibited to bind a role class to its own inner class.

(c) Base class decapsulation

If a base class referenced after `playedBy` exists but is not visible under normal visibility rules of Java, this restriction may be overridden. This concept is called **decapsulation**, i.e., the opposite of encapsulation (see also § 3.4.). A compiler should signal any occurrence of base class decapsulation. If a compiler supports to configure warnings this may be used to let the user choose to (a) ignore base class decapsulation, (b) treat it as a warning or even (c) treat it as an error.
Binding to a `final` base class is also considered as decapsulation, since a `playedBy` relationship has powers similar to an `extends` relationship, which is prohibited by marking a class as `final`.
Decapsulation is not allowed if the base class is a confined role (see § 7.2.).
Within the current role a decapsulated base class can be mentioned in the right-hand-side of any method binding (callout (§ 3.) or callin (§ 4.)). Also arguments in these positions are allowed to mention the decapsulated base class:

- the first argument of one of the role's constructors (see lifting constructor (§ 2.4.1.)).
- the base side of an argument with declared lifting (see declared lifting (§ 2.3.2.)).

(d) Base imports

If the main type in a file denotes a team, the modifier `base` can be applied to an import in order to specify that this type should be imported for application as a base type only. Example:

```
import base some.pack.MyBase;
public team class MyTeam {
  // simple name resolves to imported class:
  protected class MyRole playedBy MyBase { }
  MyBase illegalDeclaration; // base import does not apply for this position
}
```

Types imported by a base import can only be used in the same positions where also base class decapsulation (§ 2.1.2.(c)) is applicable.
It is recommended that a type mentioned after the keyword `playedBy` is always imported with the `base` modifier, otherwise the compiler will give a warning.
Base imports create a scope that is disjoint from the normal scope. Thus, names that are imported as base will never clash with normally visible names (in contrast to § 1.4.). More specifically, it is not a problem to use a base class's name also for its role if a base import is used.

(e) No free type parameters

Neither the role class nor the base class in a playedBy binding must have any *free type parameters*. If both classes are specified with a type parameter of the same name, both parameters are identified and are not considered as *free*.
From this follows that a role class cannot have more type parameters than its base. Conversely, only one situation exists where a base class can have more type parameters than a role class bound to it: if the role class has no type parameters a generic base class can be bound using the base class's raw type, i.e., without specifying type arguments.

<u>Note:</u>
The information from the `playedBy` declaration is used at run-time to associate role instances to base instances. Specifying a base class with free type parameters would imply that only such base instances are decorated by a role whose type is conform to the specified parameterized class. However, type arguments are not available at run-time, thus the run-time environment is not able to decide which base instances should have a role and which should not. This is due to the design of generics in Java which are realized by erasure.

The following example shows how generics can be used in various positions. Note, that some of the concepts used in the example will be explained in later sections.

```
1  public class ValueTrafo<T> {
2    public T transform(T val) throws Exception { /* ... */ }
3  }
4  public team class TransformTeam {
5      protected class SafeTrafo<U> playedBy ValueTrafo<U> {
6          U transform(U v) -> U transform(U val);
7          protected U safeTransform(U v) {
8              try {
9                  return transform(v);
10             } catch (Exception e) {
11                 return v;
12             }
13         }
14     }
15     <V> V perform(ValueTrafo<V> as SafeTrafo<V> trafo, V value) {
16         return trafo.safeTransform(value);
17     }
18 }
19 ...
20 ValueTrafo<String> trafo = new ValueTrafo<String>();
21 TransformTeam safeTrafo = new TransformTeam();
22 String s = safeTrafo.perform(trafo, "Testing");
```

Explanation

- Line 5 shows a role with type parameter `U` where the type parameter is identified with the corresponding type parameter of the role's base class (which is originally declared as `T` in line 1.
- Line 6 shows a callout binding (§ 3.) which mappes a base method to a corresponding role method while maintaining the flexible typing.
- The regular method in lines 7-13 just passes values of type `U` around.
- The generic method in line 15 ff. uses declared lifting (§ 2.3.2.) to obtain a role for a given base object. The method has no knowledge about the concrete type arguments of

either role nor base, but works under the guarantee that both type arguments will be the same for any single invocation.

- Lines 20 ff. finally create instances of base and team and invoke the behavior thereby instantiating type parameters to `String`.

▷ Lowering § 2.2.

Each instance of a bound role class internally stores a reference to its base object. The reference is guaranteed to exist for each bound role instance, and cannot be changed during its lifetime.

(a) Definition of lowering

Retrieving the base object from a role object is called **lowering**. No other means exists for accessing the base reference.

(b) Places of lowering

The lowering translation is not meant to be invoked by client code, but **implicit translations** are inserted by the compiler at all places where a role type is provided while the corresponding base type (or a super type) was expected.
In other words: lowering translations are inserted by the compiler at all places in a program which would otherwise not be type correct and which using lowering are statically type correct. This may concern:

- the right hand side of an assignment wrt. the static type of the left hand side,
- the argument values of a method or constructor call wrt. the static type of the corresponding formal parameter,
- the return value of a method compared to the declared return type of the method.
- a role parameter in a callout binding (§ 3.3.(d))
- or the return value in a callin binding (§ 4.5.(d))

```
1  public team class MyTeamA {
2    public class MyRole playedBy MyBase { ... }
3    void useMyBase(MyBase myb) {...}
4    MyRole returnMyRole() {...}
5    public void doSomething() {
6      MyRole r = new MyRole(new MyBase());
7      MyBase b = r;
8      useMyBase(r);
9      MyBase b2 = returnMyRole();
10   }
11 }
```

Effects: An instance of type `MyRole` is lowered to type `MyBase` when

- assigning it to `b` (line 7)
- passing it as argument to a method with formal parameter of type `MyBase` (line 8)
- assigning the return value to a variable of type `MyBase` (line 9)

Note: The constructor call in line 6 uses the *lifting constructor* as defined in § 2.4.1.

Lowering translations are not inserted for

- reference comparison (using == or !=)
- `instanceof` checks
- cast expressions
- return values in callout bindings § 3.3.(d))
- parameters in callin bindings (§ 4.5.(d))

For cases where lowering shall be *forced* see § 2.2.(d) below.

(c) Typing

The static type of an implicit lowering translation is the base class declared using `playedBy` in the respective role class.

(d) Explicit lowering

If a base type is also the super type of its role, which frequently happens, if a base reference is known only by the type `Object`, lowering cannot be deduced automatically, since a type could be interpreted both as a role type and a base type. These cases may need **explicit lowering**. For this purpose the role class must declare to implement the interface **`ILowerable`** (from `org.objectteams.Team`). This will cause the compiler to generate a method

```
public Object lower()
```

for the given role class. Client code may use this method to explicitly request the base object of a given role object.

```
public team class MyTeamA {
  public class MyRole implements ILowerable playedBy MyBase { ... }
  public void doSomething() {
    MyRole r = new MyRole(new MyBase());
    Object oMyRole = r;
    Object oMyBase = r.lower();
  }
}
```

(e) Lowering of arrays

Lowering also works for arrays of role objects. In order to lower an array of role objects, a new array is created and filled with base objects, one for each role object in the original array. The array may have any number of dimensions at any shape. The lowered array will have exactly the same shape.
Note, that each lowering translation will create a new array.

(f) Ambiguous lowering

When assigning a value of a bound role type to a variable or argument of type `java.lang.Object` this situation is considered as ambiguous lowering because the assignment could apply either (a) a direct upcast to `Object` or (b) lowering and then upcasting. In such situations the compiler will *not* insert a lowering translation, but a configurable warning will be issued.

▷ Lifting § 2.3.

Lifting is the reverse translation of lowering. However, lifting is a bit more demanding, since a given base object may have zero to many role objects bound to it. Therefor, the lifting translation requires more context information and may require to create role objects on demand.

(a) Definition of lifting

Retrieving a role for a given base object is called **lifting**. Lifting is guaranteed to yield the same role object for subsequent calls regarding the same base object, the same team instance and the same role class (see § 2.3.4. for cases of ambiguity that are signaled by compiler warnings and possibly runtime exceptions).

(b) Places of lifting

The lifting translation is not meant to be invoked by client code, but translations are inserted by the compiler at the following locations:

- Callout bindings (§ 3.3.(c)) (result)
- Callin bindings (§ 4.5.(a)) (call target and parameters)
- Declared lifting (§ 2.3.2.)

(c) Typing

A lifting translation statically expects a specific role class. This expected role class must have a `playedBy` clause (either directly, or inherited (explicitly or implicitly) from a super role), to which the given base type is conform.

(d) Lifting of arrays

Lifting also works for arrays of role objects. For lifting an array of base objects a new array is created and filled with role objects, one for each base object in the original array. In contrast to the role objects themselves, lifted arrays are never reused for subsequent lifting invocations.

The term **translation polymorphism** describes the fact that at certain points values can be passed which are not conform to the respective declared type considering only regular inheritance (`extends`). With translation polymorphism it suffices that a value can be translated using lifting or lowering.

▷ Implicit role creation § 2.3.1.

Lifting tries to reuse existing role objects so that role state persists across lifting and lowering. If no suitable role instance is found during lifting, a new role is created.

(a) Reuse of existing role objects

A role object is considered suitable for reuse during lifting, if these three items are identical:

1. the given base object
2. the given team object

3. the statically required role type

For the relation between the statically required role type and the actual type of the role object see "smart lifting" (§ 2.3.3.).

(b) Default lifting constructor

Lifting uses a default constructor which takes exactly one argument of the type of the declared base class (after `playedBy`). By default the compiler generates such a constructor for each bound role. On the other hand, default constructors that take no arguments (as in JLS §8.8.7[17]) are never generated for bound roles.
The super-constructor to be invoked by a default lifting constructor depends on whether the role's super class is a bound role or not.

- If the super-class is a bound role, the default lifting constructor will invoke the default lifting constructor of the super-class.
- If the super-class is not a bound role, the default lifting constructor will invoke the normal argumentless default constructor of the super-class.

(c) Custom lifting constructor

If a role class declares a custom constructor with the same signature as the default lifting constructor, this constructor is used during lifting. This custom constructor may pre-assume that the role has been setup properly regarding its base-link and registered in the team's internal map of roles.
If a bound role has an unbound super-class without an argumentless constructor, providing a custom lifting constructor is obligatory, because no legal default lifting constructor can be generated.

§ 2.3.2. Declared lifting

→ Syntax § A.6.2

(a) Parameters with declared lifting

A non-static team-level method or constructor may declare a parameter with two types in order to explicitly denote a place of **lifting**. Using the syntax

```
public void m (BaseClass as RoleClass param) { stmts }
```

a liftable parameter can be declared, provided the second type (`RoleClass`) is a role of (`playedBy`) the first type (`BaseClass`). Furthermore, the role type must be a role of the enclosing team class defining the given method. The role type must be given by its simple (i.e., unqualified) name.
Such a signature requires the caller to provide a base object (here `BaseClass`), but the callee receives a role object (here `RoleClass`). In fact, the client sees a signature in which the "`as RoleClass`" part is omitted.
Compatibility between caller and callee sides is achieved by an implicitly inserted lifting translation. A signature using declared lifting is only valid, if the requested lifting is possible (see § 2.3.3. and § 2.3.4. for details).

[17] `http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#16823`

(b) Super in the context of declared lifting

Calling `super` or `tsuper` in a method or constructor which declares lifting for one or more parameters refers to a method or constructor with role type parameters, i.e., lifting takes place *before* super invocation. Nevertheless, the super method may also have a declared lifting signature. It will then see the same role instance(s) as the current method.

(c) Declared lifting of arrays

If a parameter involving explicit lifting should be of an **array** type, the syntax is

```
public void m (BaseClass as RoleClass param[]) ...
```

Here the brackets denoting the array apply to both types, `BaseClass` and `RoleClass`.

(d) Declared lifting for catch blocks

Also the argument of a catch block may apply declared lifting like in:

```
catch (BaseException as RoleClass param) { stmts }
```

This syntax is only valid in a non-static scope of a team (directly or nested). In the given example, `RoleClass` must be played by `BaseException`. Note, that `RoleClass` itself need not be a throwable. As the effect of this declaration the catch block will catch any exception of type `BaseException` and provides it wrapped with a `RoleClass` instance to the subsequent block.
Also note, that re-throwing the given instance `param` has the semantics of implicitly lowering the role to its base exception before throwing, because the role conforms to the required type `Throwable` only via lowering.

(e) Generic declared lifting

A method with declared lifting may introduce a type parameter that is bounded relative to a given role type. Such bound is declared as:

```
<AnyBase base SuperRole>
void teamMethod(AnyBase as SuperRole arg) {
    // body using arg as of type SuperRole
}
```

This means that `AnyBase` is a type parameter whose instantiations must all be liftable to role `SuperRole`.
The given type bound requires the call site to supply an argument that is compatible to any base class for which the current team contains a bound role that is a sub class of `SuperRole`, including `SuperRole` itself. However, `SuperRole` itself need not be bound to any base class. On the other hand, different valid substitutions for `AnyBase` need not be related by inheritance.

<u>Note:</u>
This feature supports generalized treatment of otherwise unrelated base classes. This is done by defining one bound role for each base under consideration and by having all these roles extend a common unbound role.

Listing 7: Example code (Declared Lifting):

```
1 team class Super {
2   public class MyRole playedBy MyBase { ... }
3   void m (MyRole o) { ... };
4 }
5 team class Sub extends Super {
6   void m (MyBase as MyRole o) {
7     // inside this method o is of type MyRole
8     super.m(o);
9   }
10 }
11 Sub s_team = new Sub();
12 MyBase b = new MyBase();
13 s_team.m(b); // clients see a parameter "MyBase o"
```

Effects:

- Clients use method `m` with a base instance (type `MyBase`) as its argument (line 13).
- Before executing the body of `m`, the argument is lifted such that the method body receives the argument as of type `MyRole` (line 8).

§ 2.3.3. Smart lifting

In situations where role and base classes are part of some inheritance hierarchies (`extends`), choosing the appropriate role class during lifting involves the following rules:

(a) Static adjustment

If a base class `B` shall be lifted to a role class `R` that is not bound to (`playedBy`) `B`, but if a subclass of `R` — say `R2` — is bound to `B`, lifting is statically setup to use `R2`, the most general subclass of `R` that is bound to `B` or one of its super-types.

Restriction:

This step is not applicable for parameter mappings of `replace` callin bindings (§ 4.5.(d)).

(b) Dynamic selection of a role class

At runtime also the dynamic type of a base object is considered: Lifting always tries to use a role class that is bound to the exact class of the base object. Lifting considers all role–base pairs bound by `playedBy` such that the role class is a sub-class of the required (statically declared) role type and the base class is a super-class of the dynamic type of the base object.
From those possible pairs the most specific base class is chosen. If multiple role classes are bound to this base class the most specific of these classes is chosen.

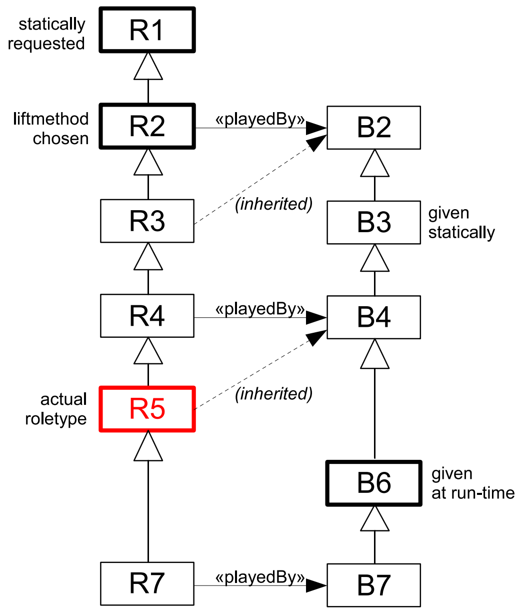
(c) Team as closed world

In the above analysis gathering all role-base pairs is performed at compile-time. From this follows, that a team class can only be compiled when all its contained role classes are known and a role class can never be compiled without its team.
The analysis includes all roles and their bindings that are inherited from the super-team.

(d) Selection regardless of abstractness

Smart lifting is not affected by abstractness of role classes. For the effect of abstract role classes see § 2.5..

Complex Example:



role class	base class
class R1	
class R2 extends R1 playedBy B2	class B2
class R3 extends R2 /* *inherited: playedBy B2* */	class B3 extends B2
class R4 extends R3 playedBy B4	class B4 extends B3
class R5 extends R4 /* *inherited: playedBy B4* */	
	class B6 extends B4
class R7 extends R5 playedBy B7	class B7 extends B6

- If declarations require lifting `B3` to `R1` this is statically refined to use `R2` instead, because this is the most general class declaring a binding to a super–class of `B3`.
- If the dynamic base type in the same situation is `B6`, three steps select the appropriate role:
  1. By searching all `playedBy` clauses (including those that are inherited) the following role–base pairs are candidates:
     `(R2,B2)`, `(R3,B2)`, `(R4,B4)` and `(R5,B4)`.
  2. From these pairs the two containing the most specific base class `B4` are chosen.
  3. This makes `R4` and `R5` role candidates, from which the most specific `R5` is finally chosen.

If the inheritance hierarchies of the involved base and role classes are given (like in the figure above) the smart lifting algorithm can be rephrased to the following "graphical" rule:

Starting with the dynamic base type (B6 in the example) move upwards the the inheritance relation until you reach a base class bound to a role class indicated by a «playedBy» arrow *pointing to the base class (B4). This role class must be conform to the requested role type. Switch to the role side along this arrow (R4). Now move downwards the role inheritance hierarchy as long as the subrole does not refine the playedBy relationship (indicated by another «playedBy» arrow). The bottom role you reach this way (R5) is the role type selected by smart lifting.*

§ 2.3.4. Binding ambiguities

While all examples so far have only shown 1-to-1 class bindings, several cases of multiple bindings are allowable. Ambiguities may be detected at compile time and/or at runtime.

(a) Potential ambiguity

A **potential ambiguity** is given, if two role classes `R1` and `R2` exist such that

- `R1` and `R2` are played by the same base class `B`, and
- `R1` and `R2` have a common super role `R0`, which is also bound to a base class `B0`, and
- neither role class `R1` nor `R2` is a (indirect) sub-class of the other.

Note:
According to § 2.1.(c), if `B` is distinct from `B0` it has to be a sub-class of `B0`.

Effect:
In this case the compiler issues a warning, stating that the `B` may not be liftable, *because both role classes `R1` and `R2` are candidates and there is no reason to prefer one over the other.*
If no potential ambiguity is detected, lifting will always be unambiguous.

In the above situation, trying to lift an instance of type `B` to the role type `R0` is an **illegal lifting request**. If `R0` is bound to the same base class `B` as its sub-roles `R1` and `R2` are, role `R0` is **unliftable**, meaning that no instance of `R0` can ever by obtained by lifting.

Listing 8: Example code (Potential Ambiguity):

```
team class MyTeam {
  public class SuperRole playedBy MyBase {...}
  public class SubRoleA extends SuperRole {...}
  public class SubRoleB extends SuperRole {...}
}
```

(b) Definite ambiguity

A **definite ambiguity** is given if

- the situation of potential ambiguity according to (a) above is given and
- lifting is requested (either by method binding or explicitly (§ 2.3.2.)) from the shared base class `B` to any role class `R0` that is a common super role for `R1` and `R2`.

Effect:
Definite ambiguity is a compile time error.

Listing 9: Example code (Definite Ambiguity):

```
team class MyTeam {
  public class SuperRole playedBy MyBase {...}
  public class SubRoleA extends SuperRole playedBy SubBase {...}
  public class SubRoleB extends SuperRole playedBy SubBase {...}

  public void useSuperRole(SubBase as SuperRole r) {...}
}
```

(c) Actual ambiguity

At runtime **actual ambiguity** may occur if for the *dynamic type* of a base to be lifted the conditions of (b) above hold accordingly. Actual ambiguity is only possible in cases reported by the compiler as potential ambiguity.

<u>*Effect:*</u>

An actual ambiguity is reported at runtime by throwing a `org.objectteams.LiftingFailedException`.

Listing 10: Example code (Actual Ambiguity):

```
team class MyTeam {
  public class SuperRole playedBy MyBase {...}
  public class SubRoleA extends SuperRole playedBy SubBase {...}
  public class SubRoleB extends SuperRole playedBy SubBase {...}

  public void useSuperRole(MyBase as SuperRole r) {...}
}
// plus these calls:
MyTeam mt = new MyTeam();
mt.useSuperRole(new SubBase());
```

(d) Mismatching role

In cases of potential ambiguity another runtime error may occur: a **mismatching role** is encountered when a role is found in the cache, which is not conform to the required type. This happens, if the base object has previously been lifted to a type that is incompatible with the currently requested type.

<u>*Effect:*</u>

This is reported by throwing a `org.objectteams.WrongRoleException`.

Listing 11: Example code (Mismatching Role):

```
team class MyTeam {
  public class SuperRole playedBy MyBase {...}
  public class SubRoleA extends SuperRole {...}
  public class SubRoleB extends SuperRole {...}

  public void useRoleA(MyBase as SubRoleA r) {...}
  public void useRoleB(MyBase as SubRoleB r) {...}
}
// plus these calls:
MyTeam mt = new MyTeam();
MyBase b = new MyBase();
mt.useRoleA(b); // creates a SubRoleA for b
mt.useRoleB(b); // finds the SubRoleA which is not compatible
                // to the expected type SubRoleB.
```

From the second item of § 2.3.4.(a) follows, that for binding ambiguities different role hierarchies are analyzed in isolation. For this analysis only those role classes are considered that are bound to a base class (directly using `playedBy` or by inheriting this relation from another role class). I.e., two role classes that have no common bound super role will never cause any ambiguity.

§ 2.4. Explicit role creation

Lifting is the normal technique by which role objects are created implicitly. This section defines under which conditions a role can also be created explicitly.

§ 2.4.1. Role creation via a lifting constructor

Lifting uses the default constructor for roles (see § 2.3.1.). This constructor can be invoked from client code, if the following rules are respected.

(a) Team context

The lifting constructor can be used only within the enclosing team of the role to be instantiated. Thus, qualified allocation expressions (`someTeam.new SomeRole(..)`) may never use the lifting constructor.

(b) Fresh base object

If the argument to a lifting constructor invocation is a `new` expression, creating a fresh base object, the use of the lifting constructor is safe. Otherwise the rules of (c) below apply.

(c) Duplicate role runtime check

If it cannot be syntactically derived, that the argument to a lifting constructor is a freshly created base object (b), a compile time warning will signal that an additional runtime check is needed: It must be prevented that a new role is created for a base object, which already has a role of the required type in the given team. It is not possible to replace an existing role by use of the lifting constructor. At runtime, any attempt to do so will cause a `org.objectteams.DuplicateRoleException` to be thrown. This exception can only occur in situations where the mentioned compile time warning had been issued.
§ 6.1. will introduce reflective functions which can be used to manually prevent errors like a duplicate role.

§ 2.4.2. Role creation via a regular constructor

Roles may also be created explicitly using a custom constructor with arbitrary signature other than the signature of the lifting constructor.
Within role constructors, four kinds of self-calls are possible:

`base(..)` A constructor of the corresponding base class (§ A.5.(c)).

`this(..)` Another constructor of the same class.

`super(..)` A constructor of the super-class (normal `extends`), unless the super-class is bound to a different base class, in which case calling `super(..)` is not legal.

`tsuper(..)` A constructor of the corresponding role of the super-team (§ A.5.(e)). Also see the constraint in § 1.3.2.(c).

(a) Unbound roles

Each constructor of a role that is **not bound** to a base class must use one of `this(..)`, `super(..)` or `tsuper(..)`.

(b) Bound roles

Each constructor of a **bound role** must directly or indirectly invoke either a `base(..)` constructor or a lifting constructor (see § 2.3.1.). Indirect calls to the base constructor or lifting constructor may use any of `this(..)`, `super(..)` or `tsuper(..)`, which simply delegates the obligation to the called constructor.
If a constructor referenced by `base(..)` is not visible according to the regular rules of Java, it may still be called using **decapsulation** (see also § 3.4., § 2.1.2.(c)).
Note, that if the super or tsuper role is not bound, delegating the obligation to that unbound role will not work.

(c) Super-call for bound roles

Instead of or prior to calling `base(..)` a constructor of a bound role explicitly or implicitly calls a super constructor. Which constructor is applicable depends on the super role and its `playedBy` clause.

- If the super role is bound to the same base class as the current role is,
  - not writing a super-call causes the lifting constructor of the super role to be invoked.
  - explicitly calling a super constructor requires the super constructor to *either*
    1. create a role instance using a base constructor call (directly or indirectly), *or*
    2. be a lifting constructor receiving a base instance, which the current role must provide as the argument.
- If the super role is bound but the current role refines the `playedBy` relationship (cf. § 2.1.(c)),
  - a lifting constructor must be called explicitly passing a base object as the argument.
- If the role has an explicit or implicit super role which is unbound the constructor may optionally call a super constructor (using `super(..)` or `tsuper(..)`) prior to calling `base(..)`. Otherwise the default constructor is implicitly invoked.

When invoking a lifting constructor of a super role the base object can optionally be obtained by using a base constructor call as an expression:

```
super(base(<args>));
```

The language system evaluates the base constructor by creating an instance of the appropriate base class using a constructor with matching signature. Also the internal links are setup that are needed for accessing the base object from the role and for lifting the base object to the new role in the future.

The syntax for base constructors follows the rule that role implementations never directly refer to any names of base classes or their features.

§ 2.4.3. Role creation in the presence of smart lifting

Explicitly instantiating a role `R1` bound to a base `B` where smart lifting of `B` to `R1` would actually provide a subrole `R2` is dangerous: Instantiation enters the `R1` into the team's internal cache. If at any time later lifting this `B` to `R2` is requested, which is a legal request, the runtime system will answer by throwing a `org.objectteams.WrongRoleException` because it finds the `R1` instead of the required `R2`. For this reason, in this specific situation the explicit instantiation `new R1(..)` will be flagged by a warning. The problem can be avoided by using `R2` in the instantiation expression.

Listing 12: Example code (WrongRoleException):

```
1  public class B { void bm() {} }
2  public team class T {
3    protected class R1 playedBy B {...}
4    protected class R2 extends R1 { // inherits the binding to B
5      void rm() { /* body omitted */ }
6    }
7    public B getDecoratedB() {
8      return new R1(new B()); // compile-time warning!
9    }
10   public void requestLifting(B as R2 r) {}
11 }
12 // plus these calls:
13 T t = new T();
14 B b = t.getDecoratedB(); // creates an R1 for b
15 t.requestLifting(b); // => org.objectteams.WrongRoleException!
```

- A note on line 8: this line passes a fresh instance of `B` to the lifting constructor of `R1` (see § 2.4.1.(b)). In order to return this `B` instance lowering is implicitly used for the return statement.
- When line 15 is executed, a lifting of `b` to `R2` is requested but due to line 8 an `R1` is found in the internal cache.

§ 2.5. Abstract Roles

Overriding of role classes and dynamic binding of role types (§ 1.3.1.(e)) adds new cases to **creation** with respect to abstract classes.

(a) Using abstract classes for creation

Abstract role classes can indeed be used for object creation. The effect of such a statement is that the team must be marked `abstract`. Only those sub-teams are concrete that provide concrete versions for all role classes used in creation expressions.

This includes the case, where a super-team has a concrete role class and creates instances of this role class and only the sub-team changes the status of this role class to abstract. Also here the sub-team must be marked abstract, because it contains an abstract role class that is used in creation expressions.

Interpretation:

*Since the type in a role creation expression is late-bound relative to the enclosing team instance, abstract role classes can be seen as the hook in a **template&hook pattern** that is raised from the method level to the class level: A super-team may already refer to the constructor of an abstract role class, only the sub-team will provide the concrete role class to fill the hook with the necessary implementation.*

(b) Relevant roles

A team must be marked `abstract` if one of its **relevant roles** is abstract.
A role is relevant in this sense if

- the role class is public *or if*
- an explicit `new` expression would require to create instances of the role class, *or if*
- any of the lifting methods of the enclosing team would require to create instances of the role class.
  A role is irrelevant with respect to lifting if either of the following holds:
  - It is not bound to a base class, neither directly nor by an inherited `playedBy` clause.
  - It has a sub-role without a `playedBy` clause.
  - It is bound to an abstract base class, and for all concrete sub-classes of the base class, a binding to a more specific role class exists.

If neither property, relevance nor irrelevance, can be shown for an abstract role, a warning is given in case the enclosing team is not abstract.

▷ Explicit base references § 2.6.

The role-base link is not meant to be accessed explicitly from programs, but it is fully under the control of compiler and runtime environment. Accessing features of a role's base object is done by callout bindings (§ 3.). Yet, a keyword `base` exists, which can be used in the following contexts:

(a) Externalized roles of a base team

If the base class of a role `T1.R1` is again a team `T2`, roles of that team `T2` can be externalized (see § 1.2.2.) using `base` as their type anchor. Given that `R2` is a role of `T2`, one could write:

```
public team class T1 {
  protected class R1 playedBy T2 {
    protected R2<@base> aRoleOfMyBase;
  }
}
```

This syntax is only legal within the body of the role `T1.R1` which is bound to the team `T2` containing role `R2`. A static type prefix can be used to disambiguate a base

anchor, so the explicit variant of the above type would be `R2<@R1.base>`.
It is not legal to use a type anchor containing `base` as an element in a path of references like `<@base.field>` or `<@field.base>`.

(b) Explicit base object creation

Within a role constructor (which is not the lifting constructor) the syntax `base(`*arguments*`)` causes an instance of the bound base class to be created and linked (see § 2.4.2.).

(c) Base call in callin method

Within a callin method (§ 4.2.(d)) an expression `base.m(`*args*`)` is used to invoke the originally called method (see § 4.3.).

(d) Base guard predicates

Guard predicates (§ 5.4.) can be specified to act on the base side using the **base when** keywords. Within such a base guard predicate `base` is interpreted as a special identifier holding a reference to the base object that is about to be lifted for the sake of a callin method interception (see § 5.4.2.(a)).

(e) Parameter mappings

An expression at the right-hand side of a parameter mapping (parameter in a callin binding (§ 4.4.) or result in a callout binding (§ 3.2.(c))) may use the keyword `base` to refer to the bound base instance. Such usage requires the role method bound in this method binding to be non-static.

(f) Inhibition of modification

In all cases, the `base` reference is immutable, i.e., `base` can never appear as the left-hand-side of an assignment.

(g) Decapsulation via base reference

In cases § 2.6.(d) and § 2.6.(e) above, members of the base object may be accessed that would not be visible under Java's visibility rules. Such references are treated as decapsulation in accordance with § 3.4.(a) and § 3.5.(e).
Note that accessing a base field via `base` only gives reading access to this field.

▷ Advanced structures § 2.7.

This section discusses how role containment and the playedBy relationship can be combined. It does not define new rules, but illustrates rules defined above. The central idea is that any class can have more than one of the three flavors *team*, *role*, and *base*.

(a) Nesting

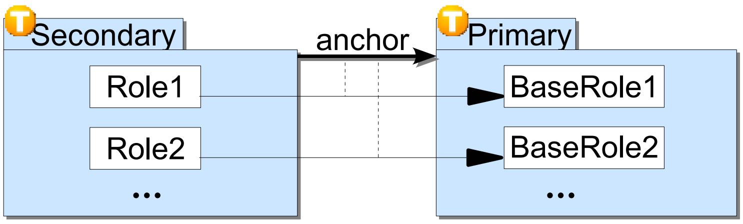
If a role (contained in a team) is also a team (marked with the `team` modifier) it is a **nested team**. The depth of nesting is not restricted.

(b) Stacking

If the base class to which a role is bound using `playedBy` is a team, the role is said to be **stacked** on the base team.

(c) Layering

If roles of a team `Secondary` are played by roles of another team `Primary` (i.e., base classes are roles), the team `Secondary` defines a **layer** over the team `Primary`. Such layering requires a final reference `anchor` from `Secondary` to an instance of `Primary`. All playedBy declarations within `Secondary` specify their base classes anchored to that final link `anchor`.



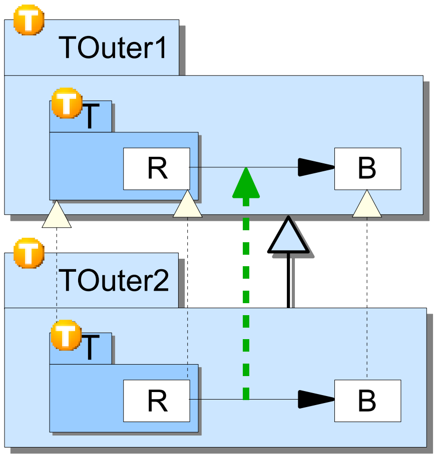
Due to the anchored base types, layered teams implicitly support the following guarantee: all base objects of roles of `Secondary` are contained within the team instance specified by the link `anchor`. If roles of `Secondary` contain any callin bindings to non-static base methods, these will be triggered only when a base method is invoked on a base instance contained in the team specified by `anchor`.
In accordance with § 2.6.(a) the anchor in such anchored playedBy declarations could also be the pseudo identifier `base`, provided that `Secondary` is a nested team, which has a playedBy binding to `Primary` as its base class. This situation is part of the second example below (§ 2.7.(d)) (see `T1 playedBy TB1`).

(d) Implicit playedBy specialization

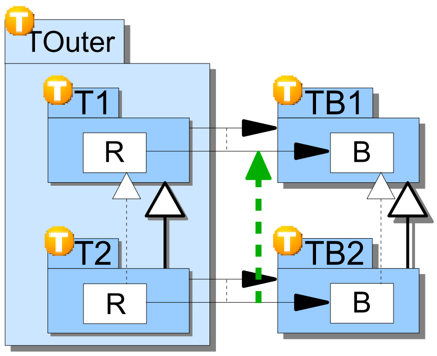
According to § 2.1.(d) an implicit sub-role may *implicitly* specialize an existing `playedBy` relation. This requires the base class to be specified relative to some implicit (`OuterTeam.this`) or explicit (`OuterTeam.base`) team anchor. Specializing that team anchor automatically specializes the playedBy declaration, too. This rule never requires any action from a programmer but only explains the interpretation of a playedBy declaration in complex situations.

Two advanced examples demonstrating the above are:

- If a role `TOuter1.T.R` of a **nested team** `TOuter1.T` is played by another role of the outer enclosing team `TOuter1.B`, subclassing the outer team `TOuter1` to `TOuter2` will produce a new role `TOuter2.T.R` which is automatically played by `TOuter2.B`, an implicit sub class of the original base class `TOuter1.B`.



- Consider the case where a **nested** `T1` as a role of `TOuter` is **stacked** on a base team `TB1`. Also, `T1` is a **layered team** over `TB1` because its role `R` adapts role `TB1.B`.
  In this situation the playedBy relation of role `TOuter.T1.R` is given by a base-anchored type `B<@T1.base>`. If furthermore `TOuter.T1` is subclassed to `TOuter.T2` which covariantly refines the inherited playedBy declaration to `TB2`, then `TOuter.T2.R` will automatically refine the inherited playedBy relation to `TB2.B` to follow the new interpretation of the `base` anchor.



▷ Callout Binding § 3.

Notion of callout binding

callout binding A callout binding declares that a method call to a role object may be **forwarded** to a base method of the associated base object *(the role object "calls out" to the base)*.

declarative completeness Even if a role class does not implement all needed methods, but forwards some to its base, also these methods must be declared within the role. Secondly, no forwarding occurs, unless explicitly declared by a callout binding.

expected/provided A callout binding binds an **expected** method of the role class (needed but not implemented here) to a **provided** method of the base class.

▷ Callout method binding § 3.1.

→ Syntax § A.3.2

A role class may acquire the implementation for any of its (expected) methods by declaring a **callout** binding.

(a) Prerequisite: Class binding

A callout binding requires the enclosing class to be a role class bound to a base class according to § 2.1..

(b) Definition

A callout binding maps an abstract role method ("expected method") to a concrete base method ("provided method"). It may appear within the role class at any place where feature declarations are allowed. It is denoted by

expected_method_designator −> *provided_method_designator;*

The effect is that any call to the role method will be forwarded to the associated base object using the provided base method.

Listing 13: Example code (Callout):

```
team class Company {
  public class Employee playedBy Person {
    abstract String getIdentification();
    // callout binding see below...
  }
}
```

(c) Kinds of method designators

A method designator may either be a method name

```
getIdentification -> getName;
```

or a complete method signature including parameter declarations and return type declaration, but excluding any modifiers and declared exceptions.

```
4 String getIdentification() -> String getName();
```

Effects:

- Line 4 declares a callout binding for the role method `getIdentification()`, providing an implementation for the abstract method defined in line 3.
- In combination with the role binding in line 2 this has the following effect:
- Any call to `Employee.getIdentification` is forwarded to the method `Person.getName`.

Both sides of a callout binding must use the same kind of designators, i.e., designators with and without signature may not be mixed.
Each method designator must uniquely select one method. If a method designator contains a signature this signature must match exactly with the signature of an existing method, i.e., no implicit conversions are applied for this matching. If overloading is involved, signatures *must* be used to disambiguate.

(d) Inheritance of role method declarations

The role method being bound by a callout may be declared in the same class as the binding or it may be inherited from a super class or super interface.

(e) Callout override

If an inherited role method is concrete, callout binding regarding this method must use the token "=>" instead of "->" in order to declare that this binding overrides an existing implementation.

Using the "=>" operator for an abstract method is an error.
It is also an error (and not useful anyway) to callout-bind a method that is implemented in the same class as the binding.

(f) Inheritance of callout bindings

Callout bindings are inherited along explicit and implicit inheritance. Inherited callout bindings can be overridden using "=>".

(g) Duplicate bindings

It is an error if a role class has multiple callout bindings for the same role method.

(h) Declared exceptions

It is an error if a base method to be bound by **callout** declares in its `throws` clause any exceptions that are not declared by the corresponding role method.

(i) Shorthand definition

A callout binding whose method designators specify full method signatures does not require an existing role method. If no role method is found matching the expected method of such a callout binding, a new method is implicitly generated. The new method is static iff the bound base method is static, and it declares the same exceptions as the bound base method.

A shorthand callout may optionally declare a **visibility modifier**, otherwise the generated method inherits the visibility modifier of the bound base method. No further modifiers are set. If a callout overrides an inherited method or callout, it must not reduce the visibility of the inherited method/callout.

(j) Inferred callout

If a non-abstract role class inherits an abstract method the compiler tries to infer a callout binding for implementing the abstract method. Similarly, if a self-call in a role class cannot be resolved, the compiler tries to infer a callout to resolve the self-call.
Inference searches for a method in the bound base class such that

1. both methods have the same name
2. both methods have the same number of arguments
3. each argument of the abstract role method is compatible to the corresponding argument of the base method directly, or using boxing/unboxing or lowering.

Callouts inferred from an interface have `public` visibility, callouts inferred from a self-call have `private` visibility.
Per default inferred callout bindings are disabled, i.e., a compiler must report these as an error. However, a compiler should allow to configure reporting to produce a warning only (which can be suppressed using a `@SuppressWarnings("inferredcallout")` annotation), or to completely ignore the diagnostic.

A callout binding either attaches an implementation to a previously declared method or adds (§ 3.1.(i) above) a forwarding method to a role class. Apart from this implementation, callout-bound methods do not differ from regular methods.

When we say, a callout binding defines **forwarding** this means that control is passed to the base object. In contrast, by a **delegation** semantics control *would* remain at the role object, such that self-calls would again be dispatched starting at the role. Callout bindings on their own do not support delegation. However, in conjunction with method overriding by means of callin bindings (see § 4.) the effect of delegation can easily be achieved.

▷ Callout parameter mapping § 3.2.

→ Syntax § A.4.1

(a) with clause

If the method designators in a callout binding are signatures (not just method names), parameters and return value may be mapped by a `with{...}` sub-clause. Parameter mappings may only occur if the enclosing role is a class, not an interface.

(b) Mapping one parameter

For each parameter of the provided base method, exactly one parameter mapping defines, which value will actually be passed to the base method. Callout parameter mappings have this form:

expression −> *base_method_parameter_name*

(c) Result mapping

The return value of a callout method may be provided by a result mapping:

```
result <- expression
```

The right hand side expression of a result mapping may use the special identifier `result` to refer to the value returned by the base method.
In a method binding with parameter mappings, it is an error to use `result` as the name of a regular method argument.

Listing 14: Example code (Callout Parameter Mapping):

```
Integer absoluteValue(Integer integer) -> int abs(int i) with {
  integer.intValue() -> i,
  result <- new Integer(result)
}
```

(d) Visible names

Each identifier that appears within the expressions of a parameter mapping must be either:

- a feature visible in the scope of the role instance.
- a parameter of the role method (for parameter mappings).
- the special name `result` (for result mappings).
- in a result mapping also the special name `base` can be used in order to refer to the bound base instance (provided the method being bound is not static).

The names of base method arguments (i.e., names after mapping) are only legal in the position given in § 3.2.(b).

(e) Implicit parameter mappings

If parameter mappings should be omitted the following conditions must hold:

1. each method parameter of the role method must conform to the corresponding parameter of the base method, *and*
2. the result type of the base method must conform to the result type of the role method.

Here conformance includes translation polymorphism (cf. § 3.3.(d)).
Parameter correspondence without parameter mapping is determined by declaration order not by names.
Two adjustments can, however, be performed implicitly:

- If the role method has more parameters than the base method, unused trailing parameters may be silently ignored.
- If the role method returns `void`, any result from the base method may be silently ignored.

Listing 15: Example code (Callout with Parameter Mapping):

```
1 public team class MyTeamA {
2   public abstract class Role1 {
3     abstract void payEuro(float euro);
4     abstract float earnEuro();
5     void idle(int seconds) { /* do nothing */ };
6   }
7   Role1 boss, worker = // initialization omitted
8   public void transaction () {
9     boss.payEuro(worker.earnEuro());
10    boss.idle(123);
11  }
12 }
13 public class Staff { // a base class
14   public void payDM (float dm) { ... };
15   public float earnDM () { ... };
16   public int doze() { ... };
17   // other methods omitted
18 }
19 public team class MySubTeam extends MyTeamA {
20   public class Role1 playedBy Staff {
21     void payEuro(float euro) -> void payDM(float dm) with {
22       euro * 1.95583f -> dm
23     }
24     float earnEuro() -> float earnDM () with {
25       result <- result / 1.95583f
26     }
27     idle => doze; // override existing implementation of idle()
28   }
29   void doit() {
30     transaction();
31   }
32 }
```

Effects:

- Class `MyTeamA` is declaratively complete and can be type checked because it only uses methods that are visible or declared within this context. `MyTeamA.Role1` can, however, not be instantiated, because it is abstract.
- Line 30 has the normal effect of invoking `transaction`.
- When executing `transaction`, the call of `worker.earnEuro()` is forwarded to the corresponding base object using method `earnDM()` (binding declaration in line 24). The result is converted by "`result / 1.95583f`" (line 25).
- Within the same execution of `transaction`, the call of `boss.payEuro()` is forwarded to the corresponding base object using method `payDM()` (binding declaration in line 21). The parameter `euro` is converted by "`euro * 1.95583f`" (line 22).
- Method `idle` is forwarded to `doze` without any parameter mapping. This requires `doze` to have a signature that is conformable to the signature of `idle`. In this case a role parameter and a base result are ignored.
  Using the => operator, this binding overrides the existing implementation of `idle`.

§ 3.3. Lifting and lowering

(For basic definitions see § 2.2. and § 2.3.)

(a) Call target translation

Invoking a base method due to a callout binding first **lowers** the role object in order to obtain the effective call target.

(b) Parameter translation

Passing a role object as parameter to a callout method implicitly **lowers** this parameter, if the base method declares a corresponding base type parameter.
Lifting of callout parameters is not possible.

(c) Result translation

When returning a base object from a callout method where the role method declares the result to be of a role class, this object is implicitly **lifted** to the appropriate role.
Lowering the result of a callout binding is not possible.

(d) Typing rules

A parameter mapping (implicit by parameter position or explicit by a `with` clause) is **well typed** if the left hand side conforms to the right hand side, either by

- type equality
- implicit primitive type conversion
- subtype polymorphism
- translation polymorphism, here: *lowering*,
- *or* by a combination of the above.

A result mapping (implicit or explicit by a `with` clause) is well typed, if the value at the right hand side conforms to the left hand side according to the rules given above, except that translation polymorphism here applies *lifting* instead of lowering.

(e) Role arrays

For arrays of roles as parameters § 2.2.(e) applies accordingly. For arrays as a return value § 2.3.(d) applies.

§ 3.4. Overriding access restrictions

In contrast to normal access restrictions, method bindings may refer to hidden base methods. This concept is the inverse of encapsulation, hence it is called **decapsulation**.
Decapsulation may occur in these positions:

- `playedBy` declaration (see § 2.1.2.(c))
- base constructor call (see § 2.4.2.(b)).
- callout bindings (see next)
- callout to field (see § 3.5.(e))
- base call within a callin method (see § 4.6.)

(a) Callout to inaccessible base method

By means of **callout** bindings it is possible to access methods of a base class regardless of their access modifiers. Method bindings are the only place in a program which may mention otherwise inaccessible methods. Access to the callout method at the role side is controlled by regular mechanisms, based on the declaration of the role method.

(b) Sealing against decapsulation

A base package may be "sealed" which re-establishes the standard Java visibility rules.
Sealing is achieved by the corresponding capability of Jar files.

(c) Warning levels

A compiler should signal any occurrence of decapsulation. If a compiler supports to configure warnings this may be used to let the user choose to (a) ignore base class decapsulation, (b) treat it as a warning or even (c) treat it as an error (cf. § 2.1.2.(c)).
Optionally, a batch compiler may support three levels of verbosity with respect to decapsulation:

`-nodecapsulation`	No warnings.
default	Warn only if/that access restrictions are overridden.
`-decapsulation`	Detailed messages containing the binding and the hidden base method.

(d) Private methods from super classes

If a callout binding shall bind to a private base method, that method must be defined in the exact base class to which the current role class is bound using `playedBy`. I.e., for private methods § 3.1.(d) does not hold.
The same holds for private base fields (see below).
If a private base feature must indeed be callout-bound, a role class must be defined that is played by the exact base class defining the private feature. Another role bound to a sub-base-class can then be defined as a sub class of the first role. It will inherit the callout binding and through this it can access the desired feature.

```
public class SuperBase {
  private int secret;
}
public class SubBase extends SuperBase { /* details omitted */ }
public team class MyTeam {
  protected class SuperRole playedBy SuperBase {
    int steal() -> get int secret; // OK
  }
  protected class SubRole extends SuperRole playedBy SubBase {
    int steal() -> get int secret; // illegal!
  }
}
```

▷ Callout to field § 3.5.

Also fields of a base class can be made accessible using a callout binding.

(a) Syntax

Using one of the callout modifiers `get` or `set` a role method can be bound to a field of the role's base class:

```
1 getValue -> get value;
2 setValue -> set value;
3 int getValue() -> get int value;
```

where `getValue`, `setValue` are abstract role methods of appropriate signatures and `value` is a field of the bound base class.
A longer syntax is available, too (see line 3 above), which uses complete signatures. For the left hand side § 3.1.(c) applies, for the right hand side, this longer version prepends the field type to the field name.

(b) Compatibility

A role method bound with the modifier `get` should have no arguments (it *may* have arbitrary arguments, which are silently ignored) and should have a return type to which the base field is compatible. A role method returning void will ignore the given value and thus has no effect at all, which will be signaled by a compiler warning.
A role method bound with the modifier `set` must have a first argument that is compatible to the base field's type (additional arguments - if present - are silently ignored) and must not declare a return type.

(c) Value mapping

Values can be mapped similar to parameter mappings in pure method bindings (§ 3.2.). Such mappings can be used to establish compatibility as required above.
In both `get` and `set` bindings, the base side value is denoted by the field's name (lines 2 and 4 below).

```
1 Integer getValue()          -> get int val
2   with { result             <- new Integer(val) }
3 void setValue(Integer i)    -> set int val
4   with { i.intValue()       -> val }
```

(d) Effect

Callout-binding a role method to a base field generates an implementation for this role method, by which it acts as a getter or setter for the given field of the associated base object.

(e) Access control

For accessing an otherwise invisible field, the rules for decapsulation (§ 3.4.) apply accordingly.
Recall, that according to JLS §8.3[18] fields may be hidden in sub-classes of a given base class. Therefore, it is relevant to know that a callout to a field will always access the field that is visible in the exact base class of the role class defining the callout. This is especially relevant for accessing private fields.

[18] `http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#40898`

(f) Shorthand definition

Just as in § 3.1.(i) a shorthand definition allows to introduce a callout field access method without prior abstract declaration. This requires the callout field binding to specify types as in line 3 of § 3.5.(a) above. The generated access method is static iff the bound base field is static.

A shorthand callout to field may optionally declare a **visibility modifier**, otherwise the generated method inherits the visibility modifier of the bound base field. No further modifiers are set. If a callout to field overrides an inherited method or callout, it must not reduce the visibility of the inherited method/callout.

(g) Callout override

Similar to method callouts a callout to field may override an existing role method if and only if the token => is used instead of -> (see § 3.1.(e) and § 3.1.(f)).

(h) Inferred callout

If a statement or expression within the body of a bound role class uses a simple name or a name qualified by `this` which can not be resolved using normal rules, the compiler may infer to use a callout to field instead, given that a field of the required name can be found in the role's declared baseclass.

If a callout to field has explicitly been declared it is used for the otherwise unresolved name, if and only if:

- the callout declares a role method name the is constructed from the token "set" for a setter or "get" for a getter plus the field name with capital first letter,
- the base field referenced by the callout has exactly the required name, and
- the callout kind (set/get) matches the application of the unresolved name as either the left-hand side of an assignment (set) or as an expression (get).

If a matching callout to field is not found, the compiler generates one automatically, which has `private` visibility.

If a callout to field has been inferred it is an error to directly invoke the implicitly generated callout accessor that is formed according to the above rules.

Per default inferred callout bindings are disabled, i.e., a compiler must report these as an error. However, a compiler should allow to configure reporting to produce a warning only (which can be suppressed using a `@SuppressWarnings("inferredcallout")` annotation), or to completely ignore the diagnostic. See also § 3.1.(j).

▷ Callin Binding § 4.

Notion of callin binding

Callin bindings realize a forwarding in the direction opposite to callout bindings (see § 3.). Both terms are chosen from the perspective of a role, which controls its communication with an associated base object. Technically, callin bindings are equivalent to weaving additional code (triggers) into existing base methods.

Callin Methods of a base class may be **intercepted** by a callin binding (*the base method "calls into" the role*).

Before/after/replace The modifiers **before, after, replace** control the composition of original method and callin method.

Activation Callin bindings may be active or inactive according to § 5..

▷ Callin method binding § 4.1.

→ Syntax § A.3.3

(a) Method call interception

A role method may intercept calls to a base method by a callin binding.

(b) Prerequisite: Class binding

A callin binding requires the enclosing class to be a role class bound to a base class according to § 2.1.. An *unliftable* role (see § 2.3.4.(a)) cannot define callin bindings. In that case callin bindings can only be introduced in sub-roles which (by an appropriately refined `playedBy` clause) disambiguate the lifting translation.

(c) Callin declaration

A callin binding composes an existing role method with a given base method. It may appear within the role class at any place where feature declarations are allowed. It is denoted by

role_method_designator <– callin_modifier base_method_designator;

Just like with callout bindings, method designators may or may not contain parameters lists and return type but no modifiers; also, each method designator must exactly and uniquely select one method (cf. § 3.1.(c)).
For *callin modifiers* see below (§ 4.2.).

(d) Multiple base methods

Base method designators may furthermore enumerate a list of methods. If multiple base methods are bound in one callin declaration generally all signatures in this binding must be conform.
However, *extraneous parameters* from base methods may be ignored at the role.
For *result types* different rules exist, depending on the applied callin modifier (see next).

(e) Named callin bindin

Any callin binding may be labeled with a name. The name of a callin binding is used for declaring *precedence* (§ 4.8.). A named callin binding *overrides* any inherited callin binding (explicit and implicit (§ 1.3.1.)) with the same name.
It is an error to use the same callin name more than once within the same role class.

(f) Callin to final

When binding to a final base method, the enclosing role must be played by the exact base class declaring the final method. I.e., callin binding to a final method inherited from the base class's super-class is not allowed. This is motivated by the fact that no sub-class may have a version of a final method with different semantics.

(g) Declared exceptions

It is an error if a role method to be bound by callin declares in its `throws` clause any exceptions that are not declared by the corresponding base method(s).

§ 4.2. Callin modifiers (before, after, replace)

(a) Method composition

The kind of method composition is controlled by adding one of the modifiers **before, after** or **replace** after the "<-" token of the binding declaration.

(b) Additive composition

The `before` and `after` modifiers have the effect of adding a call to the role method at the beginning or end of the base method, resp.
In this case no data are transferred from the role to the base, so if the role method has a result, this will always be ignored.

Listing 16: Example code (Callin):

```
1 team class Company {
2   protected class Employee playedBy Person {
3     public void recalculateIncome() { ... }
4     recalculateIncome <- after haveBirthday; // callin binding
5   }
6 }
```

Line 4 declares a callin binding for the role method `recalculateIncome()` defined in line 3.
In combination with the role binding in line 2 this has the following effect:

- **After** every call of the method `Person.haveBirthday` the method `Company.recalculateIncome` is called.

(c) Replacing composition

The `replace` modifier causes *only* the role method to be invoked, replacing the base method.
In this case, if the base method declares a result, this should be provided by the role method. Special cases of return values in callin bindings are discussed in § 4.3.(e)

(d) Callin methods

Role methods to be bound by a callin replacement binding must have the modifier `callin`. This modifier is only allowed for methods of a role class.
A method with the `callin` modifier can only be called

- via a callin replace binding
- by a `super` or `tsuper` call from an overriding callin method.

It is illegal for a `callin` method

- to be called directly,
- to be bound using a callout binding, and
- to be bound to a base method using a `before` or `after` callin binding.

Despite these rules a second level role — which is played by the current role — can intercept the execution of a callin method using any form of callin binding.
A callin method cannot override a regular method and vice versa, however, overriding one callin method with another callin method is legal and dynamic binding applies to callin method just like regular methods.
A callin method must not declare its visibility using any of the modifiers `public`, `protected` or `private`. Since callin methods can only be invoked via callin bindings such visibility control would not be useful.

▷ Base calls § 4.3.

→ Syntax § A.5.3

Role methods with a `callin` modifier should contain a *base call* which uses the special name `base` in order to invoke the original base method (original means: before replacement).

(a) Syntax

The syntax for base calls is `base.m()`, which is in analogy to super calls. A `base.m()` call must use the same name and signature as the enclosing method. This again follows the rule, that roles should never explicitly use base names, except in binding declarations.

(b) Missing base call

For each callin method, the compiler uses some flow analysis to check whether a base call will be invoked on each path of execution (analysis is very similar to the analysis for definite assignment regarding final variables - JLS §16[19]). The compiler will issue a warning if a base call is missing either on each path (definitely missing) or on some paths (potentially missing). Instead of directly invoking a base call, a callin method may also call its explicit or implicit super version using `super.m()` or `tsuper.m()` (see § 1.3.1.(f)). In this case the flow analysis will transitively include the called super/tsuper version.

(c) Duplicate base call

If a callin method contains several base calls, the compiler gives a warning if this will result in duplicate base call invocations on all paths (definitely duplicate) or on

[19] `http://java.sun.com/docs/books/jls/third_edition/html/defAssign.html`

some paths (potentially duplicate). Again super/tsuper calls are included in the flow analysis (see 4.3(b)).

(d) Parameter tunneling

If a base method has more parameters than a callin method to which it is composed, additional parameters are implicitly passed unchanged from the original call to the base call (original means: before interception). I.e., a call `base.m()` may invisibly pass additional parameters that were provided by the caller, but are hidden from the role method.

(e) Fragile callin binding

If a role method returns void, but the bound base method declares a non-void result, this is reported as a *fragile callin binding*: The result can still be provided by the base call, but omitting the base call may cause problems depending on the return type:

- For reference return types `null` will be returned in this case.
- In the case of primitive return types this will cause a `ResultNotProvidedException` at run-time.

It is an error if a callin method involved in a fragile callin binding has definitely no base call.

(f) Base super calls

If a callin method `rm` is bound to a base method `B1.m` that in turn overrides an inherited method `B0.m` (`B0` is a super class of `B1`), the callin method may use a special form of a base call denoted as

```
base.super.rm();
```

Such base super call invokes the super method of the bound base method, here `B0.m`. This invocation is not affected by any further callin binding.
A base super call bypasses both the original method `B1.m` and also other callin bindings that would be triggered by a regular base call. For this reason any application of this construct is flagged by a decapsulation warning (see § 3.4.).

Comment:
Base calls can occur in callin methods that are not yet bound. These methods have no idea of the names of base methods that a sub-role will bind to them. Also multiple base methods may be bound to the same callin method. Hence the use of the role method's own name and signature. The language implementation translates the method name and signature back to the base method that has originally been invoked.

Listing 17: Example code (Base Call):

```
1 public class ValidatorRole playedBy Point {
2   callin void checkCoordinate(int value) {
3     if (value < 0)
4       base.checkCoordinate(-value);
5     else
6       base.checkCoordinate(value);
7   }
8   checkCoordinate <- replace setX, setY;
9 }
```

Effects:

- Line 2 defines a callin method which is bound to two methods of the base class `Point` (see line 8).
- The value passed to either setX or setY is checked if it is positive (line 3).
- Lines 4 and 6 show calls of the original method (base calls). While line 6 passes the original value, in the negative case (line 4) the passed value is made positive.

▷ Callin parameter mapping § 4.4.

(a) General case parameter mapping

The rules for mapping callin parameters and result type are mainly the same as for callout bindings (§ 3.2.) except for reversing the `->` and `<-` tokens and swapping left hand side and right hand side.
Callin bindings using `before` have no result mapping. For `result` in `after` callin bindings see § 4.4.(c) below.

(b) Restrictions for callin replace bindings

The right-hand side of a parameter mapping may either be the simple name of a base method argument without further computation, or an arbitrary expression *not* containing any base method argument.
Each base method argument must either appear as a simple name in exactly one parameter mapping or not be mapped at all. In the latter case, the original argument is "tunneled" to the base call, meaning, the callin method does not see the argument, but it is passed to the base method as expected.
If the base method declares a result, then

- if the role method also declares a result, `result` must be mapped to itself:
  `result -> result`
- if the role method does not declare a result, an arbitrary expression may be mapped to result:
  `expression` `-> result`
  If in this situation no result mapping exists, the result of the base call is "tunneled" and passed to the original caller (see fragile callin binding (§ 4.3.(e)) above).

These rules ensure that these bindings are reversible for the sake of base calls (§ 4.3.).

As stated above a fragile callin binding (§ 4.3.(e)) is not allowed with a callin method that definitely has no base call (§ 4.3.(b)). A callin replace binding is not fragile if it provides the base result using a result mapping.
A callin method bound with replace to a base method returning void must not declare a non-void result.

(c) Mapping the result of a base method

In an `after` callin binding, the right-hand side of a parameter mapping may use the identifier `result` to refer to the result of the base method.

(d) Multiple base methods

A callin binding listing more than one base method may use parameter mappings with only the following restriction: if any base parameter should be mapped this parameter must have the same name and type in all listed base method designators. However, different parameter mappings for different base methods bound to the same role method can be defined if separate callin bindings are used.

§ 4.5. Lifting and lowering

For basic definition see § 2.2. and § 2.3..
(The following rules are reverse forms of those from § 3.3.)

(a) Call target translation

Invoking a role method due to a callin binding first **lifts** the base object to the role class of the callin binding, in order to obtain the effective call target. This is why callin bindings cannot be defined in roles that are *unliftable* due to *potential binding ambiguity* (see § 4.1.(b) above and § 2.3.4.(a)).

(b) Parameter translation

During callin execution, each parameter for which the role method expects a role object is implicitly **lifted** to the declared role class.

(c) Result translation

Returning a role object from a callin method implicitly **lowers** this object.

(d) Typing rules

A parameter mapping (implicit by parameter position or explicit by a `with` clause) is **well typed** if the right hand side conforms to the left hand side, either by

- type equality
- implicit primitive type conversion
- subtype polymorphism
- translation polymorphism, here: *lifting*;
  however, within `replace` bindings step 1 of the smart lifting algorithm (§ 2.3.3.(a)) is not applicable
- *or* by a combination of the above.

A result mapping (implicit or explicit by a `with` clause) is well typed, if the value at the left hand conforms to the right hand side according to the rules given above, except that translation polymorphism here applies *lowering* instead of lifting.
These rules define **translation polymorphism** as introduced in § 2.3..
Additionally, in a `replace` callin binding compatibility of parameters and return types must hold in both directions. Thus, from the above list of conversions a replace binding cannot apply subtype polymorphism nor primitive type conversion. If more flexibility is desired, type parameters can be used as defined in § 4.10..

(e) Role arrays

For arrays of roles as parameters § 2.3.(d) applies accordingly. For arrays as return value § 2.2.(e) applies.

(f) Base calls

For base calls these rules are reversed again, i.e., a base call behaves like a callout binding.

▷ Overriding access restrictions § 4.6.

Callin bindings may also mention inaccessible methods (cf. decapsulation § 3.4.). Due to the reverse call direction this is relevant only for base calls within `callin` methods. Base calls have unrestricted access to protected base methods. Accessing a base method with private or default visibility is also allowed, but signaled by a compiler warning.

Comment:
A base call to an inaccessible base method is considered harmless, since this is the originally intended method execution.

(a) Private methods from super classes

(Cf. § 3.4.(d)) If a callin binding shall bind to a private base method, that method must be defined in the exact base class to which the current role class is bound using `playedBy`.
If a private base feature must indeed be callin-bound, a role class must be defined that is played by the exact base class defining the private feature. Another role bound to a sub-base-class can then be defined as a sub class of the first role. It will inherit the callin binding and through this it can access the desired feature.

▷ Callin binding with static methods § 4.7.

The normal case of callin bindings refers to non-static methods on both sides (base and role). Furthermore, in Java inner classes can not define static methods. Both restrictions are relaxed by the following rules:

(a) Static role methods

A role class may define static methods (see also § 1.2.1.(f)).

(b) Binding static to static

A callin binding may bind a static role method to one or more static base methods. It is, however, an error to bind a static base method to a non-static role method, because such binding would require to lift a base object that is not provided.

(c) before/after

In addition to the above, `before` and `after` callin bindings may also bind a static role method to non-static base methods.

(d) replace

In contrast to § 4.7.(c) above, a `replace` callin binding cannot bind a static role method to a non-static base method.

The following table summarizes the combinations defined above:

	<-	**base method**	
		static	non-static
role method	static	OK	`before/after`: OK `replace`: illegal
	non-static	illegal	OK

(e) No overriding

Since static methods are not dynamically bound, *overriding* does not apply in the normal semantics. Regarding callin bindings this has the following consequences (assuming a role `RMid` played by `BMid` plus its super-class `BSuper` and its sub-class `BSub`.

1. If a static base method `BMid.m` is bound by a callin binding this has no effect on any method `m` in `BSub`.
2. If a callin binding mentions a method `m` which is not present in `BMid` but resolves to a static method in `BSuper` the binding only affects invocations as `BMid.m()` but not `BSuper.m()`. If the latter call should be affected, too, the callin binding must appear in a role class bound to `BSuper`, not `BMid`.
3. In order to bind two static base methods with equal signatures, one being defined in a sub-class of the other one, two roles have to be defined where one role refines the `playedBy` clause of the other role (say: `public class RSub extends RMid playedBy BSub`). Now each role may bind to the static base method accessible in its direct base-class.

§ 4.8. Callin precedence

→ Syntax § A.8

If multiple callins from the same team refer to the same base method and also have the same callin modifier (`before`, `after` or `replace`), the order in which the callin bindings shall be triggered has to be declared using a precedence declaration.

(a) Precedence declaration

A precedence declaration consists of the keyword `precedence` followed by a list of names referring to callin bindings (see § 4.1.(e) for named callin bindings).
A precedence declaration is only legal within a role or team class.

(b) Qualified and unqualified names

Within a role class a callin binding may be referenced by its unqualified name. A precedence declaration in a team class must qualify the callin name with the name of the declaring role class. A team with nested teams may concat role class names. Elements of a qualified callin name are separated by ".".
The callin binding must be found in the role specified by the qualifying prefix or in the enclosing role for unqualified names, or any super class of this role (including implicit super classes § 1.3.1.).

(c) Class based precedence

At the team level a precedence declaration may contain role class names without explicitly mentioning callin bindings in order to refer to all callin bindings of the role.

(d) Multiple precedence statements

All precedence statements are collected at the outer-most team. At that level all precedence declarations involving the same base method are merged using the C3 algorithm [3] (see p. 62). It is an error to declare incompatible precedence lists that cannot be merged by the C3 algorithm.

(e) Binding overriding

Precedence declarations may conflict with overriding of callin bindings (see § 4.1.(e)): For each pair of callin bindings of which one callin binding overrides the other one, precedence declarations are not applicable, since dynamic binding will already select exactly one callin binding.
It is an error to *explicitly mention* such a pair of overriding callin bindings in a precedence declaration.
When a class-based precedence declaration *implicitly refers to* a callin binding that is overridden by, or overrides any other callin binding within the same precedence declaration, this does not affect the fact, that the most specific callin binding overrides less specific ones.

Listing 18: Callin binding example

```
public class LogLogin playedBy Database {
  callin void log (String what) {
    System.out.println("enter " + what);
    base.log(what.toLowerCase());
    System.out.println("leave " + what);
  }
  void log(String what) <- replace void login(String uid, String passwd)
    with { what <- uid }
}
(new Database()).login("Admin", "Passwd");
```

Effects: Provided the callin bindings are active (cf. § 5.) then:

- the call in line 10 is intercepted by method `log` of role `LogLogin`.
- the call target of `log` is a role of type `LogLogin` which is created by lifting the original call target (of type `Database`) to `LogLogin`.
- only parameter `uid` is passed to `log` (bound to formal parameter `what`).
- within method `log` the base call (line 4) invokes the original method passing a modified uid (converted to lower case, cf. line 4) and the unmodified password, which is hidden from the callin method due to the parameter mapping in line 8.

§ 4.9. Callin inheritance

This section defines how callin bindings and callin methods relate to inheritance.

§ 4.9.1. Base side inheritance

Generally, a callin binding affects all sub-types of its bound base. Specifically, if a role type `R` bound to a base type `B` defines a callin binding `rm <-` *`callin_modifier`* `bm`, the following rules apply:

(a) Effect on sub-classes

The callin binding also effects instances of any type `BSub` that is a sub-type of `B`. If `BSub` overrides the bound base method `bm`, the overridden version is generally affected, too. However, if `bm` covariantly redefines the return type from its super version, the callin binding has to explicitly specify if the covariant sub-class version should be affected, too (see § 4.9.3.(b)).

(b) No effect on super-classes

The binding never affects an instance of any super-type of `B` even if the method `bm` is inherited from a super-class or overrides an inherited method. This ensures that dispatching to a role method due to a callin binding always provides a base instance that has at least the type declared in the role's `playedBy` clause.

For corresponding definitions regarding static methods see § 4.7.(e).

§ 4.9.2. Role side inheritance

Any sub-type of `R` inherits the given callin binding (for overriding of bindings see § 4.8.(e)). If the sub-role overrides the role method `rm` this will be considered for dynamic dispatch when the callin binding is triggered.

§ 4.9.3. Covariant return types

Since version 5, Java supports the covariant redefinition of a method's return type (see JLS 8.4.5[20]). This is *not* supported for `callin` methods (§ 4.9.3.(a)). If base methods with

[20] `http://java.sun.com/docs/books/jls/third_edition/html/classes.html#8.4.5`

covariant redefinition of the return type are to be bound by a callin binding the subsequent rules ensure that type safety is preserved. Two *constraints* have to be considered:

1. When a callin method issues a base-call or calls its tsuper version, this call must produce a value whose type is compatible to the enclosing method's declared return type.
2. If a replace-bound role method returns a value that is not the result of a base-call, it must be ensured that the return value actually satisfies the declared signature of the bound base method.

(a) No covariant callin methods

A method declared with the `callin` modifier that overrides an inherited method must not redefine the return type with respect to the inherited method. This reflects that fact that an inherited callin binding should remain type-safe while binding to the new, overriding role method. Binding a covariant role method to the original base method would break constraint (1) above.

(b) Capturing covariant base methods

If a callin binding should indeed affect not only the specified base method but also overriding versions which covariantly redefine the return type, the binding must specify the base method's return type with a "+" appended to the type name as in

```
void rm() <- before RT+ bm();
```

Without the "+" sign the binding would only capture base methods whose return type is exactly `RT`; by appending "+" also sub-types of `RT` are accepted as the declared return type.

(c) Covariant replace binding

When using the syntax of § 4.9.3.(b) to capture base methods with covariant return types in a callin binding with the `replace` modifier, the role method must be specified using a free type parameter as follows:

```
<E extends RT> E rm() <- replace RT+ bm();
```

The role method `rm` referenced by this callin binding must use the same style of return type using a type parameter. The only possible non-null value of type `E` to be returned from such method is the value provided by a base-call or a tsuper-call. This rule enforces the constraint (2) above.
Note that this rule is further generalized in § 4.10..

Listing 19: Binding a parametric role method

```
1  public class SuperBase {
2      SuperBase foo() { return this; }
3      void check() { System.out.print("OK"); }
4  }
5  public class SubBase extends SuperBase {
6      @Override
7      SubBase foo() { return this; }
8      void print() { System.out.print("SubBase"); }
9      String test() {
10         this.foo().print(); // print() requires a SubBase
11     }
12 }

14 public team class MyTeam {
15     protected class R playedBy SuperBase {
16         callin <E extends SuperBase> E ci() {
17             E result= base.ci();
18             result.check(); // check() is available on E via type bound SuperBase
19             return result;
20         }
21         <E extends SuperBase> E  ci() <- replace SuperBase+ foo();
22     }
23 }
```

Explanation:

- Method `SubBase.foo` in line 7 redefines the return type from `SuperBase` (inherited version) to `SubBase`, thus clients like the method call in line 10 must be safe to assume that the return value will always conform to `SubBase`.
- The callin binding in line 21 explicitly captures both versions of `foo` by specifying `SuperBase+` as the expected return type. Thus, if an instance of `MyTeam` is active at the method call in line 10, this call to `foo` will indeed be intercepted even though this call is statically known to return a value of type `SubBase`.
- The callin method in lines 16-20 has a return type which is not known statically, but the return type is represented by the type variable `E`. Since the base call is known to have the exact same signature as its enclosing method, the value provided by the base call is of the same type `E` and thus can be safely returned from `ci`. *Note,* that no other non-null value is known to have the type `E`.
- By specifying `SuperBase` as an upper bound for the type `E` the callin method `ci` may invoke any method declared in type `SuperBase` on any value of type `E`. For an example see the call to `check` in line 18.

*As an aside note that the above example uses type `SuperBase` in an undisciplined way: within role `R` this type is bound using `playedBy` **and** the same type is also used directly (as the upper bound for `E`). This is considered bad style and it is prohibited if `SuperBase` is imported using an base import (§ 2.1.2.(d)). Here this rule is neglegted just for the purpose of keeping the example small.*

§ 4.10. Generic replace bindings

As mentioned in § 4.5.(d) replace bindings do not support subtype polymorphism in either direction. On the other hand, binding several base methods to the same `callin` method may

require some more flexibility if these base methods have different signatures. This is where type parameter come to the rescue to allow for generic `callin` methods and their binding to base methods with different signatures.
Note that this rule is a generalization of rule § 4.9.3.(c).

(a) Fresh type parameter

If a callin method declares a type parameter `<T>` this type `T` can be used for specifying the type of exactly one parameter or the return type. It is not legal to use the same type parameter in more than one position of a `callin` method signature.

(b) Type bounds

The type parameter of a callin binding may be bounded by an upper bound as in `<T extends C>`. In this case `T` can only be instantiated by types conforming to the upper bound `C`.

(c) Generic replace binding

A generic `callin` method according to the above rules is bound using a replace binding that declares the same number of type parameters, where type parameters of the binding and its callin method are identified. If the `callin` method declares bounds for its type parameters so should the replace binding.

(d) Binding to a type parameter

A fresh type parameter can be used to capture arbitrary types in the base methods to be bound. The type parameter may be instantiated differently for each bound base method. By such type parameter instantiation the types in role and base signatures are actually identical, thus satisfying the requirement of two-way substitutability.

Within the body of a generic `callin` method no further rules have to be followed, because the fresh type variable actually guarantees, that the role method cannot replace the original value (initial argument or base-call result) with a different object, because no type exists that is guaranteed to conform to the type parameters. Yet, the type bound allows the role method to invoke methods of the provided object.

Listing 20: Generic replace binding

```
public team class MyTeam {
    protected class R playedBy Figures {
        callin <E extends Shape, F extends Shape> E ci(F arg) {
            E result= base.ci(arg);
            result= arg; // illegal, types E and F are incommensurable
            arg= result; // illegal, types E and F are incommensurable
            int size= arg.getSize(); // getSize() is availabel on F via type bound Shape
            result.resize(size);     // resize() is available on E via type bound Shape
            return result; // only two legal values exist: result and null
        }
        <E extends Shape, F extends Shape>
        E ci(F arg) <- replace Rectangle getBoundingBox(Shape original),
                               Rectangle stretch(Square original);
    }
}
```

Explanation:

These declaration generate two version of the `callin` method `ci`:

1. `Rectangle ci (Shape arg)`
2. `Rectangle ci (Square arg)`

Within the `callin` method the following observations hold:

- Line 5 is illegal for the first signature as `Shape` is not conform to `Rectangle`
- Line 6 is illegal for the second signature as `Rectangle` is not conform to `Square`
- Everything else is type-safe.

Open issues:

The query language for specifying sets of base methods (§ 4.1.(d)) has not been finalized yet. In this version of the OTJLD § 8. acts as a placeholder for the section that will define a join point query language in the future.

References:

[3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington. *A monotonic superclass linearization for Dylan*. OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 69-82, 1996.

▷ Team Activation § 5.

The concept of Activation

Binding activation All **callin** bindings of a team only have effect if the team is **active**. Activation may be caused by explicit statements and also happens implicitly at certain points during program execution.

Guard predicates Callin bindings can further be controlled using guard predicates, which can be attached to roles and teams at different levels. If a guard predicate evaluates to `false`, all affected callin bindings are disabled.

▷ Effect of team activation § 5.1.

Activating a team instance has the effect of enabling all its callin bindings. All effects defined in § 4. apply only if a corresponding team instance is active.
The **order** of team activation controls the order of callin executions. If more than one team intercepts calls to the same base method, the most recently activated team has highest priority in that its before or replace callins are executed first while its after callins are executed last.

▷ Global vs. thread local team activation § 5.1.1.

While **thread local** activation only enables the callin bindings of a team instance for a certain thread, **global** activation activates the team instance for all threads of the application.

▷ Effect on garbage collection § 5.1.2.

Any active team is referenced by internal infrastructure. Thus, a team cannot be reclaimed by the garbage collector while it is active.

▷ Explicit team activation § 5.2.

(a) Activation block

A team can be activated thread local by the block construct

```
within (myTeam) { stmts }
```

If *stmts* has only one statement this can be abbreviated to

```
within (myTeam) stmt
```

In these statements, `myTeam` must denote a team instance. For the time of executing this block, this team instance is activated for the current thread, which has entered the within block.
The `within` block statement guarantees that it leaves the team in exactly the same activation state as it was in when entering this block. This includes the cases of exceptions, meaning that deactivation will also occur if the execution of the block terminates abnormally.

(b) Imperative activation

Each team class inherits methods from the predefined class `org.objectteams.Team` (super class of all team classes) to control team activation disregarding the block structure of the program. The methods `activate()` and `deactivate()` are used to activate and deactivate a team instance for the current thread.
If a team should be de-/activated for another thread this can be done by the methods `activate(Thread aThread)` and `deactivate(Thread aThread)`. In order to achieve global activation for all threads the predefined constant `Team.ALL_THREADS` is passed to these methods (e.g. `activate(Team.ALL_THREADS)`).
Note, that this methods make no guarantees with respect to exceptions.

(c) Multiple and mixed activations

- If `activate()` is invoked on a team instance that has been explicitly activated before, this statement has no effect at all (note the difference in § 5.3.(a) below). The same applies to deactivating an inactive team.
- If a team was already active when entering a `within` block, it will remain active after leaving the block.
- If the team was active on entry of a `within` block and if `deactivate()` is invoked on the same team instance from within the `within` block, leaving the block will re-activate the team.

§ 5.3. Implicit team activation

When the control flow is passed to a team or one of its roles, the team is implicitly activated for the current thread. So a programmer may in general assume, that whenever a role forwards calls to its base object via callout, the callin bindings of the same role will be active at that time. Exceptions to this rule have to be programmed explicitly.

(a) Team level methods

While executing a **team level method**, the target team is always active. Activation is reset to the previous state when leaving the team method, unless the team has been explicitly activated during execution of the team method by a call to `activate()`. Explicit activation is stronger than implicit activation and thus persists after the team level method terminates. Ie., leaving a team level method will never reset an explicit activation.

(b) Methods of externalized roles

Invoking a method on an **externalized role** (see § 1.2.2.) also has the effect of temporary activation of the team containing the role for the current thread. Regarding deactivation the rule of § 5.3.(a) above applies accordingly.

(c) Nested teams

Implicit activation has additional consequences for nested teams (see § 1.5.):

- Implicit activation of a team causes the activation of its outer teams.
- Implicit deactivation of a team causes the deactivation of its inner teams.

Note that among the different mechanisms for activation, `within` is strongest, followed by `(de)activate()`, weakest is implicit activation. In this sense, explicit imperative (de)activation may override the block structure of implicit activation (by explicit activation within a team level method), but not that of a `within` block (by deactivation from a within block).

▷ Guard predicates § 5.4.

→ Syntax § A.7

The effect of callins can further be controlled using so called guard predicates. Guards appear at four different levels:

- callin method binding
- role method
- role class
- team class

Guards can be specified as *regular* guards or `base` guards, which affects the exact point in the control flow, where the guard will be evaluated.

(a) General syntax for guards

A guard is declared using the keyword `when` followed by a boolean expression in parentheses:

when (*predicateExpression*)

Depending on the kind of guard different objects are in scope using special identifiers like `this`, `base`.
Any predicate expression that evaluates to `true` enables the callin binding(s) to which it applies. Evaluation to `false` disables the callin binding(s).

(b) No side effects

A guard predicate should have no side effects. A compiler should optionally check this condition, but inter-procedural analysis actually depends on the availability of appropriate means to mark any method as side-effect free.

(c) Exceptions

A guard predicate should not throw any exceptions. Yet, any exception thrown within a guard predicate cause the guard to evaluate to `false` rather than propagating the exception, meaning that the evaluation of a guard predicate will never interrupt the current base behaviour.
A compiler should flag any checked exception that is thrown within a guard. Such diagnosis should by default be treated as an error, with the option of configuring its severity to warning or ignore.

§ 5.4.1. Regular guards

This group of guards evaluates within the context of a given role. These guards are evaluated *after* a callin target is lifted and *before* a callin bound role method is invoked.

(a) Method binding guards

A guard may be attached to a callin method binding as in:

```
void roleMethod(int ir) <- after void baseMethod(int ib)
  when (ir > MyTeam.this.threshold);
```

Such a guard only affects the callin binding to which it is attached, i.e., this specific callin binding is only effective, if the predicate evaluates to `true`.
The following values are within the scope of the predicate expression, and thus can be used to express the condition:

- The role instance denoted by `this`.
  Features of the role instance can also be accessed relative to `this` with or without explicit qualifying `this`.
- The team instance denoted by a qualified this reference as in `MyTeam.this`.
- If the callin binding includes signatures (as in the example above): Parameters of the role method.
  If parameter mappings are involved, they will be evaluated before evaluating the guard.
  Furthermore, if the modifier of the callin binding is `after`: The result (denoted by the special identifier `result`) of the role method, if it is not `void`.

(b) Method guards

A method guard is similar to a method binding guard, but it applies to all callin method bindings of this method.
A method guard is declared between the method signature and the method body:

```
void roleMethod(int ir)
  when (ir > MyTeam.this.threshold) { body statements }
```

(c) Role level guards

When a guard is specified at the role level, i.e., directly before the class body of a role class, it applies to all callin method bindings of the role class:

```
protected class MyRole
  when (value > MyTeam.this.threshold)
{
    int value;
    other class body declarations
}
```

The following values are within the scope of the predicate expression:

- The role instance denoted by `this` (explicit or implicit, see above). Thus, in the example `value` will be interpreted as a field of the enclosing role.
- The team instance denoted by a qualified this reference as in `MyTeam.this`

(d) Team level guards

A guard specified in the header of a team class may disable the callin bindings of all contained role classes. The syntax corresponds to the syntax of role level guards.
The only value directly available within team level guard is the team instance (denoted by `this`) and its features.

Of course all guards can also access any visible static feature of a visible class.

Even if a guard has no direct effect, because, e.g., a role class has no callin bindings (maybe not even a role-base binding), predicates at such abstract levels are useful, because all predicates are inherited by all sub classes (explicit and implicit).

▷ Base guards § 5.4.2.

The intention behind base guards is to prevent lifting of a callin-target if a guard evaluates to `false` and thus refuses to invoke the callin bound role method. Using base guards it is easier to prevent any side-effects caused by a callin binding, because lifting could cause side-effects at two levels:

- Creating a role on-demand already is a side-effect (observable e.g. by the reflective function `hasRole (§ 6.1.)`)
- Role creation triggers execution of a role constructor (see custom lifting constructor (§ 2.3.1.(c))) which could produce arbitrary side-effects.

Both kinds of side-effects can be avoided using a base guard which prevents unnecessary lifting.

Any guard (5.4.1 (b)-(e)) can be turned into a base guard by adding the modifier `base` as in:

```
protected class MyRole playedBy MyBase
  base when (base.value > MyTeam.this.threshold)
{
    class body declarations
}
```

However, different scoping rules apply for the identifiers that can be used in a base guard:

(a) Base object reference

In all base guard predicates the special identifier `base` can be used to denote the base object that is about to be lifted.

(b) Method binding guards

A base method binding guard may access parameters as passed to the base method.
Parameter mappings are not considered.
Additionally, for `after` callin bindings, the identifier `result` may be used to refer to the result of the base method (if any).

Note:
In order to achieve the same effect of accessing the base method's result, a regular binding guard (not a base guard) must use a suitable parameter mapping (see § 4.4.(c)).

(c) Method guards

In contrast to regular method guards, a *base* guard attached to a role method cannot access any method parameters. See the next item (d) for values that are actually in scope.

(d) Role level guards

Role level base guards may use these values:

- The base instance using the special identifier `base`.
- The team instance using a qualified this references (`MyTeam.this`).

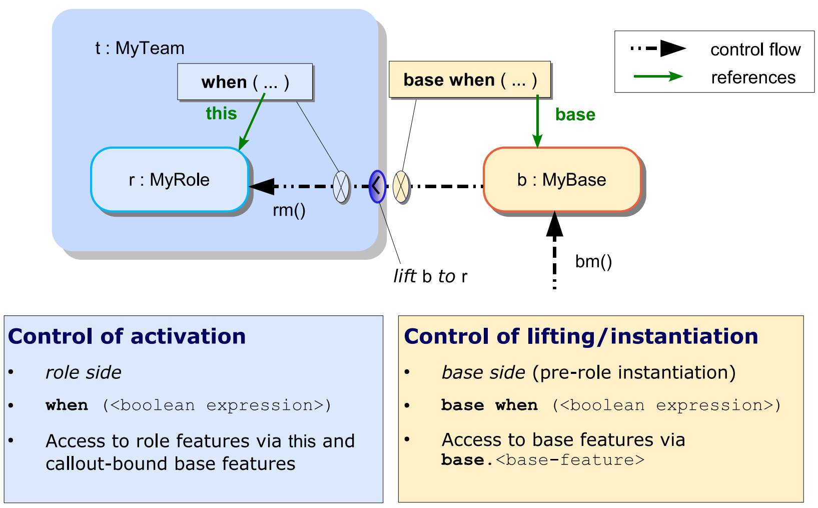
(e) Team level guards

Team level base guards have the same scope as role level base guards (d). However, the type of the role instance is not known here, i.e., here `base` has the static type `java.lang.Object`.

(f) Unbound roles

In contrast to regular guards, base guards cannot be attached to unbound role classes nor to their methods.
Only team level base guards are independent of role binding.

Overview: Guard predicates:



§ 5.4.3. Multiple guards

Due to the different ranges of applicability different guards may affect the same method binding. In that case all applicable guards are conjoined using a logical `and`.
Any guard is interpreted as the conjunction of these predicates (if present):

- The direct predicate expression of the guard.
- The next outer guard along the chain *method binding -> method -> role level -> team level*
- The guard at the same level that is inherited from the *implicit* super role.
- The guard at the same level that is inherited from the *explicit* super role.

Listing 21: Example code (Guard Predicates):

```
1  public team class ATM {
2    private Bank myBank;
3    public class ForeignAccount playedBy Account
4      base when (!ATM.this.myBank.equals(base.getBank()))
5    {
6      callin void debitWithFee(int amount) {
7        base.debitWithFee(fee+amount);
8      }
9      void debitWithFee(int i) <- replace void debit(int amount)
10       base when (amount < 1000);
11   }
12 }
```

Effects:

The team in this example causes that an additional fee has to be payed while debiting less than 1000 Euros from a "foreign" account.

- The base guard in line 4 ensures that `Account` objects only get `ForeignAccount` roles, if they belong to a different bank than the surrounding `ATM` team.
  It accesses the bank of the base via the `base` identifier.
- The method binding guard in line 10 restricts the callin to `debitWithFee` to calls where the base method argument `amount` is lower than 1000.
- A call to `Account.debit` causes a replace callin to `debitWithFee` *only* if *both* predicates evaluate to true.

▷ Unanticipated team activation § 5.5.

If an application should be adapted unanticipatedly by one or more teams, this can be achieved without explicitly changing the program code of this application.

General activation via config file:
Instead of adding the team initialization and activation code to the main program, it is possible to add the respective teams via a config file. Every line of this text file contains the fully qualified name of a compiled team, which has to be available on the classpath. For the instantiation of these teams the default constructor is used, which means adding a team to an application this way requires the team to provide a default constructor. The activation order (see § 5.1.) for these teams corresponds to the order in which they are listed in the config file. Lines starting with a '#' denote comment lines.

Listing 22: Example config file:

```
# Config file for an ObjectTeams application:
mypackage1.MyTeam1
# ...
mypackageM.MyTeamN
```

To get this config file recognized by the application the VM argument
'-Dot.teamconfig=<*config_file_name*>'
has to be used when starting the application.

<u>Note:</u>
In the ObjectTeams Development Tooling (OTDT) teams are activated unanticipatedly via a special tab in the "Run-Configuration" (see OTDT features[21]), instead.

<u>Activation adjustment example:</u>
Teams added via the config file mechanism are activated by default. Because no reference to them is stored anywhere, it is not possible to deactivate them later. If deactivation of unanticipated added teams is required, this can be achieved by adding a manager team via config file and encapsulate the actual functionality in another team managed by the manager team. This way a functional team can be activated and deactivated as needed.

Listing 23: Example code (Activation Adjustment):

```
public team class MyManagerTeam {
  private FunctionalTeam myFunctionalTeam = new FunctionalTeam();
  protected class MyRole playedBy MyApplication {
    void startAdaption() { myFunctionalTeam.activate(); }
    startAdaption <- before startMethod;
    void stopAdaption() { myFunctionalTeam.deactivate(); }
    stopAdaption <- after stopMethod;
  }
}
```

```
# Config file for the manager team example:
MyManagerTeam
```

Effects:

- `startMethod` and `stopMethod` are methods which demand the activation and deactivation respectively.
- If the activation/deactivation depends on other conditions these can be checked in addition.

[21] `http://www.objectteams.org/distrib/features.html#execution`

▷ Object Teams API § 6.

The role of predefined types and methods

Application Programming Interface (API) Some features of ObjectTeams/Java are supported without introducing new syntax but by predefined types and methods.

▷ Reflection § 6.1.

Object Teams supports reflection with respect to teams, roles, and role-base relationships.

(a) Interface to the role registry

Each team instance internally has a registry of known role objects indexed by their base object. Programmers may make use of this registry using the following reflective methods defined in org.objectteams.Team:

boolean hasRole (Object aBase) ; This method checks whether a role for the passed base object already exists in the target team.

boolean hasRole (Object aBase, Class expectedRole) ; This method checks whether a instance of type expectedRole as a role for the passed base object aBase already exists in the target team. The role may also be of any subtype of the specified role type.

Object getRole (Object aBase) ; If the passed base object aBase already has a role in the target team, this role is returned. Otherwise null is returned.

<T> T getRole (Object aBase, Class<T> expectedRole) ; If the passed base object aBase already has a role in the target team that is assignable to the type represented by expectedRole, this role is returned. Otherwise null is returned.

Object[] getAllRoles () ; Retrieves all existing (registered) bound roles (§ 2.1.(a)) in the target team.
This method uses internal structures of weak references. For that reason it may return role instances which were about to be reclaimed by the garbage collector. If performance permits, it is thus advisable to always call System.gc() prior to calling getAllRoles() in order to achieve deterministic results (see also § 2.1.(f)).

<T> T[] getAllRoles (Class<T> expectedRole) ; Retrieves all existing (registered) bound roles (§ 2.1.(a)) in the target team that are assignable to the type represented by expectedRole. Class expectedRole must be a bound role otherwise an IllegalArgumentException is thrown.
See the note about garbage collection above.

void unregisterRole (Object aRole) ; This method unregisters the passed role object from the target team. Thus the corresponding base looses this role. After calling this method the role should no longer be used.

void unregisterRole (Object aRole, Class roleClass) ; This method unregisters the passed role object from the target team. Thus the corresponding base looses this role. After calling this method the role should no longer be used. The only difference to the previous method is improved speed because no search for the corresponding registry has to be performed.

It is desirable and possible to use these methods within guards (see § 5.4.). These methods allow to write the specification of guards in a more concise and more expressive way. Determined by the signature, the first four methods can only be used in a base-level guard (§ 5.4.2.) because they require a reference to a base object.

Listing 24: Example code (Guards and Reflection):

```
1  public team class SpecialConditions {
2    public void participate(Account as BonusAccount ba) {}
3    public class BonusAccount playedBy Account
4      base when(SpecialConditions.this.hasRole(base, BonusAccount.class))
5    {
6      callin void creditBonus(int amount) {
7        base.creditBonus(amount + bonus);
8      }
9      void creditBonus(int amount) <- replace void credit(int i)
10       base when (i > 1000);
11   }
12 }
```

Effects:

This teams provides a bonus system for registered `Accounts`. Every time an amount of more than 1000 is deposited to a registered account, additional 1% of the amount is credited.

- The team level method `participate` in line 2 uses declared lifting (see § 2.3.2.) to allow the passed `Account` object to participate the bonus system provided by the `SpecialConditions` team.
- The base guard in line 4 uses the reflective method `hasRole` to check whether the base object already has a role of type `BonusAccount` in the surrounding team. The expression `BonusAccount.class` returns the `java.lang.Class` object representing the role `BonusAccount` (see JLS §15.8.2[22]). This guard ensures, that only accounts explicitly registered via `participate` are ever decorated with a role of type `BonusAccount`.
- The method binding guard in line 10 restricts the callin to `creditBonus` to calls where the base method argument `amount` is greater than 1000.

(b) Behavioral reflection

The following reflective methods defined in org.objectteams.Team can be used to inspect the dynamic behavior of a team:

`boolean isExecutingCallin () ;` This method is used to inspect whether a control flow has already been intercepted by at least one callin binding of the current team. It can be used to avoid undesirable re-entrance to a team.

`boolean isActive () ;` This method checks whether the team instance is active for the current thread.

`boolean isActive ( Thread aThread ) ;` This method checks whether the team instance is active for the thread `aThread`.

(c) Class literals for roles

The Java syntax for so-called class literals, `MyClass.class` (see JLS §15.8.2[23]) can be used for role types with slightly changed semantics: Role types are virtual types

[23] http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#251530

(§ 1.3.1.) that are bound dynamically (§ 1.3.1.(e)). This applies to role class literals, too. From this follows the constraint that a role class literal can only be used within the non-static context of a team, ie., for evaluating a role class literal an enclosing team instance must be in scope.
Unlike regular type checking for role types, the class literal itself does not have a dependent type. Thus type checking of calls to methods like `hasRole(Object, Class)` cannot detect, whether the `Class` instance has actually been obtained from the correct team instance. Any attempt to pass a class that is not known as a bound role within the given team results in an `IllegalArgumentException` at run-time.

▷ Other API Elements § 6.2.

(a) Interfaces for role encapsulation

A set of pre-defined types exist that do not extend `java.lang.Object` and have no features except the operators `==` and `!=`.

Note:

The JLS defines that each interface declares all methods defined in `java.lang.Object` (JLS §9.2[24]) and also each object referenced by an interface type can be widened to `java.lang.Object`. Compilers commonly implement this by declaring `java.lang.Object` the super-type of all interfaces. Such implementation has no visible difference with respect to the more complex definition in the JLS.

These predefined types are

`org.objectteams.IConfined` regular interface

`org.objectteams.Team.IConfined` role interface

`org.objectteams.Team.Confined` role class

These types provide no new functionality but inheriting from these types influences the semantics with respect to encapsulation. The purpose and usage of these types is described in § 7..

(b) Interface for explicit lowering

The following role interface exists for the purpose of allowing explicit lowering:

`org.objectteams.Team.ILowerable` role interface

This interface was introduced in detail in § 2.2.(d).

(c) Team activation methods

Every team can be activated and deactivated by predefined methods of the class `org.objectteams.Team`.

`activate()` and `activate(Thread th)` Methods for activation of a team

`deactivate()` and `deactivate(Thread th)` Methods for deactivation of a team

The usage of these Methods is described in § 5.2.(b).

[24] `http://java.sun.com/docs/books/jls/second_edition/html/interfaces.doc.html#32392`

(d) Exceptions

The following `Exceptions` can be thrown during the execution of an ObjectTeam/-Java program:

`ResultNotProvidedException` Thrown if a replace callin without a base call does not provide the necessary (primitive type) base result (see § 4.3.(e)).

`LiftingFailedException` Thrown if an actual ambiguity occurs during lifting (see § 2.3.4.(c)).

`WrongRoleException` Thrown during lifting if the base object has, with respect to the same team instance, previously been lifted to a role type that is not conform to the currently requested type (see § 2.3.4.(d) and § 2.4.3.).

`DuplicateRoleException` Thrown during explicit role creation, if a new role is created for a base object, which already has a role of the required type in the given team (see § 2.4.1.(c)).

`RoleCastException` Thrown during cast of an externalized role, if the casted expression is anchored to a different team instance than the cast type (see § 1.2.4.(b)).

`LiftingVetoException` This exception is used internally to abort the process of lifting when a relevant guard predicate (§ 5.4.) evaluated to false. Such exceptions thrown from generated code will never appear in client code, so there is usually no need to catch a `LiftingVetoException`. However, in some situations it is useful to explicitly *throw* a `LiftingVetoException` from a lifting constructor (§ 2.3.1.(b)) of a role. This style allows to abort lifting even after the lifting constructor has started to work and also for method parameters requiring lifting. If lifting was triggered due to a callin method binding, this binding will simply not trigger if a `LiftingVetoException` is thrown while preparing the call to the role method.

(e) Role migration

The following interfaces can be used to enable role migration:

`IBaseMigratable` This interface declares a method

```
<B> void migrateToBase(B otherBase)
```

and instructs the compiler to generate an implementation of this method for any bound role declaring `IBaseMigratable` as its super-interface.
The effect of calling `migrateToBase` on a role instance is to re-bind this role to a new base instance. The base instance must be compatible to the role's base class (in order to avoid problems during lifting the compiler may require the base to be of the exact type of the role's base class). Passing `null` to this method causes an `NullPointerException` to be thrown.

`ITeamMigratable` This interface declares a method

```
<R> R<@otherTeam> migrateToTeam(final Team otherTeam)
```

and instructs the compiler to generate an implementation of this method for any role declaring `ITeamMigratable` as its super-interface.

The effect of calling `migrateToTeam` on a role instance is to re-bind this role to become a contained part of a new team instance. The team instance must be of the exact type of the role's enclosing team. Passing `null` to this method causes a `NullPointerException` to be thrown.

Caveat:
This method intentionally breaks the rules of family polymorphism: any reference `R<@previousTeam> r` which was established before migration will incorrectly imply that the role's enclosing team still is `previousTeam`, which is no longer true after migration. While this does not effect any method lookup (which is still safe), further assumptions based on a role's dependent type are invalidated by team migration. The same holds for references from the migrating role to any sibling role instances.
If the rules of family polymorphism should be maintained one should just refrain from declaring `ITeamMigratable` as a role's super-interface.

For both methods the signature declared in the interface is over-generalized, yet the compiler performs the necessary checks to ensure that role, base and team instances are indeed compatible and additionally the return type of `migrateToTeam` is checked as a self-type, i.e., it reflects the exact type of the call target.

▷ Role Encapsulation § 7.

Concepts of encapsulation

Protected roles A role with visibility `protected` cannot be externalized, which means its type cannot be used outside the declaring team (§ 1.2.3.).

Confined roles Confined roles are encapsulated even stricter than protected roles: the compiler will ensure that by no means any object outside the enclosing team will ever have a reference to a confined role.

Opaque roles Opaque roles build on the guarantees of confined roles but allow to be shared in a limited way such that no information is exposed.

▷ Opaque roles § 7.1.

The purpose of the two `IConfined` interfaces (see § 6.2.(a)) is to define **opaque roles**: Any role implementing `IConfined` can be externalized using this type, such that external clients cannot access any features of the role. The type `IConfined` exposes no features and references of this type cannot be widened to any type not even to `java.lang.Object`.
If the actual role type is furthermore invisible outside the team (by not declaring it `public`), it is perfectly safe to externalize such roles using type `IConfined` (which is a public interface) and pass them back to the owning team. The encapsulation of the team is in no way breached by externalizing opaque roles, which can only be used as a handle into internal state of the team.

The difference between the two mentioned interfaces is that `Team.IConfined` requires to use this type or any subtype as externalized role. Such a reference contains the information of the enclosing team. Even stricter control can be imposed using the regular interface `IConfined`. Here not even team membership is visible to clients using a reference of this type.

▷ Confined roles § 7.2.

Subclassing `Team.Confined` with a protected class yields a role class to which no object outside the team will ever have a reference. The point here is that instances of a role class with a regular super class can be widened to this super class. Widening can occur either in an assignment or when invoking a method which the role inherits from the regular super class, where the `this` reference is widened. In both cases the widened reference is no longer protected by the team and can leak out. This would break encapsulation of a role object that should only be accessible within the enclosing team.
Subclasses of `Team.Confined` are not compatible to any class outside their enclosing team (including `java.lang.Object`) and do not inherit any methods that have the danger of leaking `this`.

(a) Inhibition of overriding

The types `Team.IConfined` and `Team.Confined` cannot be overridden (cf. § 1.3.1.(c)).

(b) Arrays of Confined

For any confined type `C`, i.e., a type which is not compatible to `Object`, an array of `C` is not compatible to an array of `Object` nor to `Object` itself. This rule ensures that confinement cannot be bypassed by a sequence of compatible assignments and casts.

Upcoming:
Only by widening to a non-role super-type, a role instance can be accessed from outside the team. In the future this can be inhibited by restricted inheritance.

Listing 25: Example code (Role Encapsulation):

```
1  public team class Company {
2    private HashMap<String,Employee> employees;
3    ...
4    protected class Employee implements IConfined {
5      void pay(int amount) { ... }
6      ...
7    }
8    public IConfined getEmployee(String ID) {
9      return employees.get(ID); // implicit widening to IConfined
10   }
11   public void payBonus(IConfined emp, int amount) {
12     ((Employee)emp).pay(amount); // explicit narrowing
13   }
14 }
```

```
15 public class Main {
16   public static void main(String[] args) {
17     final Company comp = new Company();
18     IConfined<@comp> emp = comp.getEmployee("emp1");
19     // System.out.println(emp); <- forbidden!
20     comp.payBonus(emp, 100);
21   }
22 }
```

Effects:

- The `protected` role `Employee` implements the above described interface `IConfined` and therefore becomes **opaque** (line 4).
- Methods sharing such an opaque role with the outside of the enclosing team have to use the type `IConfined` (line 8, line 11).
- It is possible to obtain an instance of such a role by using the type `IConfined` (line 18).
- Trying to access any feature of this instance, for example `toString()`, will cause a compilation error (line 19).
- Passing the opaque role reference back into the team works well (line 20).
- Inside the team some conversions between the types `IConfined` and the intrinsic role type `Employee` may be necessary (line 9 and 12).

▷ Join Point Queries § 8.

Defining sets of join points for interception

join point In ObjectTeams/Java a join point is considered to be an element of the program. A meta model exists which defines the **kinds** of join points that can be identified.

join point interception The purpose of identifying join points is to intercept program execution at these points by means of *callin* bindings.

join point query Sets of join points are defined using functional queries of the program's reflective representation.

pointcuts Dynamic "pointcuts" comparable to AspectJ's `cflow` or even the Trace-Matches approach are not subject to the join point language of OT/J. These features will be added at a different level of abstraction. Note, that guard predicates (§ 5.4.) subsume the dynamic capabilities of the pointcuts `if, target, this, args`.

▷ Join point queries § 8.1.

This section will describe the query language used to define sets of join points. As of version 1.0.0 of the OTDT this query language is not yet supported.

▷ Query expressions § 8.2.

▷ OT/J meta model § 8.3.

▷ Value Dependent Classes § 9.

Generalizing externalized roles

Type Value Parameter In addition to regular generics, a class may declare parameters that represent an object value. Such a value parameter is called the **type anchor** for this class, the class's type is said to be **anchored** to this parameter.

Value Dependent Classes A class that declares one or more value parameters depends on the runtime instance(s) denoted by its anchor(s).

Externalized Roles The concept of externalized roles (§ 1.2.2.) is a special case of the concepts presented here.

▷ Defining classes with value parameters § 9.1.

→ Syntax § A.9.1

(a) Value parameter declaration

Within the angle brackets that mark the parameters of a generic class also value parameters can be declared. In contrast to a type parameter, a value parameter is denoted as a pair of two identifiers: a type and a free name, e.g.,

```
class MyClass<YourType aName> { ...
```

Note that value parameters are valid for classes only, not for interfaces.

(b) Value parameter application

Within the given class (`MyClass`) the parameter name (`aName`) can be used like a final field of the given type (`YourType`). In contrast to regular final fields the assignment to this name occurs even before the constructor is executed.

(c) Role types as dependent types

Any role type can be interpreted as a value dependent type, however, in the declaration of a role type the value parameter remains implicit: it is identical to the enclosing team instance.

▷ Using classes with value parameters § 9.2.

→ Syntax § A.9.2

When using a class which declares one or more value parameters (type anchors) a corresponding **anchor value** has to be provided.

▷ Parameter substitution § 9.2.1.

Substitution of a type anchor of a class `MyClass<YourType p>` is denoted as `MyClass<@v>`. In this term `v` must be a value which is conform to the declaration of the value parameter "`YourType p`", ie., `v` must have the static type `YourType`.

The value passed for substituting a type anchor must be a path of variables declared as final. Obviously, only the first element in such a path can be a local variable or a method argument, all other elements have to be fields. The reason for requiring final variables is in type checking as discussed next.

Note:
Externalized roles as defined in § 1.2.2.(b) are a special case of types with a value parameter, where the value is an instance of the enclosing team.

§ 9.2.2. Type conformance

Two value dependent types (anchored types) are considered conform only if the anchors of both types refer to *the same object(s)*. The compiler must be able to statically analyze this anchor identity.

(a) Substitutions for type anchors

Only two substitutions are considered for determining anchor identity:

1. If a method signature uses this as the anchor of any of its types, type checking an application of this method performs the following substitutions:
   A simple this expression is substituted by the actual call target of the method application.
   A qualified Outer.this expression is substituted by the corresponding enclosing instance of the call target.
2. Assignments from a final identifier to another final identifier are transitively followed, i.e., if t1, t2 are final, after an assignment t1=t2 the types C<@t1> and C<@t2> are considered identical. Otherwise C<@t1> and C<@t2> are incommensurable.
   Attaching an actual parameter to a formal parameter in a method call is also considered as an assignment with respect to this rule.

(b) Conformance of raw types

After anchors have been proven identical, the raw types are checked for compatibility using the standard Java rules.

§ 9.3. Restrictions and limitations

(a) No overriding

Types with value parameters that are declared outside a team cannot be overridden, as roles can be. Therefor, implicit inheritance does not apply for these types.

(b) Only first parameter

Currently only the first parameter of a class may be a value parameter. This restriction may be removed in the future.

▷ ObjectTeams/Java Syntax § A.

Notation

The following grammar rules extend the Java grammar given in the Java Language Specification[25]. We adopt the conventions of printing non-terminal symbols in italic font (e.g., *ClassDeclaration*), and terminal symbols in roman font (e.g., `class`). Names printed in black refer to definitions from the original Java grammar. Object Teams additions are printed in **blue boldface**. For those rules that simply add a new option to an existing rule, the original options are indicated by an ellipse (. . .).

▷ Keywords § A.0.

The keywords introduced by OT/J have different scopes, which means outside their given scope these keywords can be used for regular identifiers. Only these names are keywords unconditionally:

readonly , **team** , **within**

▷ Scoped keywords § A.0.1.

The following names are keywords in OT/J only if they appear within a team or role class, ie., after the keyword **team** has been recognized:

as , **base** , **callin** , **playedBy** , **precedence** , **tsuper** , **with** , **when**

These names are keywords only in the context of a callin or callout binding respectively (§ A.3.):

after , **before** , **replace** , **get** , **set**

▷ Inheriting scoped keywords § A.0.2.

While regular Java classes may use the scoped keywords (§ A.0.1.) of OT/J freely, it is an error if a role class inherits a feature whose name is a scoped keyword.

▷ Internal names § A.0.3.

Compiler and runtime environment generate internal methods and fields which start with the prefix `_OT$`. It is illegal to use any of these methods and fields within client code.

▷ Class definitions § A.1.

Class definitions add two new keywords `team` and `playedBy`. Classes which use these keywords are called **teams** and **bound roles**, respectively. Any class that inherits from a bound role class (either by an `extends` clause or by implicit inheritance, cf. § 1.3.1.(c)) is again a bound role class.

[25] `http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html`

§ A.1.1	***ClassDeclaration:*** *[Modifiers]* ***[team]*** class *Identifier [*extends *Type] [*implements *TypeList]* ***[playedBy Type] [Guard]*** *ClassBody*

Contextual constraints:

(a) A class which has a `playedBy` clause (a **bound role** class) may not be declared static and must be directly contained in a class that has the `team` modifier (a **team** class).

(b) A class which inherits from a team class must have the `team` modifier, too.

(c) A class which has a guard (see § 5.4.) must be a team or a role.

§ A.2. Modifiers

The rule for method modifiers adds one keyword: `callin`:

§ A.2.1	***Modifier:*** ... **callin**

Contextual constraints:

(a) The class of a method which has the `callin` modifier may not be declared static and must be directly contained in a team class.

(b) A method that has the `callin` modifier may not appear in an explicit method call (rule Apply in JLS).

§ A.3. Method bindings

The rule of items declarable in a class body is augmented by method bindings:

§ A.3.1	***ClassBodyDeclaration:*** ... ***CalloutBinding*** ***CallinBinding***
§ A.3.2	***CalloutBinding:*** *[Modifier]* ***MethodSpec CalloutKind MethodSpec CalloutParameterMappings*** *[Modifier]* ***MethodSpec CalloutKind CalloutModifier FieldSpec***
§ A.3.3	***Callin binding:*** *[Identifier :]* ***MethodSpec <- CallinModifier MethodSpecs*** ***[Guard] CallinParameterMappings***
§ A.3.4	***MethodSpec:*** *Identifier* *ResultType MethodDeclarator*

Note, that ResultType *and* MethodDeclarator *are not explicit in the overall syntax of the Java language specification. For convenience we refer to the definition in section 8.4. Method Declarations*[26] *of the Java language specification.*

[26] `http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#40420`

§ A.3.5	*MethodSpecs:* *MethodSpec [, MethodSpecs]*
§ A.3.6	*CalloutKind:* -> =>
§ A.3.7	*CallinModifier:* **before** **after** **replace**
§ A.3.8	*CalloutModifier:* **get** **set**
§ A.3.9	*FieldSpec:* *[Type] Identifier*

Contextual constraints:

(a) `CalloutBindings` and `CallinBindings` may occur only in bound role classes.

(b) A `CalloutBinding` or `CallinBinding` may not mix identifiers and full signatures (`MethodDeclarationHead`) for its method specifiers (`MethodSpec`).
Binding a full method signature to a field requires the `FieldSpec` to include the `Type`.

(c) The method specifier at the left hand side of a `CallinBinding` which has the `replace` modifier must refer to a method that has the `callin` modifier.

(d) The `Modifier` of a callout binding can only be one of the visility modifiers `public`, `protected` or `private`. A short callout binding (i.e., without signatures) must not specify a visibility modifier.

▷ Parameter mappings § A.4.

§ A.4.1	*CalloutParameterMappings:* **with** { *CalloutParameterMappingList [,]* } ;
§ A.4.2	*CallinParameterMappings:* **with** { *CallinParameterMappingList [,]* } ;
§ A.4.3	*CalloutParameterMappingList:* *CalloutParameterMapping [, CalloutParameterMappingList]*
§ A.4.4	*CallinParameterMappingList:* *CallinParameterMapping [, CallinParameterMappingList]*
§ A.4.5	*CalloutParameterMapping:* *Expression* -> *Identifier* **result** <- *Expression*
§ A.4.6	*CallinParameterMapping:* *Identifier* <- *Expression* *Expression* -> **result**

Note:
By defining ";" as an option for parameter mappings, the grammar enforces that method

bindings without a parameter mapping are terminated by a ";". Also method bindings with parameter mappings may optionally be terminated by a ";", which in that case is interpreted as an empty member declaration, following the same pattern how non-abstract methods in Java may optionally have a trailing ";".

§ A.5. Statements

§ A.5.1	***Statement:*** ... ***Within*** ***BaseCall*** ***TSuperCall***
§ A.5.2	***Within:*** **within** (*Expression*) *Statement*
§ A.5.3	***BaseCall:*** **base** . *Identifier* ($Arguments_{opt}$) **base** ($Arguments_{opt}$)
§ A.5.4	***TSuperCall:*** **tsuper** . *Identifier* ($Arguments_{opt}$) **tsuper** ($Arguments_{opt}$)

Contextual constraints:

(a) The expression of a `Within` must evaluate to an instance of a team class.

(b) The first form of a `BaseCall` may occur only in the body of a method that has the `callin` modifier. The identifier must be the name of the enclosing method.

(c) The second form of a `BaseCall` may occur only in a constructor of a bound role class.

(d) The first form of a `TSuperCall` may occur only in a method of a role class.

(e) The second form of a `TSuperCall` may occur only in a constructor of a role class.

§ A.6. Types

§ A.6.1	***Type:*** ... ***LiftingType*** ***AnchoredType***
§ A.6.2	***LiftingType:*** *Type* **as** *Type*
§ A.6.3	***AnchoredType:*** ***Path***. *Type*
§ A.6.4	***Path:*** *Identifier* ***Path***.*Identifier*

Contextual constraints:

(a) Location
A `LiftingType` may only occur in the parameter list of a method of a team class.

(b) Role in scope
The right hand side type in a `LiftingType` must be a class directly contained in the enclosing team class (the class may be acquired by implicit inheritance (§ 1.3.1.(c))).

(c) Team path
Note, that the syntax of §A.6.3/4 is deprecated in favor of § A.9..
The path in an `AnchoredType` must refer to an instance of a team class. Each identifier in the path must be declared with the `final` modifier.

▷ Guard predicates § A.7.

§ A.7.1	***Guard:*** ***[base]* when** (*Expression*)
§ A.7.2	***MethodDeclaration:*** ... *MethodHeader* ***[Guard]*** *MethodBody*

Other rules referring to *Guard*: ClassDeclaration (§ A.1.), CallinBinding (§ A.3.)

Contextual constraints:

(a) The `Expression` in a guard must have type `boolean`.

▷ Precedence declaration § A.8.

§ A.8.1	***PrecedenceDeclaration:*** **precedence** *CallinNameList* ;
§ A.8.2	***CallinNameList:*** *Name [, CallinNameList]*

▷ Value dependent types § A.9.

§ A.9.1	***TypeParameter:*** *TypeVariable [TypeBound]* ***ReferenceType Name***

See JLS 3 §4.4[27]

§ A.9.2	***ActualTypeArgument:*** *ReferenceType* *Wildcard* ***@Name***

See JLS 3 §4.5.1[28]

Contextual constraints:

(a) ActualTypeParameter
An `ActualTypeArgument` of the form `@Name` may only occur as a parameter of a simple name type reference.

[27] `http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#108850`
[28] `http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#107353`

§ A.10. Packages and imports

§ A.10.1	***PackageDeclaration:*** ... **team** *QualifiedName* ;
§ A.10.2	***Import:*** ... **import base** *QualifiedName* ;

▷ Changes between versions § B.

▷ Paragraphs changed between versions § B.1.

(a) Between OTJLD 1.0 and OTJLD 1.1

- § 3.2.(a) : **Parameter mappings**
  Disallow parameter mappings in a role interface.
- § 4.5.(d) : **Replace bindings**
  Disallow unsafe use of polymorphism and primitive type conversions.
- § 6.1.(a) : **Signatures of reflective methods**
  Made two methods generic so that return values can be used without the need of casting.
- § 7.2. : **Confined roles**
  Improved explanation.

(b) Between OTJLD 1.1 and OTJLD 1.2

- § 1.2.1.(e): **Visibility of role features**
  Clarification has been added that a role can always access all the features that its enclosing team has access to.
- § 2.1.2.(e): **Generic roles/bases**
  Relaxed the rules about generic bound roles. This change also subsumes what previously was a specific restriction in § 4.1.(b).
- § 3.1.(i) and § 3.5.(f): **Visibility of shorthand callout**
  A role method defined by a shorthand callout binding can now specify a visibility modifier (see also § A.3.), otherwise it inherits the visibility modifier of it's bound base method/field.
- § 3.1.(j) and § 3.5.(h): **Visibility of inferred callout**
  Role methods inferred as a callout binding are either `public` (inferred via interface) or `private` inferred from self call / field access.
- § 3.5.(h): **No explicit use of inferred callout to field**
  Clarification has been added that an accessor method generated for an inferred callout to field can not be explicitly invoked.
- § 4.1.(b): **No callin in generic role**
  A restriction has been made explicit that a generic role cannot define callin bindings.
- § 4.2.(d) : **Callin methods**
  Slightly rephrased and extended the rule to make explicit that a callin method can indeed be intercepted using a second level callin binding.
- § 6.1.(a) : **Reflective methods `getAllRoles`**
  More precision: answer only *bound* roles.

§ B.2. Additions between versions

(a) Between OTJLD 1.0 and OTJLD 1.1

- § 1.2.4.(c): **Role class literal**
  Made existing feature explicit and introduce new qualified class literal for externalized roles.
- § 3.1.(j) and § 3.5.(h) : **Inferred callout**
  New feature.
- § 4.6.(a) : **Callin-binding private methods from super classes**
  Added a necessary restriction.
- § 4.9. : **Callin inheritance**
  Clarified issues that where under-specified or insufficiently explained, specifically:
  - Effect of callin bindings on inherited or overridden base methods (§ 4.9.1.).
  - Interplay of callin bindings and base methods with covariant return types (§ 4.9.3.)
- § 4.10.: **Generic replace bindings**
  Reconcile type safety of replace bindings as introduced in § 4.5.(d) with desirable flexibility by using type parameters.
- § 7.2.(b) : **Arrays of Confined**
  Added a necessary restriction.

(b) Between OTJLD 1.1 and OTJLD 1.2

- § 1.2.2.(h) : **Externalized creation**
  Added alternative syntax using value parameter and changed title.
- § 1.2.5.(f) : **Imports in role files**
  Added a missing rule defining the effect of imports in role files.
- § 1.3.1.(c) : **@Override annotation for roles**
  The regular `@Override` annotation (Java âĽĕ5) has been extended to apply to role classes, too.
- § 1.3.1.(k) : **Covariant return types**
  Necessary constraint for covariant return types in the presence of both implicit and explicit inheritance.
- § 2.1.2.(c) : **Binding to final base class**
  It has been added that binding to a final base class is now considered as decapsulation, too.
- § 2.2.(f) : **Ambiguous lowering**
  A diagnostic has been added to detect situations where lowering might be intended but fails because the declared type is `java.lang.Object`, which makes a potential lowering translation unnecessary and thus ambiguous.
- § 2.3.2.(e) : **Generic declared lifting**
  Support passing unrelated base types into the same method with declared lifting.

- § 2.6.(g) : **Decapsulation via base reference**
  Extended applicability of decapsulation to two more positions.
- § 4.3.(f) : **Base super call**
  Support base calls directly to the super version of the bound base method, thus bypassing both the exact bound base method and also any further callins relating to this base method or its super version.
- § 5.4.(b) : **Side-effects in guard predicates**
  Migrate previous note about a future feature to a regular paragraph.
- § 5.4.(c) : **Exceptions in guard predicates**
  Clarify the effect of exceptions thrown from a guard predicate.
- § 6.2.(d) : **LiftingVetoException**
  Added documentation for the mostly internal `LiftingVetoException` and how it could actually be used in client code.
- § 6.2.(e) : **Role migration**
  Added two interfaces to add migration capabilities to a role class.