CAT calls (Changed Availability or Type)

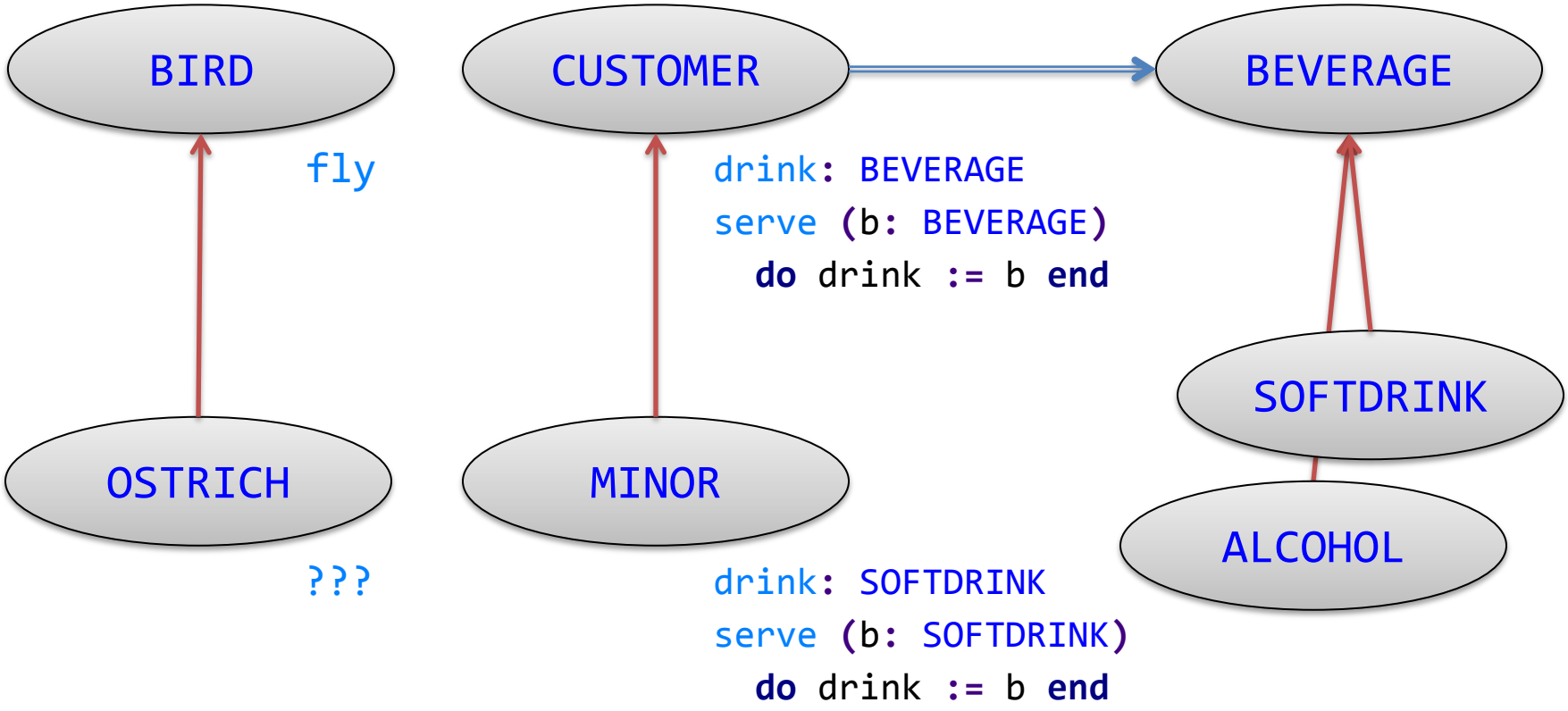
these slides contain advanced material and are optional

Changed Availability or Type (CAT)

- Changed Availability (export status)
  - Lifting restrictions
  - Tighten restirctions
- Changed Type (argument or result type)
  - Covariant redefinition (changing to a narrower type)
  - Contravariant redefinition (changing to a wider type)

Why change availabilty or type?

- Greater expressiveness when modeling



Changed availability

- Changed Availability (export status)
  - Lifting restrictions

```
class B
Inherit
    A
        export {ANY} f end
end
```

  - Tighten restirctions

```
class C
Inherit
    A
        export {NONE} f end
end
```

```
class A
feature {X}
    f
end
```

Possible problems

```
class X
feature
  make
    local
      a: A
    do
      a := create {B}
      a.f

      a := create {C}
      a.f
    end
end
```

{B}.f is available to all classes, including X

{C}.f should not be available to X

```
class A
feature {X}
    f
end

class B
inherit A
    export {ANY} f end
end

class C
inherit A
    export {NONE} f end
end
```

Solution to availability problem

- When inheriting conformingly, restricting export status is not allowed anymore (Eiffel standard 2006)
  - EiffelStudio still allows it for backwards compatibility

- When inheriting non-conformingly, restricting export status is allowed

```
class A
feature {X}
    f
end
```

```
class C
Inherit {NONE}
    A
        export {NONE} f end
end
```

```
local
    a: A
do
    a := create {C}
    a.f
end
```

Invalid assignment.
C does not conform to A

Changed type: covariant

```
class A
feature
  f (a: ANY): ANY
    deferred
    end
end
```

```
class B
inherit A redefine f end
feature
  f (a: STRING): ANY
    do
      a.to_upper
      create Result
    end
end
```

Covariant redefinition of argument type: Call to invalid feature.

```
class X
feature
  make
    local
      a: A
    do
      a := create {B}
      a.f (1).do_nothing
      a := create {C}
      a.f (1).do_nothing
    end
end
```

```
class C
inherit A redefine f end
feature
  f (a: ANY): STRING
    do
      a.do_nothing
      create Result.
          make_from_string (a.out)
    end
end
```

Covariant redefinition of result type: Call Ok.

Changed type: contravariant

```
class A
feature
  f (a: STRING): STRING
    deferred
    end
end
```

```
class B
inherit A redefine f end
feature
  f (a: STRING): ANY
    do
      a.to_upper
      create Result
    end
end
```

Contravariant redefinition of result type: Call to invalid feature.

```
class X
feature
  make
    local
      a: A
    do
      a := create {B}
      a.f ("abc").append ("def")
      a := create {C}
      a.f ("abc").append ("def")
    end
end
```

```
class C
inherit A redefine f end
feature
  f (a: ANY): STRING
    do
      a.do_nothing
      create Result.
          make_from_string (a.out)
    end
end
```

Contravariant redefinition of argument type: Call Ok.

ANY.is_equal

- Routine {ANY}.is_equal has anchored argument:

```
is_equal (other: like Current): BOOLEAN
  do ... end
```

- This is a covariant redefinition in every type
- You should never use is_equal directly
- Use the ~-operator, which checks the dynamic type of both entities before calling is_equal

```
f (a, b: ANY): BOOLEAN
  do
    Result := a.is_equal (b)
    Result := a ~ b
  end
```

could be invalid at runtime

always safe

Changed type: summary

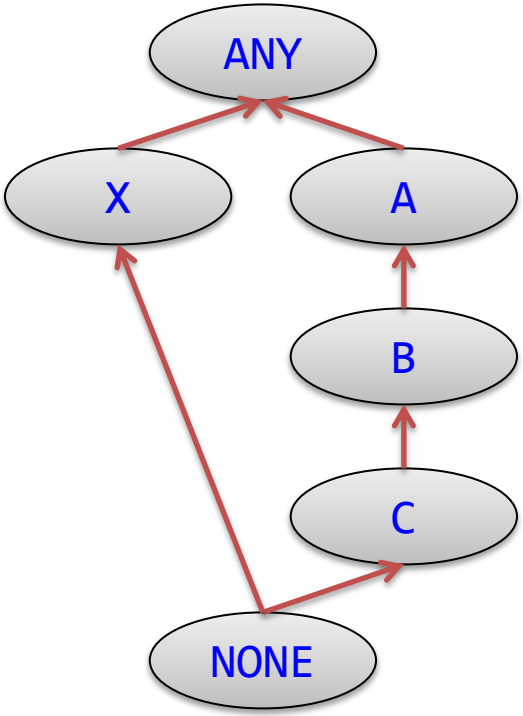
- It is safe to:
  - Change result type covariantly
  - Change argument type contravariantly
- It is unsafe to
  - Change result type contravariantly
  - Change argument type covariantly

- Eiffel allows
  - Covariant change of result type (safe)
  - Covariant change of argument type (unsafe)

CAT as contracts

```
class PARENT
feature {X}
  f (a: B): B
end
```

```
class CHILD
inherit PARENT
feature {NONE}
  f (a: C): A
end
```




```
class PARENT
feature
  f (a: ANY): ANY
    require
      type_of (Caller).conforms_to ({X})
      type_of (a).conforms_to ({B})
    ensure
      type_of (Result).conforms_to ({B})
end
```

```
class CHILD inherit PARENT
feature
  f (a: ANY): ANY
    require
      type_of (Caller).conforms_to ({NONE})
      type_of (a).conforms_to ({C})
    ensure
      type_of (Result).conforms_to ({A})
end
```

Stronger precondition

Weaker postcondition

Generic conformance

```
class APPLICATION
feature
  make
    local
      any_list: LIST [ANY]
      string_list: LIST [STRING]
      integer_list: LIST [INTEGER]
    do
      string_list := any_list        ×
      string_list := integer_list    ×
      integer_list := any_list       ×
      integer_list := string_list    ×
      any_list := string_list        ✓
      any_list := integer_list       ✓
    end
end
```

Changed types due to generics

```
class LIST [G]
feature
    put (a: G) do end
    first: G
end

interface class LIST [ANY]
feature
    put (a: ANY)
    first: ANY
end

interface class LIST [STRING]
feature
    put (a: STRING)
    first: STRING
end
```

LIST [STRING] conforms to LIST [ANY], thus the changed type in the argument and result are like a covariant redefinition.

Problems with generics

```
class APPLICATION
feature
  make
    local
      any_list: LIST [ANY]
      string_list: LIST [STRING]
    do
      create string_list.make
      any_list := string_list
      any_list.put (1)
      string_list.first.to_upper
    end
end
```

Wrong element type!

Different solutions for generics

- Novariant conformance
  - No conformance between generics of different type
  - Used by C#
- Usage-site variance
  - Specify conformance for each generic derivation
  - Used by Java
- Definition-site variance
  - Specify conformance for each generic class
  - Implemented by CLR

Possible solution: Type intervals

- All types have an upper and lower bound

```
c: CUSTOMER..MINOR
```

- Special abbreviations:

```
c1: CUSTOMER
c2: frozen CUSTOMER
```

```
c1: CUSTOMER..NONE
c2: CUSTOMER.. CUSTOMER
```

- Calls on any entity have to be valid for any possible type in the interval

```
c: CUSTOMER..CUSTOMER
c.serve (vodka)
```

Valid

```
c: CUSTOMER..MINOR
c.serve (vodka)
```

Invalid, since invalid for MINOR

Possible solution: usage-site variance

- Conformence of different generic instantiations require **`variant`** mark

```
justin: MINOR
bus: LIST [CUSTOMER]
vbus: LIST [variant CUSTOMER]
school_bus: LIST [MINOR]
```

- You can't call features on entities using the **`variant`** mark that have a formal generic parameter type.

```
bus := school_bus
bus.extend (justin)
```

Invalid

Ok

```
vbus := school_bus
vbus.extend (justin)
```

Ok

Invalid