Classes & Objects

Classes and Objects

```
TmplDef          ::= [‘case’] ‘class’ ClassDef
                   |  [‘case’] ‘object’ ObjectDef
                   |  ‘trait’ TraitDef
```

Classes and objects are both defined in terms of *templates*.

Templates

```
ClassTemplate   ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate   ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents    ::=  Constr {‘with’ AnnotType}
TraitParents    ::=  AnnotType {‘with’ AnnotType}
TemplateBody   ::= [nl] ‘{’ [SelfType] TemplateStat {semi TemplateStat} ‘}’
SelfType        ::=  id [‘:’ Type] ‘=>’
                  |   this ‘:’ Type ‘=>’
```

A *template* defines the type signature, behavior and initial state of a trait or class of objects or of a single object. Templates form part of instance creation expressions, class definitions, and object definitions. A template `$sc$ with $mt_1$ with $\ldots$ with $mt_n$ { $\mathit{stats}$ }` consists of a constructor invocation *sc* which defines the template's *superclass*, trait references `$mt_1 , \ldots , mt_n$` ($n \geq 0$), which define the template's *traits*, and a statement sequence *stats* which contains initialization code and additional member definitions for the template.

Each trait reference mt_i must denote a trait. By contrast, the superclass constructor *sc* normally refers to a class which is not a trait. It is possible to write a list of parents that starts with a trait reference, e.g. `$mt_1$ with $\ldots$ with $mt_n$`. In that case the list of parents is implicitly extended to include the supertype of mt_1 as first parent type. The new supertype must have at least one constructor that does not take parameters. In the following, we will always assume that this implicit extension has been performed, so that the first parent class of a template is a regular superclass constructor, not a trait reference.

The list of parents of a template must be well-formed. This means that the class denoted by the superclass constructor *sc* must be a subclass of the superclasses of all the traits $mt_1, \ldots, mt_n$. In other words, the non-trait classes inherited by a template form a chain in the inheritance hierarchy which starts with the template's superclass.

The *least proper supertype* of a template is the class type or compound type consisting of all its parent class types.

The statement sequence *stats* contains member definitions that define new members or override members in the parent classes. If the template forms part of an abstract class or trait definition, then *stats* may also contain declarations of abstract members. If the template forms part of a concrete class definition, *stats* may still contain declarations of abstract type members, but not of abstract term members. Furthermore, *stats* may in any case also contain strictly evaluated expressions: these are executed in the order they are given as part of the initialization of a template, even if they appear in the definition of overridden members.

The sequence of template statements may be prefixed with a formal parameter definition and an arrow, e.g. `$x$ =>`, or `$x$:$T$ =>`. If a formal parameter is given, it can be used as an alias for the reference `this` throughout the body of the template. If the formal parameter comes with a type T, this definition affects the *self type* S of the underlying class or object as follows: Let C be the type of the class or trait or object defining the template. If a type T is given for the formal self parameter, S is the greatest lower bound of T and C. If no type T is given, S is just C. Inside the template, the type of `this` is assumed to be S.

The self type of a class or object must conform to the self types of all classes which are inherited by the template t.

A second form of self type annotation reads just `this: $S$ =>`. It prescribes the type S for `this` without introducing an alias name for it.

Example

Consider the following class definitions:

```
class Base extends Object {}
trait Mixin extends Base {}
object O extends Mixin {}
```

In this case, the definition of `O` is expanded to:

```
object O extends Base with Mixin {}
```

Inheriting from Java Types A template may have a Java class as its superclass and Java interfaces as its mixins.

Template Evaluation Consider a template `$sc$ with $mt_1$ with $mt_n$ { $\mathit{stats}$ }`.

If this is the template of a trait then its *mixin-evaluation* consists of an evaluation of the statement sequence *stats*.

If this is not a template of a trait, then its *evaluation* consists of the following steps.

- First, the superclass constructor *sc* is evaluated.
- Then, all base classes in the template's linearization up to the template's superclass denoted by *sc* are mixin-evaluated. Mixin-evaluation happens in reverse order of occurrence in the linearization.
- Finally the statement sequence *stats* is evaluated.

Delayed Initialization

The initialization code of an object or class (but not a trait) that follows the superclass constructor invocation and the mixin-evaluation of the template's base classes is passed to a special hook, which is inaccessible from user code. Normally, that hook simply executes the code that is passed to it. But templates inheriting the `scala.DelayedInit` trait can override the hook by re-implementing the `delayedInit` method, which is defined as follows:

```
def delayedInit(body: => Unit)
```

Constructor Invocations

```
Constr  ::=  AnnotType {‘(’ [Exprs] ‘)’}
```

Constructor invocations define the type, members, and initial state of objects created by an instance creation expression, or of parts of an object's definition which are inherited by a class or object definition. A constructor invocation is a function application `$x$.$c$[$\mathit{targs}$]($\mathit{args}_1$)$\ldots$($\mathit{args}_n$)`, where x is a stable identifier, c is a type name which either designates a class or defines an alias type for one, *targs* is a type argument list, $args_1, \ldots, args_n$ are argument lists, and there is a constructor of that class which is applicable to the given arguments. If the constructor invocation uses named or default arguments, it is transformed into a block expression using the same transformation as described here.

The prefix `$x$.` can be omitted. A type argument list can be given only if the class c takes type parameters. Even then it can be omitted, in which case a type argument list is synthesized using local type inference. If no explicit arguments are given, an empty list `()` is implicitly supplied.

An evaluation of a constructor invocation `$x$.$c$[$\mathit{targs}$]($\mathit{args}_1$)$\ldots$($\mat` consists of the following steps:

- First, the prefix x is evaluated.
- Then, the arguments $args_1, \ldots, args_n$ are evaluated from left to right.
- Finally, the class being constructed is initialized by evaluating the template of the class referred to by c.

Class Linearization

The classes reachable through transitive closure of the direct inheritance relation from a class C are called the *base classes* of C. Because of mixins, the inheritance relationship on base classes forms in general a directed acyclic graph. A linearization of this graph is defined as follows.

Definition: linearization

Let C be a class with template `$C_1$ with ... with $C_n$ { $\mathit{stats}$ }`. The *linearization* of C, $\mathcal{L}(C)$ is defined as follows:

$$\mathcal{L}(C) = C, \mathcal{L}(C_n) \vec{+} \ ... \ \vec{+} \ \mathcal{L}(C_1)$$

Here $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:

$$\begin{aligned} \{a, A\} \vec{+} B &= a, (A \vec{+} B) \quad if\ a \notin B \\ &= A \vec{+} B \qquad if\ a \in B \end{aligned}$$

Example

Consider the following class definitions.

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

Then the linearization of class `Iter` is

```
{ Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any }
```

Note that the linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. Linearization also satisfies the property that a linearization of a class always contains the linearization of its direct superclass as a suffix.

For instance, the linearization of `StringIterator` is

```
{ StringIterator, AbsIterator, AnyRef, Any }
```

which is a suffix of the linearization of its subclass `Iter`. The same is not true for the linearization of mixins. For instance, the linearization of `RichIterator` is

```
{ RichIterator, AbsIterator, AnyRef, Any }
```

which is not a suffix of the linearization of `Iter`.

Class Members

A class C defined by a template `$C_1$ with $\ldots$ with $C_n$ { $\mathit{stats}$ }` can define members in its statement sequence *stats* and can inherit members from all parent classes. Scala adopts Java and C#'s conventions for static overloading of methods. It is thus possible that a class defines and/or inherits several methods with the same name. To decide whether a defined member of a class C overrides a member of a parent class, or whether the two co-exist as overloaded variants in C, Scala uses the following definition of *matching* on members:

Definition: matching

A member definition M *matches* a member definition M', if M and M' bind the same name, and one of following holds.

1. Neither M nor M' is a method definition.
2. M and M' define both monomorphic methods with equivalent argument types.
3. M defines a parameterless method and M' defines a method with an empty parameter list `()` or *vice versa*.
4. M and M' define both polymorphic methods with equal number of argument types $\overline{T}$, $\overline{T}'$ and equal numbers of type parameters $\overline{t}$, $\overline{t}'$, say, and $\overline{T}' = [\overline{t}'/\overline{t}]\overline{T}$.

Member definitions fall into two categories: concrete and abstract. Members of class C are either *directly defined* (i.e. they appear in C's statement sequence *stats*) or they are *inherited*. There are two rules that determine the set of members of a class, one for each category:

A *concrete member* of a class C is any concrete definition M in some class $C_i \in \mathcal{L}(C)$, except if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which directly defines a concrete member M' matching M.

An *abstract member* of a class C is any abstract definition M in some class $C_i \in \mathcal{L}(C)$, except if C contains already a concrete member M' matching M, or if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which directly defines an abstract member M' matching M.

This definition also determines the overriding relationships between matching members of a class C and its parents. First, a concrete definition always overrides an abstract definition. Second, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

It is an error if a template directly defines two matching members. It is also an error if a template contains two members (directly defined or inherited) with the same name and the same erased type. Finally, a template is not allowed to contain two methods (directly defined or inherited) with the same name which both define default arguments.

Example

Consider the trait definitions:

```
trait A { def f: Int }
trait B extends A { def f: Int = 1 ; def g: Int = 2 ; def h: Int = 3 }
trait C extends A { override def f: Int = 4 ; def g: Int }
trait D extends B with C { def h: Int }
```

Then trait `D` has a directly defined abstract member `h`. It inherits member `f` from trait `C` and member `g` from trait `B`.

Overriding

A member M of class C that matches a non-private member M' of a base class of C is said to *override* that member. In this case the binding of the overriding member M must subsume the binding of the overridden member M'. Furthermore, the following restrictions on modifiers apply to M and M':

- M' must not be labeled `final`.
- M must not be `private`.
- If M is labeled `private[$C$]` for some enclosing class or package C, then M' must be labeled `private[$C'$]` for some class or package C' where C' equals C or C' is contained in C.
- If M is labeled `protected`, then M' must also be labeled `protected`.
- If M' is not an abstract member, then M must be labeled `override`. Furthermore, one of two possibilities must hold:
  - either M is defined in a subclass of the class where is M' is defined,
  - or both M and M' override a third member M'' which is defined in a base class of both the classes containing M and M'
- If M' is incomplete in C then M must be labeled `abstract override`.
- If M and M' are both concrete value definitions, then either none of them is marked `lazy` or both must be marked `lazy`.
- A stable member can only be overridden by a stable member. For example, this is not allowed:

```
class X { val stable = 1}
class Y extends X { override var stable = 1 } // error
```

Another restriction applies to abstract type members: An abstract type member with a volatile type as its upper bound may not override an abstract type member which does not have a volatile upper bound.

A special rule concerns parameterless methods. If a parameterless method defined as `def $f$: $T$ = ...` or `def $f$ = ...` overrides a method of type $()T'$ which has an empty parameter list, then f is also assumed to have an empty parameter list.

An overriding method inherits all default arguments from the definition in the superclass. By specifying default arguments in the overriding method it is possible to add new defaults (if the corresponding parameter in the superclass does not have a default) or to override the defaults of the superclass (otherwise).

Example

Consider the definitions:

```
trait Root { type T <: Root }
trait A extends Root { type T <: A }
trait B extends Root { type T <: B }
trait C extends A with B
```

Then the class definition `C` is not well-formed because the binding of `T` in `C` is `type T <: B`, which fails to subsume the binding `type T <: A` of `T` in type `A`. The problem can be solved by adding an overriding definition of type `T` in class `C`:

```
class C extends A with B { type T <: C }
```

Inheritance Closure

Let C be a class type. The *inheritance closure* of C is the smallest set $\mathcal{S}$ of types such that

- C is in $\mathcal{S}$.
- If T is in $\mathcal{S}$, then every type T' which forms syntactically a part of T is also in $\mathcal{S}$.
- If T is a class type in $\mathcal{S}$, then all parents of T are also in $\mathcal{S}$.

It is a static error if the inheritance closure of a class type consists of an infinite number of types. (This restriction is necessary to make subtyping decidable[1]).

Early Definitions

```
EarlyDefs         ::= '{' [EarlyDef {semi EarlyDef}] '}' 'with'
EarlyDef          ::=  {Annotation} {Modifier} PatVarDef
```

A template may start with an *early field definition* clause, which serves to define certain field values before the supertype constructor is called. In a template

```
{ val $p_1$: $T_1$ = $e_1$
  ...
  val $p_n$: $T_n$ = $e_n$
} with $sc$ with $mt_1$ with $mt_n$ { $\mathit{stats}$ }
```

[1] Kennedy, Pierce. On Decidability of Nominal Subtyping with Variance. in FOOL 2007

The initial pattern definitions of $p_1, \ldots, p_n$ are called *early definitions*. They define fields which form part of the template. Every early definition must define at least one variable.

An early definition is type-checked and evaluated in the scope which is in effect just before the template being defined, augmented by any type parameters of the enclosing class and by any early definitions preceding the one being defined. In particular, any reference to `this` in the right-hand side of an early definition refers to the identity of `this` just outside the template. Consequently, it is impossible that an early definition refers to the object being constructed by the template, or refers to one of its fields and methods, except for any other preceding early definition in the same section. Furthermore, references to preceding early definitions always refer to the value that's defined there, and do not take into account overriding definitions. In other words, a block of early definitions is evaluated exactly as if it was a local bock containing a number of value definitions.

Early definitions are evaluated in the order they are being defined before the superclass constructor of the template is called.

Example

Early definitions are particularly useful for traits, which do not have normal constructor parameters. Example:

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting {
  println(msg)
}
```

In the code above, the field `name` is initialized before the constructor of `Greeting` is called. Therefore, field `msg` in class `Greeting` is properly initialized to `"How are you, Bob"`.

If `name` had been initialized instead in `C`'s normal class body, it would be initialized after the constructor of `Greeting`. In that case, `msg` would be initialized to `"How are you, <null>"`.

Modifiers

```
Modifier          ::=  LocalModifier
                     |  AccessModifier
                     |  'override'
LocalModifier     ::=  'abstract'
```

```
                    |  'final'
                    |  'sealed'
                    |  'implicit'
                    |  'lazy'
AccessModifier    ::=  ('private' | 'protected') [AccessQualifier]
AccessQualifier   ::=  '[' (id | 'this') ']'
```

Member definitions may be preceded by modifiers which affect the accessibility and usage of the identifiers bound by them. If several modifiers are given, their order does not matter, but the same modifier may not occur more than once. Modifiers preceding a repeated definition apply to all constituent definitions. The rules governing the validity and meaning of a modifier are as follows.

`private`

The `private` modifier can be used with any definition or declaration in a template. Such members can be accessed only from within the directly enclosing template and its companion module or companion class.

A `private` modifier can be *qualified* with an identifier C (e.g. `private[$C$]`) that must denote a class or package enclosing the definition. Members labeled with such a modifier are accessible respectively only from code inside the package C or only from code inside the class C and its companion module.

A different form of qualification is `private[this]`. A member M marked with this modifier is called *object-protected*; it can be accessed only from within the object in which it is defined. That is, a selection $p.M$ is only legal if the prefix is `this` or `$O$.this`, for some class O enclosing the reference. In addition, the restrictions for unqualified `private` apply.

Members marked private without a qualifier are called *class-private*, whereas members labeled with `private[this]` are called *object-private*. A member *is private* if it is either class-private or object-private, but not if it is marked `private[$C$]` where C is an identifier; in the latter case the member is called *qualified private*.

Class-private or object-private members may not be abstract, and may not have `protected` or `override` modifiers. They are not inherited by subclasses and they may not override definitions in parent classes.

`protected`

The `protected` modifier applies to class member definitions. Protected members of a class can be accessed from within - the template of the defining class, - all templates that have the defining class as a base class, - the companion module of any of those classes.

A `protected` modifier can be qualified with an identifier C (e.g. `protected[$C$]`) that must denote a class or package enclosing the definition. Members labeled with such a modifier are also accessible respectively from all code inside the package C or from all code inside the class C and its companion module.

A protected identifier x may be used as a member name in a selection `$r$.$x$` only if one of the following applies: - The access is within the template defining the member, or, if a qualification C is given, inside the package C, or the class C, or its companion module, or - r is one of the reserved words `this` and `super`, or - r's type conforms to a type-instance of the class which contains the access.

A different form of qualification is `protected[this]`. A member M marked with this modifier is called *object-protected*; it can be accessed only from within the object in which it is defined. That is, a selection $p.M$ is only legal if the prefix is `this` or `$O$.this`, for some class O enclosing the reference. In addition, the restrictions for unqualified `protected` apply.

`override`

The `override` modifier applies to class member definitions or declarations. It is mandatory for member definitions or declarations that override some other concrete member definition in a parent class. If an `override` modifier is given, there must be at least one overridden member definition or declaration (either concrete or abstract).

`abstract override`

The `override` modifier has an additional significance when combined with the `abstract` modifier. That modifier combination is only allowed for value members of traits.

We call a member M of a template *incomplete* if it is either abstract (i.e. defined by a declaration), or it is labeled `abstract` and `override` and every member overridden by M is again incomplete.

Note that the `abstract override` modifier combination does not influence the concept whether a member is concrete or abstract. A member is *abstract* if only a declaration is given for it; it is *concrete* if a full definition is given.

`abstract`

The `abstract` modifier is used in class definitions. It is redundant for traits, and mandatory for all other classes which have incomplete members. Abstract classes cannot be instantiated with a constructor invocation unless followed by mixins and/or a refinement which override all incomplete members of the class. Only abstract classes and traits can have abstract term members.

The `abstract` modifier can also be used in conjunction with `override` for class member definitions. In that case the previous discussion applies.

`final`

The `final` modifier applies to class member definitions and to class definitions. A `final` class member definition may not be overridden in subclasses. A `final` class may not be inherited by a template. `final` is redundant for object definitions. Members of final classes or objects are implicitly also final, so the `final` modifier is generally redundant for them, too. Note, however, that constant value definitions do require an explicit `final` modifier, even if they are defined in a final class or object. `final` is permitted for abstract classes but it may not be applied to traits or incomplete members, and it may not be combined in one modifier list with `sealed`.

`sealed`

The `sealed` modifier applies to class definitions. A `sealed` class may not be directly inherited, except if the inheriting template is defined in the same source file as the inherited class. However, subclasses of a sealed class can be inherited anywhere.

`lazy`

The `lazy` modifier applies to value definitions. A `lazy` value is initialized the first time it is accessed (which might never happen at all). Attempting to access a lazy value during its initialization might lead to looping behavior. If an exception is thrown during initialization, the value is considered uninitialized, and a later access will retry to evaluate its right hand side.

Example

The following code illustrates the use of qualified private:

```
package outerpkg.innerpkg
class Outer {
  class Inner {
    private[Outer] def f()
    private[innerpkg] def g()
    private[outerpkg] def h()
  }
}
```

Here, accesses to the method `f` can appear anywhere within `Outer`, but not outside it. Accesses to method `g` can appear anywhere within the package `outerpkg.innerpkg`, as would be the case for package-private methods in Java.

Finally, accesses to method `h` can appear anywhere within package `outerpkg`, including packages contained in it.

Example

A useful idiom to prevent clients of a class from constructing new instances of that class is to declare the class `abstract` and `sealed`:

```
object m {
  abstract sealed class C (x: Int) {
    def nextC = new C(x + 1) {}
  }
  val empty = new C(0) {}
}
```

For instance, in the code above clients can create instances of class `m.C` only by calling the `nextC` method of an existing `m.C` object; it is not possible for clients to create objects of class `m.C` directly. Indeed the following two lines are both in error:

```
new m.C(0)    // **** error: C is abstract, so it cannot be instantiated.
new m.C(0) {} // **** error: illegal inheritance from sealed class.
```

A similar access restriction can be achieved by marking the primary constructor `private` (example).

Class Definitions

```
TmplDef           ::=  'class' ClassDef
ClassDef          ::=  id [TypeParamClause] {Annotation}
                       [AccessModifier] ClassParamClauses ClassTemplateOpt
ClassParamClauses ::=  {ClassParamClause}
                       [[nl] '(' implicit ClassParams ')']
ClassParamClause  ::=  [nl] '(' [ClassParams] ')'
ClassParams       ::=  ClassParam {',' ClassParam}
ClassParam        ::=  {Annotation} {Modifier} [('val' | 'var')]
                       id [':' ParamType] ['=' Expr]
ClassTemplateOpt  ::= 'extends' ClassTemplate | [['extends'] TemplateBody]
```

The most general form of class definition is

```
class $c$[$\mathit{tps}\,$] $as$ $m$($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$) extends $t$
```

Here,

- *c* is the name of the class to be defined.
- *tps* is a non-empty list of type parameters of the class being defined. The scope of a type parameter is the whole class definition including the type parameter section itself. It is illegal to define two type parameters with the same name. The type parameter section `[$\mathit{tps}\,$]` may

be omitted. A class with a type parameter section is called *polymorphic*, otherwise it is called *monomorphic*.

- *as* is a possibly empty sequence of annotations. If any annotations are given, they apply to the primary constructor of the class.
- *m* is an access modifier such as `private` or `protected`, possibly with a qualification. If such an access modifier is given it applies to the primary constructor of the class.
- $(ps\_1) \ldots (ps\_n)$ are formal value parameter clauses for the *primary constructor* of the class. The scope of a formal value parameter includes all subsequent parameter sections and the template t. However, a formal value parameter may not form part of the types of any of the parent classes or members of the class template t. It is illegal to define two formal value parameters with the same name.

  If a class has no formal parameter section that is not implicit, an empty parameter section `()` is assumed.

  If a formal parameter declaration $x : T$ is preceded by a `val` or `var` keyword, an accessor (getter) definition for this parameter is implicitly added to the class.

  The getter introduces a value member x of class c that is defined as an alias of the parameter. If the introducing keyword is `var`, a setter accessor `$x$_=` is also implicitly added to the class. In invocation of that setter `$x$_=($e$)` changes the value of the parameter to the result of evaluating e.

  The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s). When access modifiers are given for a parameter, but no `val` or `var` keyword, `val` is assumed. A formal parameter prefixed by `val` or `var` may not at the same time be a call-by-name parameter.
- t is a template of the form

  ```
  $sc$ with $mt_1$ with $\ldots$ with $mt_m$ { $\mathit{stats}$ } // $m \geq 0$
  ```

  which defines the base classes, behavior and initial state of objects of the class. The extends clause `extends $sc$ with $mt_1$ with $\ldots$ with $mt_m$` can be omitted, in which case `extends scala.AnyRef` is assumed. The class body `{ $\mathit{stats}$ }` may also be omitted, in which case the empty body `{}` is assumed.

This class definition defines a type `$c$[$\mathit{tps}\,$]` and a constructor which when applied to parameters conforming to types *ps* initializes instances of type `$c$[$\mathit{tps}\,$]` by evaluating the template t.

Example – `val` and `var` parameters

The following example illustrates `val` and `var` parameters of a class `C`:

```
class C(x: Int, val y: String, var z: List[String])
val c = new C(1, "abc", List())
c.z = c.y :: c.z
```

Example – Private Constructor

The following class can be created only from its companion module.

```
object Sensitive {
  def makeSensitive(credentials: Certificate): Sensitive =
    if (credentials == Admin) new Sensitive()
    else throw new SecurityViolationException
}
class Sensitive private () {
  ...
}
```

Constructor Definitions

```
FunDef         ::= ‘this’ ParamClause ParamClauses
                   (‘=’ ConstrExpr | [nl] ConstrBlock)
ConstrExpr     ::= SelfInvocation
                 |  ConstrBlock
ConstrBlock    ::= ‘{’ SelfInvocation {semi BlockStat} ‘}’
SelfInvocation ::= ‘this’ ArgumentExprs {ArgumentExprs}
```

A class may have additional constructors besides the primary constructor. These are defined by constructor definitions of the form `def this($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$) = $e$`. Such a definition introduces an additional constructor for the enclosing class, with parameters as given in the formal parameter lists $ps_1, \ldots, ps_n$, and whose evaluation is defined by the constructor expression e. The scope of each formal parameter is the subsequent parameter sections and the constructor expression e. A constructor expression is either a self constructor invocation `this($\mathit{args}_1$)$\ldots$($\mathit{args}_n$)` or a block which begins with a self constructor invocation. The self constructor invocation must construct a generic instance of the class. I.e. if the class in question has name C and type parameters `[$\mathit{tps}\,$]`, then a self constructor invocation must generate an instance of `$C$[$\mathit{tps}\,$]`; it is not permitted to instantiate formal type parameters.

The signature and the self constructor invocation of a constructor definition are type-checked and evaluated in the scope which is in effect at the point of the enclosing class definition, augmented by any type parameters of the enclosing class and by any early definitions of the enclosing template. The rest of the constructor expression is type-checked and evaluated as a function body in the current class.

If there are auxiliary constructors of a class C, they form together with C's primary constructor an overloaded constructor definition. The usual rules for overloading resolution apply for constructor invocations of C, including for the self constructor invocations in the constructor expressions themselves. However, unlike other methods, constructors are never inherited. To prevent infinite cycles of constructor invocations, there is the restriction that every self constructor invocation must refer to a constructor definition which precedes it (i.e. it must refer to either a preceding auxiliary constructor or the primary constructor of the class).

Example

Consider the class definition

```
class LinkedList[A]() {
  var head: A = _
  var tail: LinkedList[A] = null
  def this(head: A) = { this(); this.head = head }
  def this(head: A, tail: LinkedList[A]) = { this(head); this.tail = tail }
}
```

This defines a class `LinkedList` with three constructors. The second constructor constructs an singleton list, while the third one constructs a list with a given head and tail.

Case Classes

```
TmplDef  ::=  'case' 'class' ClassDef
```

If a class definition is prefixed with `case`, the class is said to be a *case class*.

A case class is required to have a parameter section that is not implicit. The formal parameters in the first parameter section are called *elements* and are treated specially. First, the value of such a parameter can be extracted as a field of a constructor pattern. Second, a `val` prefix is implicitly added to such a parameter, unless the parameter already carries a `val` or `var` modifier. Hence, an accessor definition for the parameter is generated.

A case class definition of `$c$[$\mathit{tps}\,$]($\mathit{ps}_1\,$)$\ldots$($\mathit{ps}_n$)` with type parameters *tps* and value parameters *ps* implies the definition of a companion object, which serves as an extractor object. It has the following shape:

```
object $c$ {
  def apply[$\mathit{tps}\,$]($\mathit{ps}_1\,$)$\ldots$($\mathit{ps}_n$): $c$[$\mathit{tps}
  def unapply[$\mathit{tps}\,$]($x$: $c$[$\mathit{tps}\,$]) =
    if (x eq null) scala.None
    else scala.Some($x.\mathit{xs}_{11}, \ldots , x.\mathit{xs}_{1k}$)
}
```

Here, Ts stands for the vector of types defined in the type parameter section tps, each xs_i denotes the parameter names of the parameter section ps_i, and $xs\_11, \ldots, xs\_1k$ denote the names of all parameters in the first parameter section xs_1. If a type parameter section is missing in the class, it is also missing in the `apply` and `unapply` methods.

If the companion object c is already defined, the `apply` and `unapply` methods are added to the existing object. If the object c already has a matching `apply` (or `unapply`) member, no new definition is added. The definition of `apply` is omitted if class c is `abstract`.

If the case class definition contains an empty value parameter list, the `unapply` method returns a `Boolean` instead of an `Option` type and is defined as follows:

```
def unapply[$\mathit{tps}\,$]($x$: $c$[$\mathit{tps}\,$]) = x ne null
```

The name of the `unapply` method is changed to `unapplySeq` if the first parameter section ps_1 of c ends in a repeated parameter.

A method named `copy` is implicitly added to every case class unless the class already has a member (directly defined or inherited) with that name, or the class has a repeated parameter. The method is defined as follows:

```
def copy[$\mathit{tps}\,$]($\mathit{ps}'_1\,$)$\ldots$($\mathit{ps}'_n$): $c$[$\mathit{tps}\
```

Again, `$\mathit{Ts}$` stands for the vector of types defined in the type parameter section `$\mathit{tps}$` and each `$xs_i$` denotes the parameter names of the parameter section `$ps'_i$`. The value parameters `$ps'_{1,j}$` of first parameter list have the form `$x_{1,j}$:$T_{1,j}$=this.$x_{1,j}$`, the other parameters `$ps'_{i,j}$` of the `copy` method are defined as `$x_{i,j}$:$T_{i,j}$`. In all cases `$x_{i,j}$` and `$T_{i,j}$` refer to the name and type of the corresponding class parameter `$\mathit{ps}_{i,j}$`.

Every case class implicitly overrides some method definitions of class `scala.AnyRef` unless a definition of the same method is already given in the case class itself or a concrete definition of the same method is given in some base class of the case class different from `AnyRef`. In particular:

- Method `equals: (Any)Boolean` is structural equality, where two instances are equal if they both belong to the case class in question and they have equal (with respect to `equals`) constructor arguments (restricted to the class's *elements*, i.e., the first parameter section).
- Method `hashCode: Int` computes a hash-code. If the hashCode methods of the data structure members map equal (with respect to equals) values to equal hash-codes, then the case class hashCode method does too.
- Method `toString: String` returns a string representation which contains the name of the class and its elements.

Example

Here is the definition of abstract syntax for lambda calculus:

```
class Expr
case class Var   (x: String)          extends Expr
case class Apply (f: Expr, e: Expr)   extends Expr
case class Lambda(x: String, e: Expr) extends Expr
```

This defines a class `Expr` with case classes `Var`, `Apply` and `Lambda`. A call-by-value evaluator for lambda expressions could then be written as follows.

```
type Env = String => Value
case class Value(e: Expr, env: Env)

def eval(e: Expr, env: Env): Value = e match {
  case Var (x) =>
    env(x)
  case Apply(f, g) =>
    val Value(Lambda (x, e1), env1) = eval(f, env)
    val v = eval(g, env)
    eval (e1, (y => if (y == x) v else env1(y)))
  case Lambda(_, _) =>
    Value(e, env)
}
```

It is possible to define further case classes that extend type `Expr` in other parts of the program, for instance

```
case class Number(x: Int) extends Expr
```

This form of extensibility can be excluded by declaring the base class `Expr` `sealed`; in this case, all classes that directly extend `Expr` must be in the same source file as `Expr`.

Traits

```
TmplDef          ::=  'trait' TraitDef
TraitDef         ::=  id [TypeParamClause] TraitTemplateOpt
TraitTemplateOpt ::= 'extends' TraitTemplate | [['extends'] TemplateBody]
```

A *trait* is a class that is meant to be added to some other class as a mixin. Unlike normal classes, traits cannot have constructor parameters. Furthermore, no constructor arguments are passed to the superclass of the trait. This is not necessary as traits are initialized after the superclass is initialized.

Assume a trait D defines some aspect of an instance x of type C (i.e. D is a base class of C). Then the *actual supertype* of D in x is the compound type consisting of all the base classes in $\mathcal{L}(C)$ that succeed D. The actual supertype gives the context for resolving a `super` reference in a trait. Note that the actual supertype depends on the type to which the trait is added in a mixin composition; it is not statically known at the time the trait is defined.

If D is not a trait, then its actual supertype is simply its least proper supertype (which is statically known).

Example

The following trait defines the property of being comparable to objects of some type. It contains an abstract method `<` and default implementations of the other comparison operators `<=`, `>`, and `>=`.

```
trait Comparable[T <: Comparable[T]] { self: T =>
  def < (that: T): Boolean
  def <=(that: T): Boolean = this < that || this == that
  def > (that: T): Boolean = that < this
  def >=(that: T): Boolean = that <= this
}
```

Example

Consider an abstract class `Table` that implements maps from a type of keys `A` to a type of values `B`. The class has a method `set` to enter a new key / value pair into the table, and a method `get` that returns an optional value matching a given key. Finally, there is a method `apply` which is like `get`, except that it returns a given default value if the table is undefined for the given key. This class is implemented as follows.

```
abstract class Table[A, B](defaultValue: B) {
  def get(key: A): Option[B]
  def set(key: A, value: B): Unit
  def apply(key: A) = get(key) match {
    case Some(value) => value
    case None => defaultValue
  }
}
```

Here is a concrete implementation of the `Table` class.

```
class ListTable[A, B](defaultValue: B) extends Table[A, B](defaultValue) {
  private var elems: List[(A, B)] = Nil
  def get(key: A) = elems.find(_._1 == key).map(_._2)
  def set(key: A, value: B) = { elems = (key, value) :: elems }
}
```

Here is a trait that prevents concurrent access to the `get` and `set` operations of its parent class:

```
trait SynchronizedTable[A, B] extends Table[A, B] {
  abstract override def get(key: A): B =
    synchronized { super.get(key) }
  abstract override def set(key: A, value: B) =
    synchronized { super.set(key, value) }
}
```

Note that `SynchronizedTable` does not pass an argument to its superclass, `Table`, even though `Table` is defined with a formal parameter. Note also that the `super` calls in `SynchronizedTable`'s `get` and `set` methods statically refer to abstract methods in class `Table`. This is legal, as long as the calling method is labeled `abstract override`.

Finally, the following mixin composition creates a synchronized list table with strings as keys and integers as values and with a default value `0`:

```
object MyTable extends ListTable[String, Int](0) with SynchronizedTable[String, Int]
```

The object `MyTable` inherits its `get` and `set` method from `SynchronizedTable`. The `super` calls in these methods are re-bound to refer to the corresponding implementations in `ListTable`, which is the actual supertype of `SynchronizedTable` in `MyTable`.

Object Definitions

```
ObjectDef       ::=  id ClassTemplate
```

An *object definition* defines a single object of a new class. Its most general form is `object $m$ extends $t$`. Here, m is the name of the object to be defined, and t is a template of the form

```
$sc$ with $mt_1$ with $\ldots$ with $mt_n$ { $\mathit{stats}$ }
```

which defines the base classes, behavior and initial state of m. The extends clause `extends $sc$ with $mt_1$ with $\ldots$ with $mt_n$` can be omitted, in which case `extends scala.AnyRef` is assumed. The class body `{ $\mathit{stats}$ }` may also be omitted, in which case the empty body `{}` is assumed.

The object definition defines a single object (or: *module*) conforming to the template t. It is roughly equivalent to the following definition of a lazy value:

```
lazy val $m$ = new $sc$ with $mt_1$ with $\ldots$ with $mt_n$ { this: $m.type$ => $\mathit{s
```

Note that the value defined by an object definition is instantiated lazily. The `new $m$\$cls` constructor is evaluated not at the point of the object definition, but is instead evaluated the first time m is dereferenced during execution of the program (which might be never at all). An attempt to dereference m again during evaluation of the constructor will lead to an infinite loop or run-time error. Other threads trying to dereference m while the constructor is being evaluated block until evaluation is complete.

The expansion given above is not accurate for top-level objects. It cannot be because variable and method definition cannot appear on the top-level outside of a package object. Instead, top-level objects are translated to static fields.

Example

Classes in Scala do not have static members; however, an equivalent effect can be achieved by an accompanying object definition E.g.

```
abstract class Point {
  val x: Double
  val y: Double
  def isOrigin = (x == 0.0 && y == 0.0)
}
object Point {
  val origin = new Point() { val x = 0.0; val y = 0.0 }
}
```

This defines a class `Point` and an object `Point` which contains `origin` as a member. Note that the double use of the name `Point` is legal, since the class definition defines the name `Point` in the type name space, whereas the object definition defines a name in the term namespace.

This technique is applied by the Scala compiler when interpreting a Java class with static members. Such a class C is conceptually seen as a pair of a Scala class that contains all instance members of C and a Scala object that contains all static members of C.

Generally, a *companion module* of a class is an object which has the same name as the class and is defined in the same scope and compilation unit. Conversely, the class is called the *companion class* of the module.

Very much like a concrete class definition, an object definition may still contain declarations of abstract type members, but not of abstract term members.