Export Clauses

An export clause defines aliases for selected members of an object. Example:

```
class BitMap
class InkJet

class Printer:
  type PrinterType
  def print(bits: BitMap): Unit = ???
  def status: List[String] = ???

class Scanner:
  def scan(): BitMap = ???
  def status: List[String] = ???

class Copier:
  private val printUnit = new Printer { type PrinterType = InkJet }
  private val scanUnit = new Scanner

  export scanUnit.scan
  export printUnit.{status => _, *}

  def status: List[String] = printUnit.status ++ scanUnit.status
```

The two `export` clauses define the following *export aliases* in class `Copier`:

```
final def scan(): BitMap            = scanUnit.scan()
final def print(bits: BitMap): Unit = printUnit.print(bits)
final type PrinterType              = printUnit.PrinterType
```

They can be accessed inside `Copier` as well as from outside:

```
val copier = new Copier
copier.print(copier.scan())
```

An `export` clause has the same format as an import clause. Its general form is:

```
export path . { sel_1, ..., sel_n }
```

It consists of a qualifier expression `path`, which must be a stable identifier, followed by one or more selectors `sel_i` that identify what gets an alias. Selectors can be of one of the following forms:

- A *simple selector* `x` creates aliases for all eligible members of `path` that are named `x`.
- A *renaming selector* `x ⇒ y` creates aliases for all eligible members of `path` that are named `x`, but the alias is named `y` instead of `x`.
- An *omitting selector* `x ⇒ _` prevents `x` from being aliased by a subsequent wildcard selector.
- A *given selector* `given x` has an optional type bound `x`. It creates aliases for all eligible given instances that conform to either `x`, or `Any` if `x` is omitted, except for members that are named by a previous simple, renaming, or omitting selector.
- A *wildcard selector* `*` creates aliases for all eligible members of `path` except for given instances, synthetic members generated by the compiler and those members that are named by a previous simple, renaming, or omitting selector.
  Notes:
  - eligible construtor proxies are also included, even though they are synthetic members.
  - members created by an export are also included. They are created by the compiler, but are not considered synthetic.

A member is *eligible* if all of the following holds:

- its owner is not a base class of the class(*) containing the export clause,
- the member does not override a concrete definition that has as owner a base class of the class containing the export clause.
- it is accessible at the export clause,
- it is not a constructor, nor the (synthetic) class part of an object,
- it is a given instance (declared with `given`) if and only if the export is from a *given selector*.

It is a compile-time error if a simple or renaming selector does not identify any eligible members.

Type members are aliased by type definitions, and term members are aliased by method definitions. Export aliases copy the type and value parameters of the

members they refer to. Export aliases are always `final`. Aliases of given instances are again defined as givens (and aliases of old-style implicits are `implicit`). Aliases of extensions are again defined as extensions. Aliases of inline methods or values are again defined `inline`. There are no other modifiers that can be given to an alias. This has the following consequences for overriding:

- Export aliases cannot be overridden, since they are final.
- Export aliases cannot override concrete members in base classes, since they are not marked `override`.
- However, export aliases can implement deferred members of base classes.

Export aliases for public value definitions that are accessed without referring to private values in the qualifier path are marked by the compiler as "stable" and their result types are the singleton types of the aliased definitions. This means that they can be used as parts of stable identifier paths, even though they are technically methods. For instance, the following is OK:

```
class C { type T }
object O { val c: C = ... }
export O.c
def f: c.T = ...
```

Restrictions:

1. Export clauses can appear in classes or they can appear at the top-level. An export clause cannot appear as a statement in a block.

2. If an export clause contains a wildcard or given selector, it is forbidden for its qualifier path to refer to a package. This is because it is not yet known how to safely track wildcard dependencies to a package for the purposes of incremental compilation.

3. An export renaming hides un-renamed exports matching the target name. For instance, the following clause would be invalid since `B` is hidden by the renaming `A as B`.

   ```
   export {A as B, B}        // error: B is hidden
   ```

4. Renamings in an export clause must have pairwise different target names. For

instance, the following clause would be invalid:

```
export {A as C, B as C}   // error: duplicate renaming
```

5. Simple renaming exports like

```
export status as stat
```

are not supported yet. They would run afoul of the restriction that the exported `a` cannot be already a member of the object containing the export. This restriction might be lifted in the future.

(*) Note: Unless otherwise stated, the term "class" in this discussion also includes object and trait definitions.

Motivation

It is a standard recommendation to prefer composition over inheritance. This is really an application of the principle of least power: Composition treats components as blackboxes whereas inheritance can affect the internal workings of components through overriding. Sometimes the close coupling implied by inheritance is the best solution for a problem, but where this is not necessary the looser coupling of composition is better.

So far, object-oriented languages including Scala made it much easier to use inheritance than composition. Inheritance only requires an `extends` clause whereas composition required a verbose elaboration of a sequence of forwarders. So in that sense, object-oriented languages are pushing programmers to a solution that is often too powerful. Export clauses redress the balance. They make composition relationships as concise and easy to express as inheritance relationships. Export clauses also offer more flexibility than extends clauses since members can be renamed or omitted.

Export clauses also fill a gap opened by the shift from package objects to top-level definitions. One occasionally useful idiom that gets lost in this shift is a package object inheriting from some class. The idiom is often used in a facade like pattern, to make members of internal compositions available to users of a package. Top-level definitions are not wrapped in a user-defined object, so they can't inherit anything. However, top-level definitions can be export clauses, which supports the facade design pattern in a safer and more flexible way.

Syntax changes:

```
TemplateStat      ::=  ...
                     |  Export

TopStat           ::=  ...
                     |  Export
Export            ::=  'export' ImportExpr {',' ImportExpr}
ImportExpr        ::=  SimpleRef {'.' id} '.' ImportSpec
ImportSpec        ::=  NamedSelector
                     |  WildcardSelector
                     | '{' ImportSelectors) '}'
NamedSelector     ::=  id ['as' (id | '_')]
WildCardSelector  ::=  '*' | 'given' [InfixType]
ImportSelectors   ::=  NamedSelector [',' ImportSelectors]
                     |  WildCardSelector {',' WildCardSelector}
```

Elaboration of Export Clauses

Export clauses raise questions about the order of elaboration during type checking. Consider the following example:

```
class B { val c: Int }
object a { val b = new B }
export a.*
export b.*
```

Is the `export b.*` clause legal? If yes, what does it export? Is it equivalent to `export a.b.*`? What about if we swap the last two clauses?

```
export b.*
export a.*
```

To avoid tricky questions like these, we fix the elaboration order of exports as follows.

Export clauses are processed when the type information of the enclosing object or class is completed. Completion so far consisted of the following steps:

1. Elaborate any annotations of the class.
2. Elaborate the parameters of the class.
3. Elaborate the self type of the class, if one is given.
4. Enter all definitions of the class as class members, with types to be completed on demand.
5. Determine the types of all parents of the class.

With export clauses, the following steps are added:

6. Compute the types of all paths in export clauses.
7. Enter export aliases for the eligible members of all paths in export clauses.

It is important that steps 6 and 7 are done in sequence: We first compute the types of *all* paths in export clauses and only after this is done we enter any export aliases as class members. This means that a path of an export clause cannot refer to an alias made available by another export clause of the same class.

‹ Univer… Opaqu… ›