22c181:
Formal Methods in Software Engineering

The University of Iowa

Spring 2008

Introduction to OCL

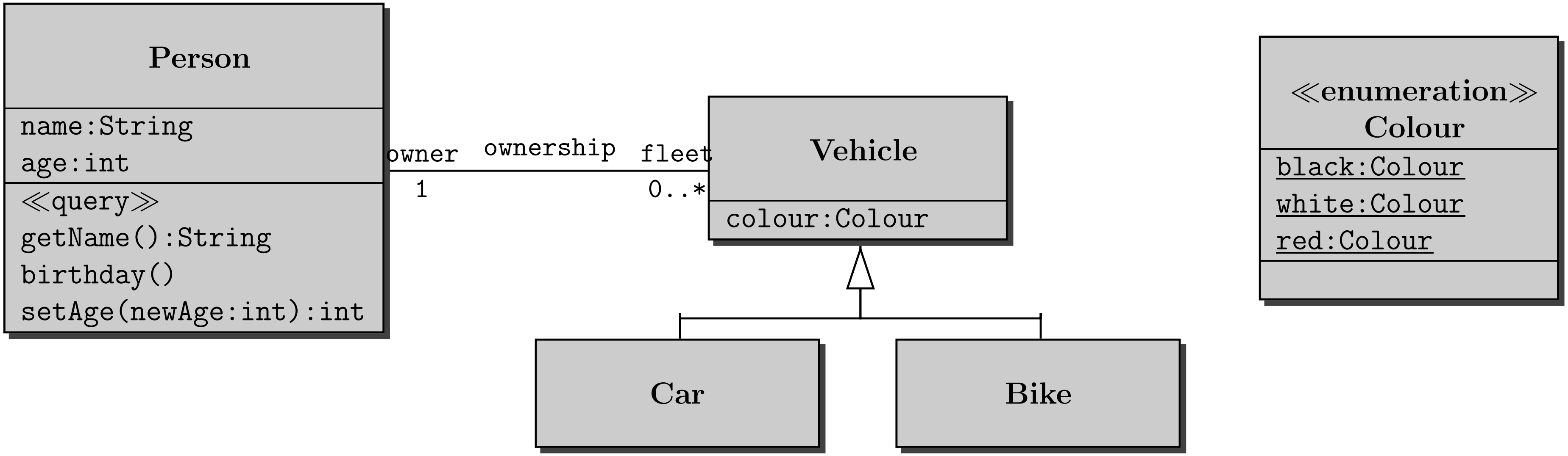
Contents

Object Constraint Language (OCL)

- Part of the UML standard
- Formal Specification Language

  Standardized formal semantics from OCL 2.0 onwards
- In this course: OCL 1.5
  - Semantics by mapping to typed FOL
  - Not all features realized, some extra features
- OCL syntax less mathematical,
  more programming language-oriented than Z, RSL, FOL, etc.
- Why OCL? UML is not expressive enough!

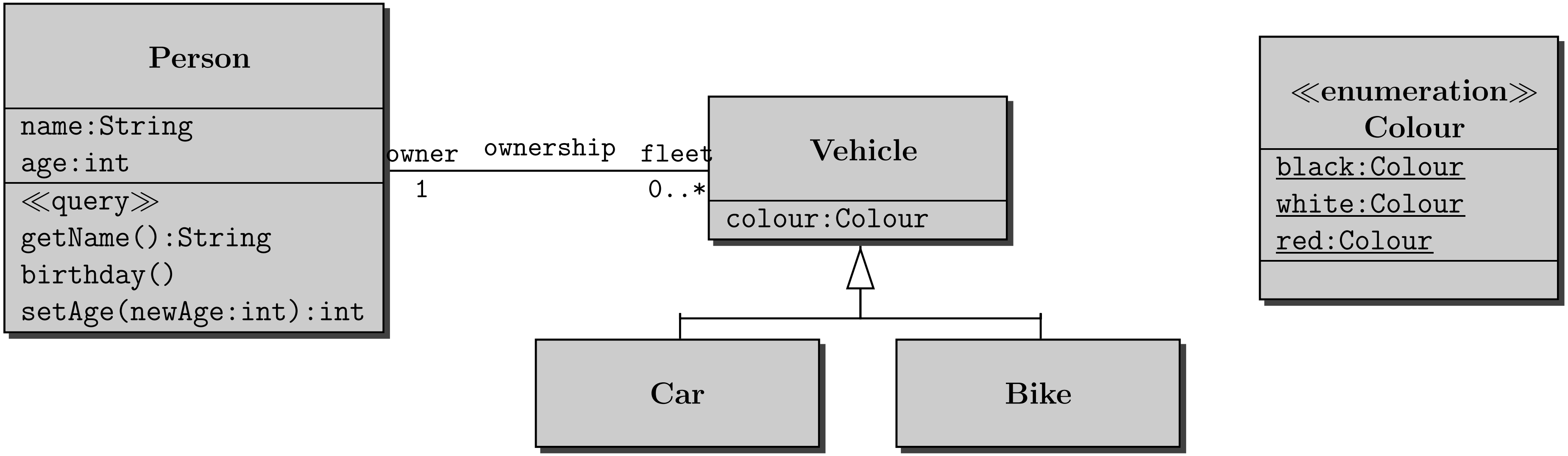
UML is not enough . . .



- **How old must a car owner be?**
- **How to express that a person can own at most own one black car?**
- **How to specify that value of age is `i` after calling `setAge(i)`?**

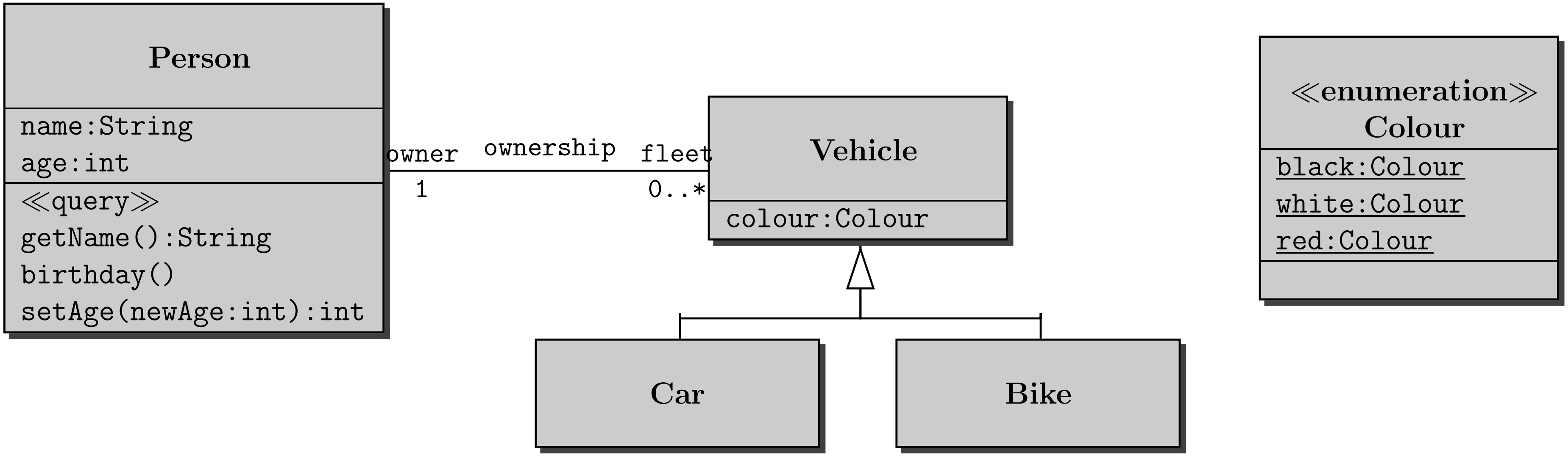
UML unsuitable to express semantics of design

Some OCL examples I



“A vehicle owner must be at least 18 years old”:

Some OCL examples I

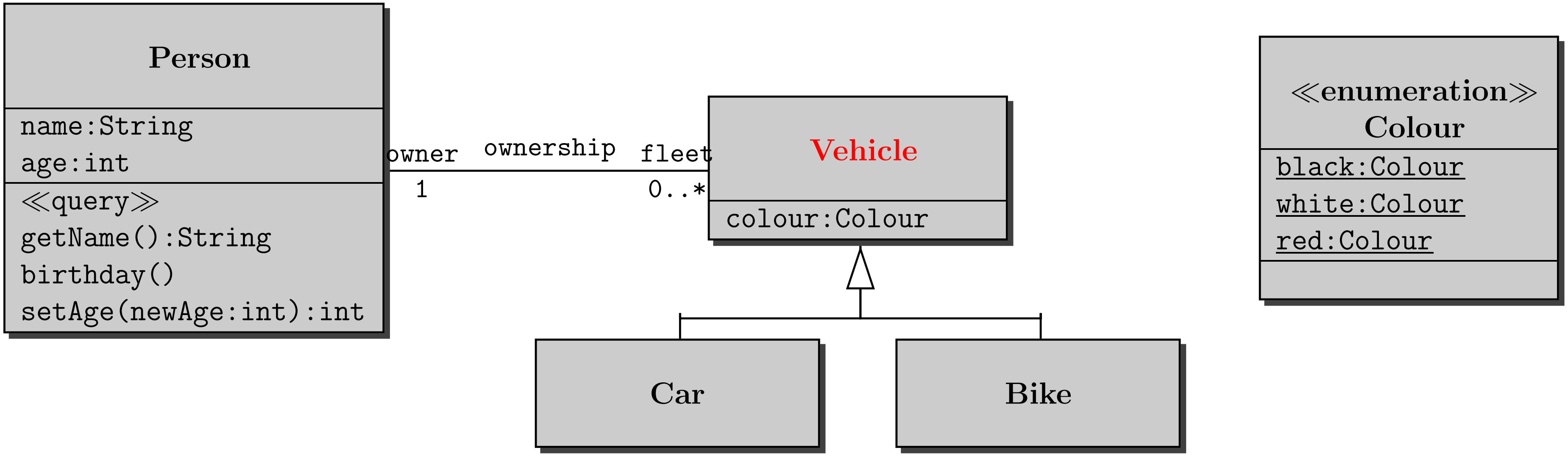


"A vehicle owner must be at least 18 years old":

context **Vehicle**

inv **:** **self. owner. age** $>=$ **18**

Some OCL examples I

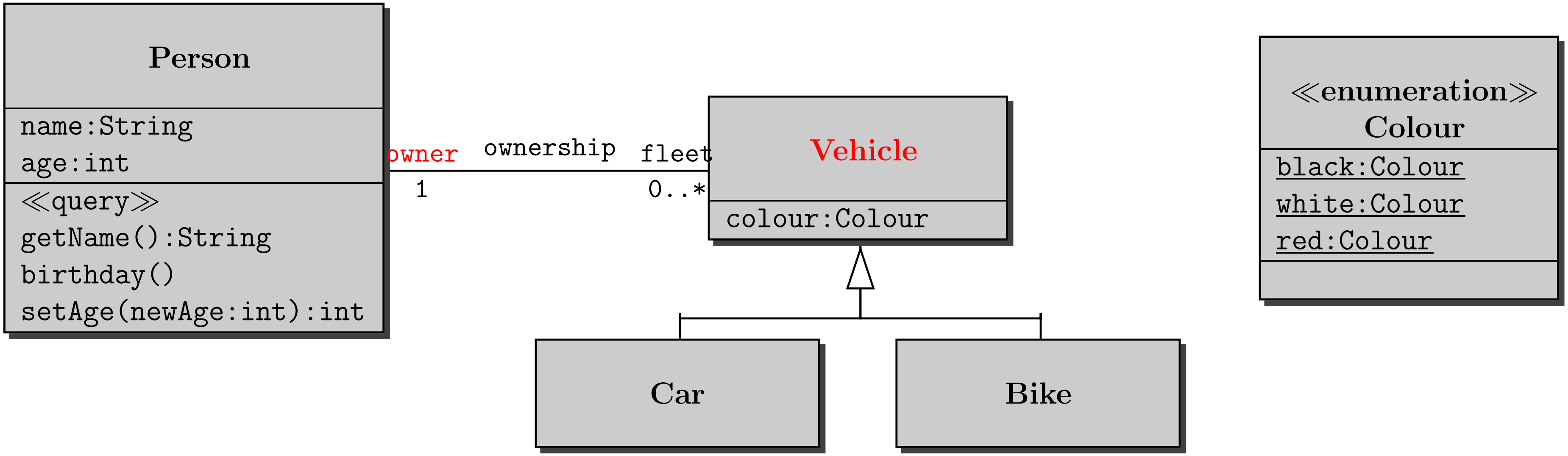


"A vehicle owner must be at least 18 years old":

context **Vehicle** **- - context declaration for all instances of this class**

inv : self. owner. age $>=$ 18 **- - 'self' is like JAVA's 'this'**

Some OCL examples I

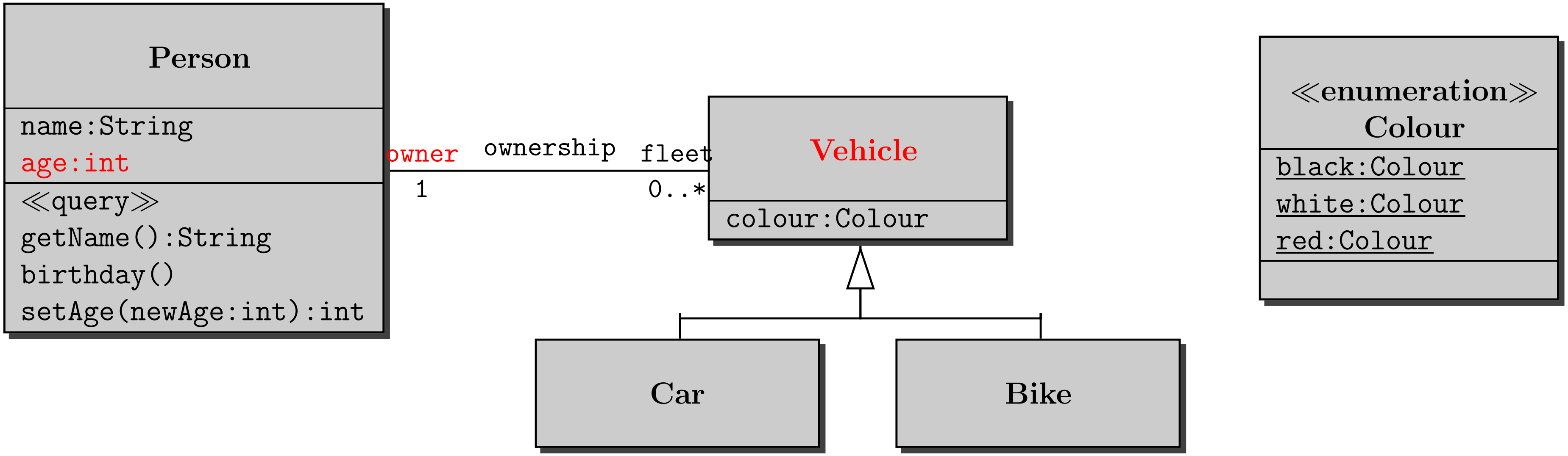


"A vehicle owner must be at least 18 years old":

context **Vehicle**

inv : **self. owner. age $>=$ 18** **- - navigate to instance of supplier**

Some OCL examples I

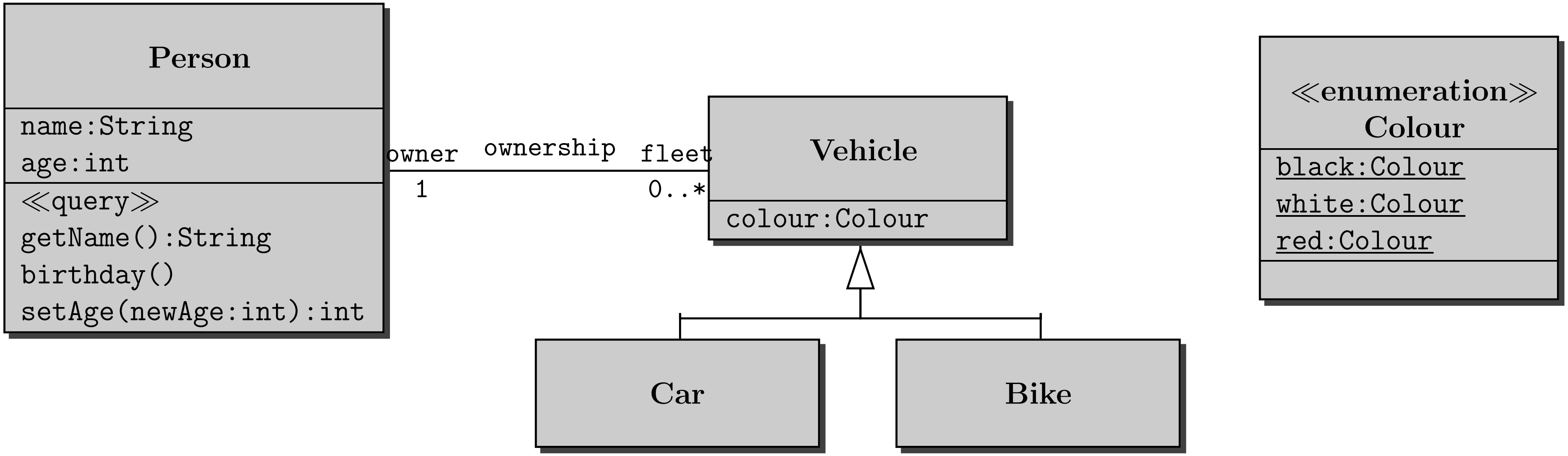


"A vehicle owner must be at least 18 years old":

context **Vehicle**

inv : **self. owner. age** $>=$ **18**

Some OCL examples I

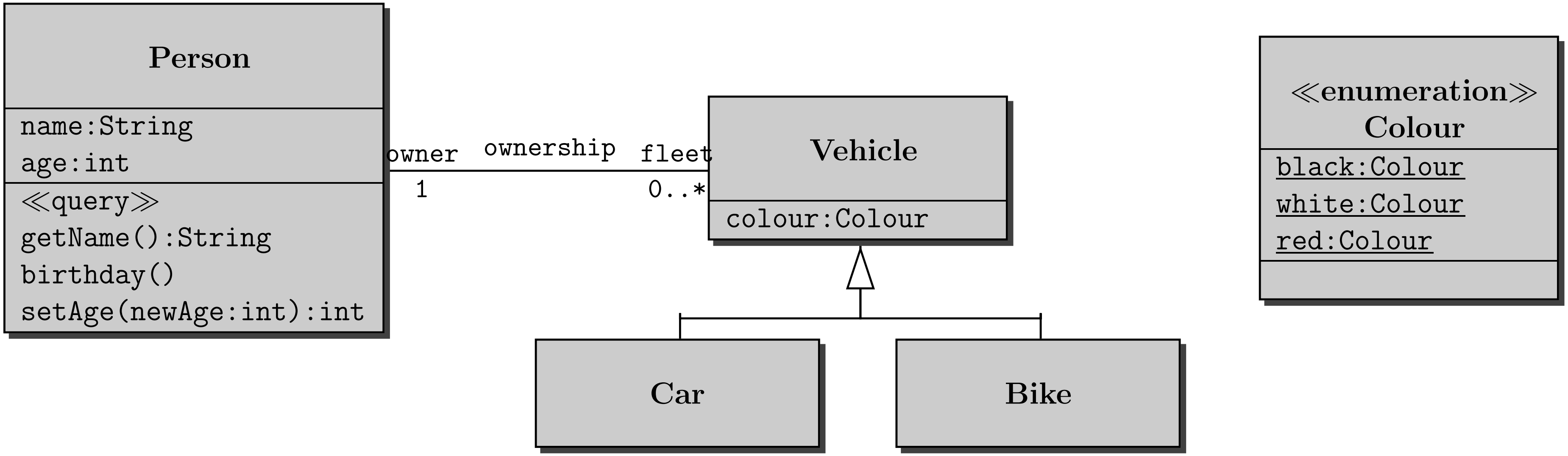


"A vehicle owner must be at least 18 years old":

context **Vehicle**

inv **:** **self. owner. age** $>=$ **18**

Some OCL examples I



"A vehicle owner must be at least 18 years old":

context **Vehicle**

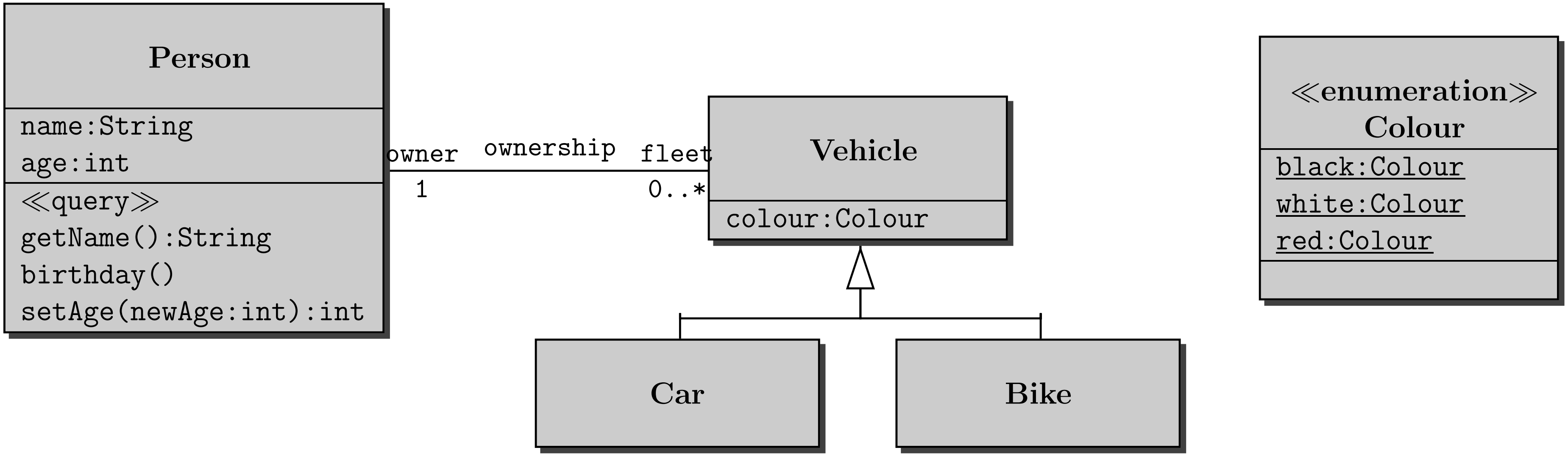
inv : **self. owner. age** $>=$ **18**

What does that mean, instead? Relation between the constraints?

context **Person**

inv : **self.age** $>=$ **18**

Some OCL examples I



“A vehicle owner must be at least 18 years old”:

context **Vehicle**

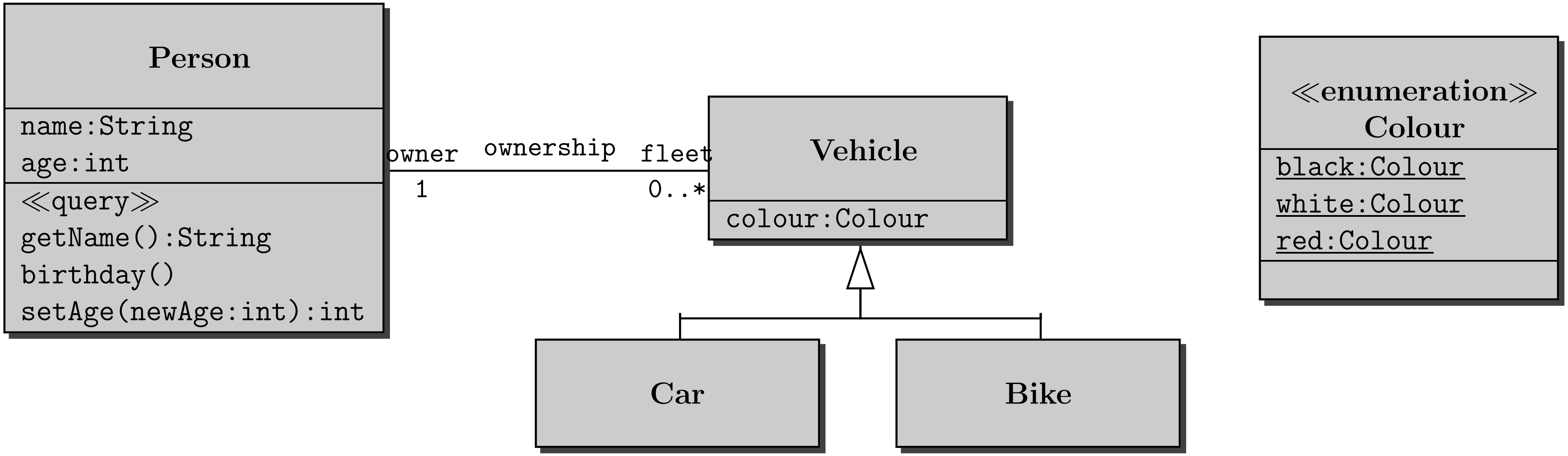
inv : **self. owner. age** $>=$ **18**

“A car owner must be at least 18 years old”:

context **Car**

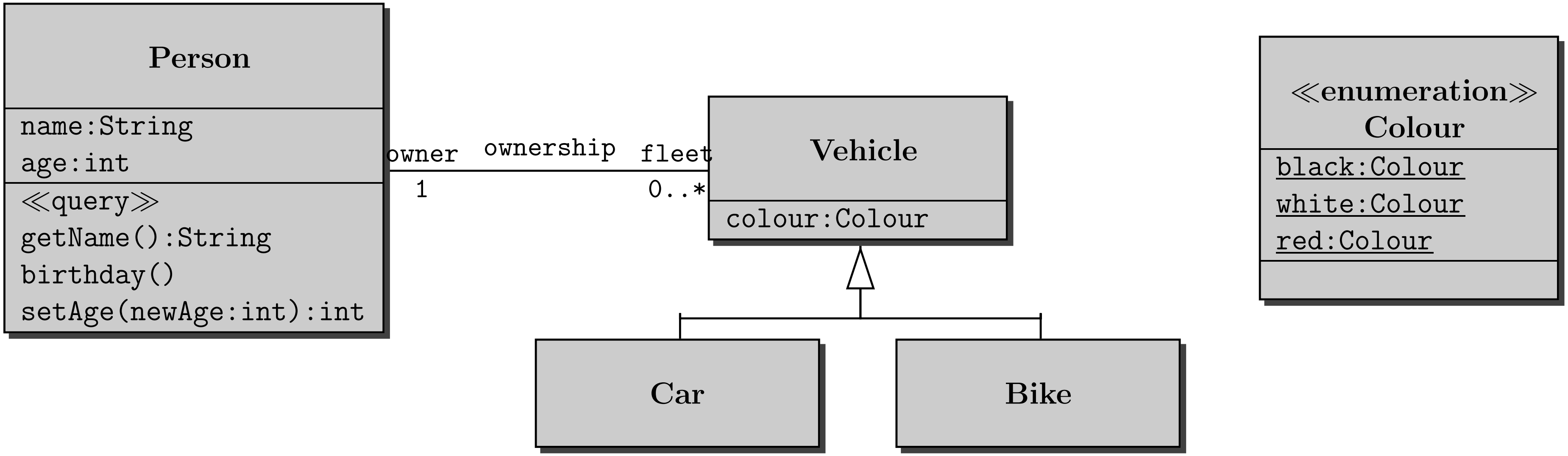
inv : **self.owner.age** $>=$ **18**

Some OCL examples II



“No person owns more than 3 vehicles”:

Some OCL examples II



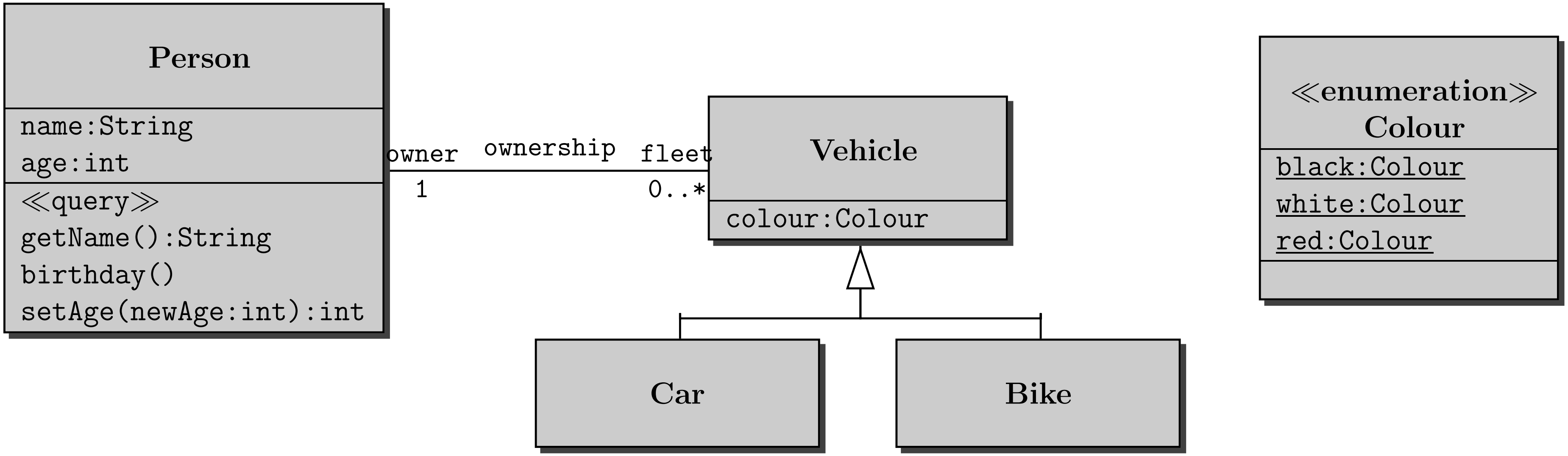
“No person owns more than 3 vehicles”:

context **Person**

inv : **self.fleet->size() <= 3**

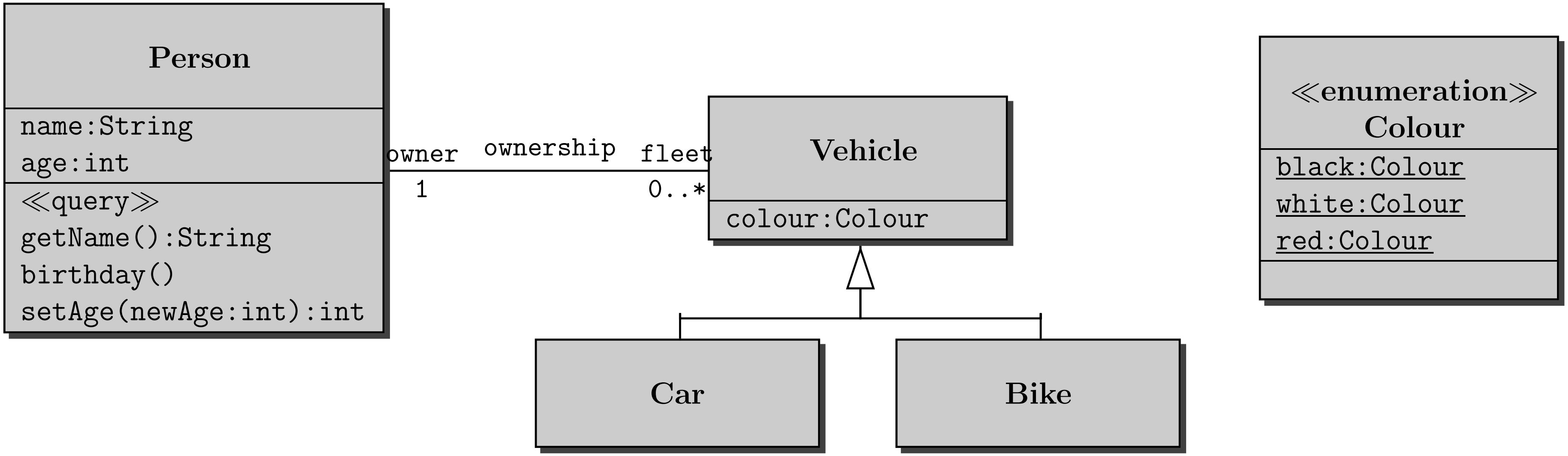
or change multiplicity

Some OCL examples II



“All vehicles of a person are black”:

Some OCL examples II

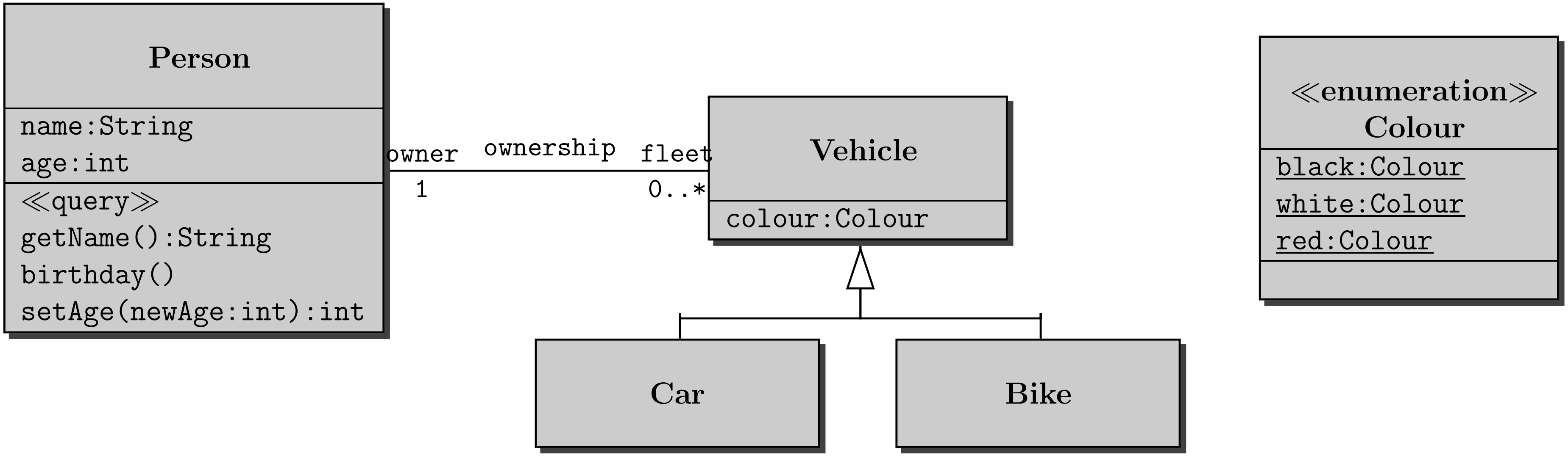


“All vehicles of a person are black”:

context **Person**

inv **: self.fleet–>forAll(v | v.colour = Colour.black)**

Some OCL examples II



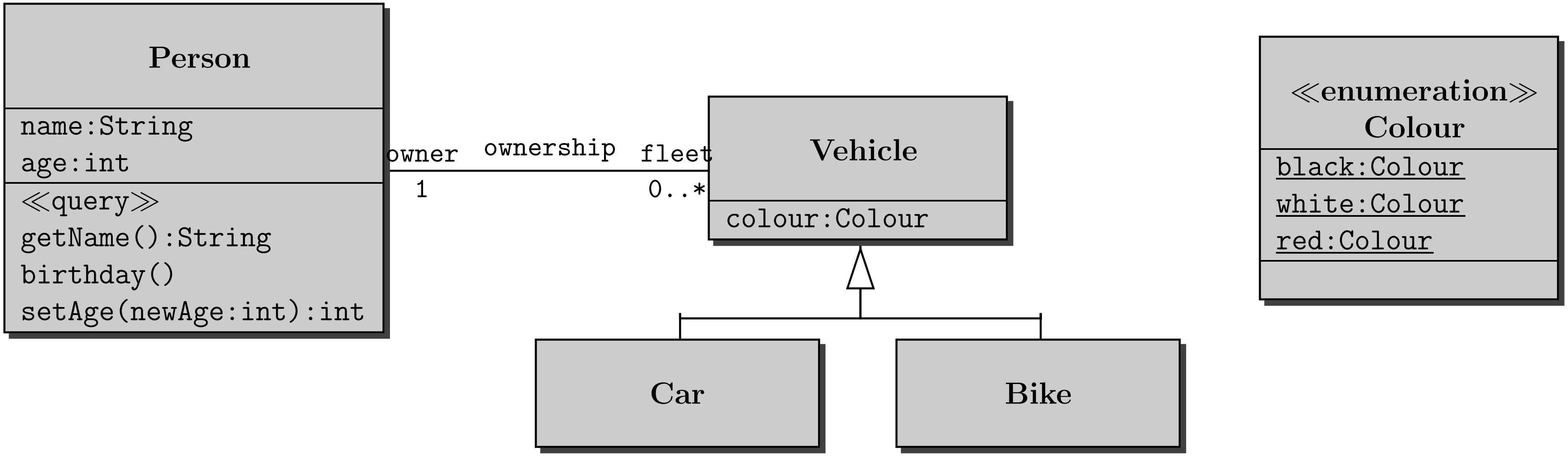
“All vehicles of a person are black”:

context **Person**

inv **: self.fleet–>forAll(v | v.colour = Colour.black)**

“No person owns more than 3 black vehicles”:

Some OCL examples II



“All vehicles of a person are black”:

context **Person**

inv : **self.fleet–>forAll(v | v.colour = Colour.black)**

“No person owns more than 3 black vehicles”:

context **Person**

inv : **self.fleet–>select(v | v.colour = Colour.black)–>size() <= 3**

The Classifier Context

context [instanceName :] classPath – – class from UML model

  inv [invariantName] : oclExpression

context aCar:Car

  inv minimumAge: aCar.owner.age $>=$ 18

- Class classPath is context of invariant constraint
- Invariant must hold for all instances of classPath at all times

  Instances can be named invariantName (not in Together)
- May declare invariantName for the constraint (not in Together)
- Type of oclExpression must be Boolean

The Classifier Context

context [instanceName :] classPath – – class from UML model

inv [invariantName] : oclExpression

context [instanceName :] classPath

inv [invariantName$_1$] : oclExpression$_1$

...

...

inv [invariantName$_n$] : oclExpression$_n$

More than one invariant can be declared in same context

When Do Invariants Hold?

Consider $\mathrm{insert}()$ **operation for** List **type with attribute** $\mathrm{length} : \mathrm{int}$

- **Assume the invariant of** List **states that**

  the number of nodes in a list is equal to the value of length

- **During execution of** $\mathrm{insert}()$ **usually the invariant is violated**

Therefore, semantics of invariants in KeY and OCL:

Invariants hold at all times before and after execution of operations

How to relax this rigid requirement is topic of active research

The Operator Context: Contract

Specifying the semantics of operations: their contract

context [instanceName :]

classPath ::opName(p_1: type$_1$; . . . ;p_k: type$_k$)[:resultType]

{pre [preName] : oclExpression }

{post [postName] : oclExpression }

The Operator Context: Contract

Specifying the semantics of operations: their contract

context [instanceName :]

classPath ::opName(p_1: type$_1$; . . . ;p_k: type$_k$)[:resultType]

{pre [preName] : oclExpression }

{post [postName] : oclExpression }

Example

"Calling getName() returns the current value of the attribute name"

context Person::getName():String

post : result = name

Special variable result contains return value, has type resultType

Together 6.2 Syntax for OCL Context Declarations

Classifiers

```
/**
 * @invariants OCLExpression
 */
```

Operators

```
/**
 * @preconditions  OCLExpression
 * @postconditions OCLExpression
 */
```

At most one may be present, connect multiple conditions with and**.**

Write constraints in `.java` file directly before feature they apply to.

Design by Contract

Pre-/postconditions like clauses in a contract about an operation

If the caller fulfills the precondition before the operation is called,

then the called object ensures the postcondition to hold after execution of the operation

Design by Contract

Pre-/postconditions like clauses in a contract about an operation

If the caller fulfills the precondition before the operation is called,

then the called object ensures the postcondition to hold after execution of the operation

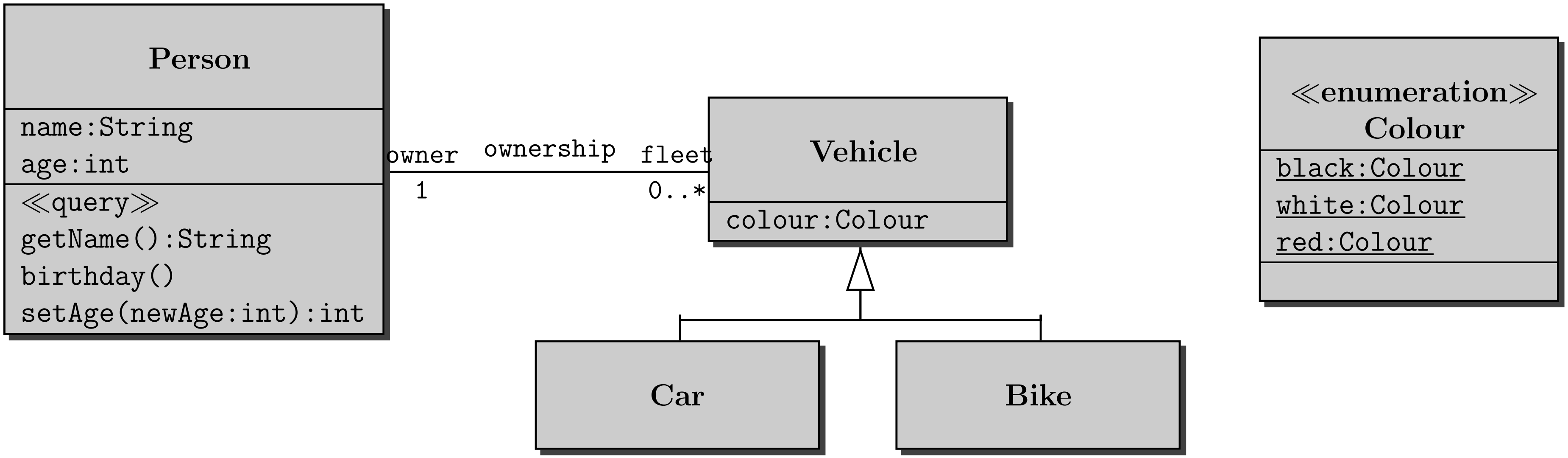
NOT

> **“Before executing an operation its precondition must hold”**

or

> **“Whenever the precondition holds, the operation is called”**

Constraints with Attributes



context **Person**

inv **: age** ≥ 18

Equivalent notational variations

context **Person**

inv **: self.age** ≥ 18

Equivalent notational variations

context **Person**

inv **: self.age** ≥ 18

context **p:Person**

inv **: p.age** ≥ 18

Equivalent notational variations

context **Person**

inv **: self.age** ≥ 18

context **p:Person**

inv **: p.age** ≥ 18

context **p:Person**

inv **minimumAge : p.age** ≥ 18

Equivalent notational variations

context **Person**
  inv **: self.age** ≥ 18

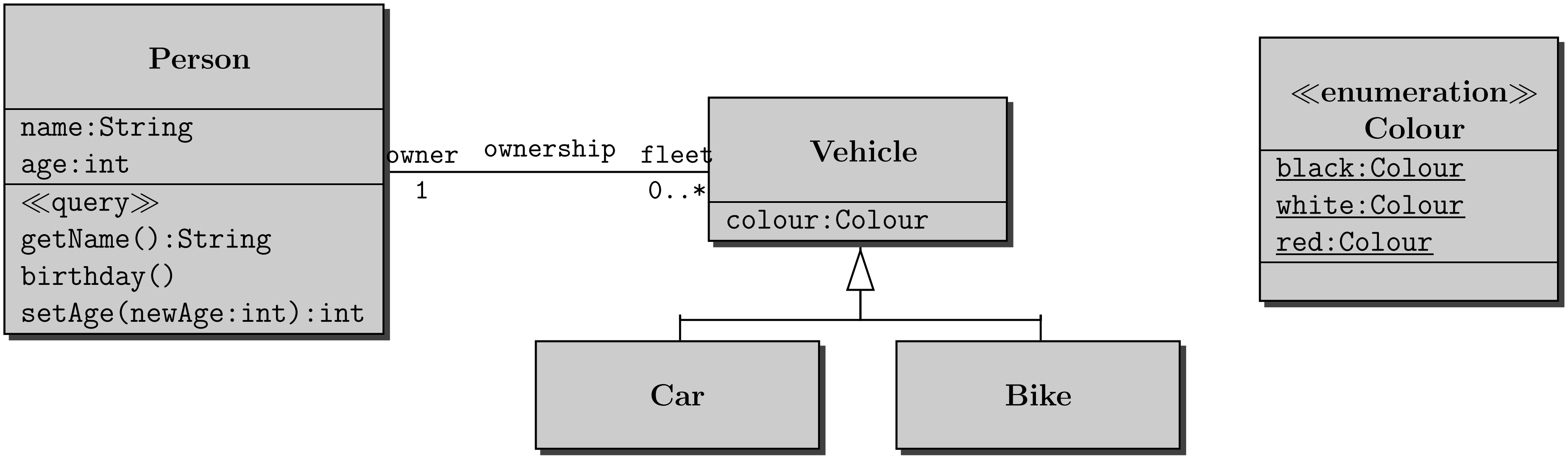
context **p:Person**
  inv **: p.age** ≥ 18

context **p:Person**
  inv **minimumAge : p.age** ≥ 18

context **Person**
  inv **minimumAge : age** ≥ 18

Beware: **variants using named instances not possible in Togther**

Operator Constraint: Contract



context **Person::setAge(newAge: int):int**

pre **: self.age ≥ 0 and newAge ≥ 0**

post **: self.age = newAge**

Which implementation satisfies the contract?

context **Person::setAge(newAge: int):int**

pre : **self.age** ≥ 0 **and newAge** ≥ 0

post : **self.age = newAge**

```
int setAge(int newAge) {
    if (age>=0 && newAge>=0) { this.age = newAge; }
    return this.age;
}
int setAge(int newAge) {
    return this.age = newAge;
}
int setAge(int newAge) {
    this.age = newAge;
    return -1;
}
```

OCL Types

UML class types

User-defined classes from context diagram of an OCL constraint

Each class of UML context diagram is legal type in OCL constraint

OCL Types

UML class types

User-defined classes from context diagram of an OCL constraint
Each class of UML context diagram is legal type in OCL constraint

Primitive types

$Integer$, $Real$, $Boolean$ **and** $String$ **(Together:** int, $real$, $boolean$**)**
int, $real$ **not in JAVA CARD, but** int, $short$, $byte$ **work in KeY**

OCL Types

UML class types

User-defined classes from context diagram of an OCL constraint
Each class of UML context diagram is legal type in OCL constraint

Primitive types

$Integer$, $Real$, $Boolean$ and $String$ (Together: int, $real$, $boolean$)
int, $real$ not in JAVA CARD, but int, $short$, $byte$ work in KeY

Enumeration types

User-defined enumeration types (not supported in Together and KeY)

OCL Types

UML class types

User-defined classes from context diagram of an OCL constraint
Each class of UML context diagram is legal type in OCL constraint

Primitive types

$Integer$, $Real$, $Boolean$ and $String$ (Together: int, $real$, $boolean$)
int, $real$ not in JAVA CARD, but int, $short$, $byte$ work in KeY

Enumeration types

User-defined enumeration types (not supported in Together and KeY)

Collection types

Set, Bag, $Sequence$

OCL Types

UML class types

User-defined classes from context diagram of an OCL constraint
Each class of UML context diagram is legal type in OCL constraint

Primitive types

$Integer$, $Real$, $Boolean$ and $String$ (Together: int, $real$, $boolean$)
int, $real$ not in JAVA CARD, but int, $short$, $byte$ work in KeY

Enumeration types

User-defined enumeration types (not supported in Together and KeY)

Collection types

Set, Bag, $Sequence$

Special types

e.g. $OclAny$, $OclType$

Type Conformance in OCL

- $Integer < Real$ **(subtype relation)**

Type Conformance in OCL

- $Integer < Real$ **(subtype relation)**
- T_1, T_2 **class types:**

  $T_1 < T_2$ **holds exactly if** T_1 **is a subclass of** T_2 **in context diagram**

Type Conformance in OCL

- $Integer < Real$ **(subtype relation)**
- T_1, T_2 **class types:**

  $T_1 < T_2$ **holds exactly if** T_1 **is a subclass of** T_2 **in context diagram**
- **For all type expressions** T**, not denoting a collection type:**
  - $Set(T) < Collection(T)$
  - $Bag(T) < Collection(T)$
  - $Sequence(T) < Collection(T)$

Type Conformance in OCL

- $Integer < Real$ **(subtype relation)**
- T_1, T_2 **class types:**

  $T_1 < T_2$ **holds exactly if** T_1 **is a subclass of** T_2 **in context diagram**
- **For all type expressions** T**, not denoting a collection type:**
  - $Set(T) < Collection(T)$
  - $Bag(T) < Collection(T)$
  - $Sequence(T) < Collection(T)$
- **If** T **is not a collection type:** $T < OCLAny$

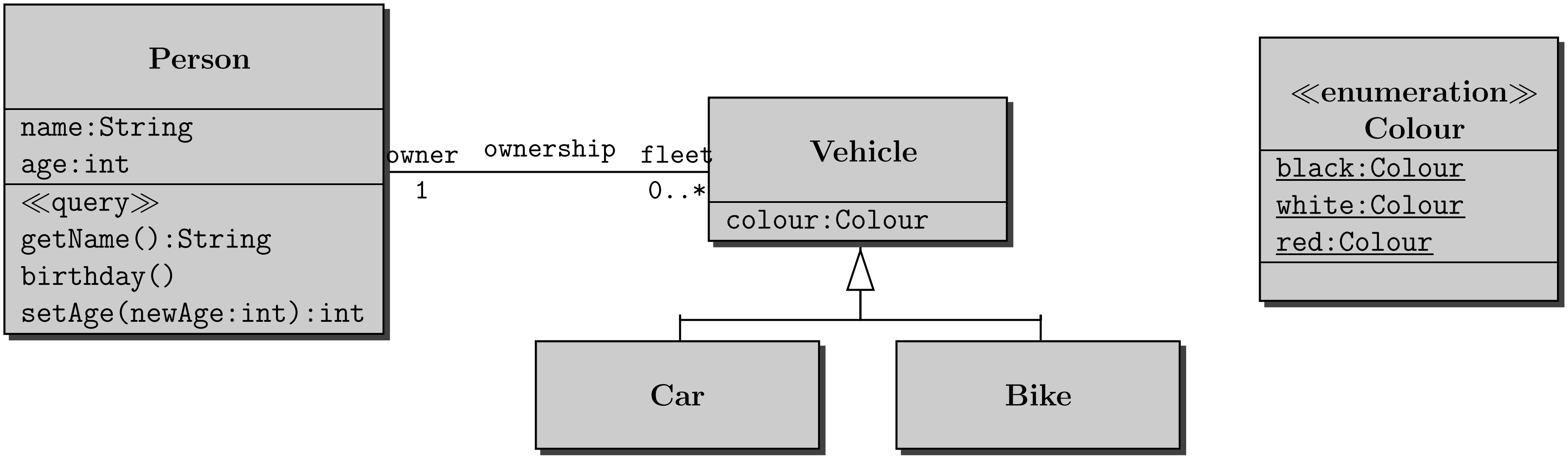
Type Conformance in OCL

- $Integer < Real$ **(subtype relation)**
- T_1, T_2 **class types:**

  $T_1 < T_2$ **holds exactly if** T_1 **is a subclass of** T_2 **in context diagram**
- **For all type expressions** T**, not denoting a collection type:**
  - $Set(T) < Collection(T)$
  - $Bag(T) < Collection(T)$
  - $Sequence(T) < Collection(T)$
- **If** T **is not a collection type:** $T < OCLAny$
- **If** $T_1 < T_2$ **and** C **is any of the type constructors** $Collection$**,** Set**,** Bag**,** $Sequence$**:**

  $C(T_1) < C(T_2)$.

Typing Examples



context **Person** **- - self.name has type** $String$

- - self.age has type $Integer$

- - self.fleet has type $Set(Vehicle)$

context **Vehicle** **- - self.colour has type** $Colour$

context **...** **- - Colour.black has type** $Colour$

Navigation: Accessing Properties

OCL Properties (functions that may occur in OCL expr)

- **Attributes from underlying UML model**
- **Association ends from underlying UML model**
- **Operations with stereotype ≪query≫ from UML model**
- **Predefined OCL properties**

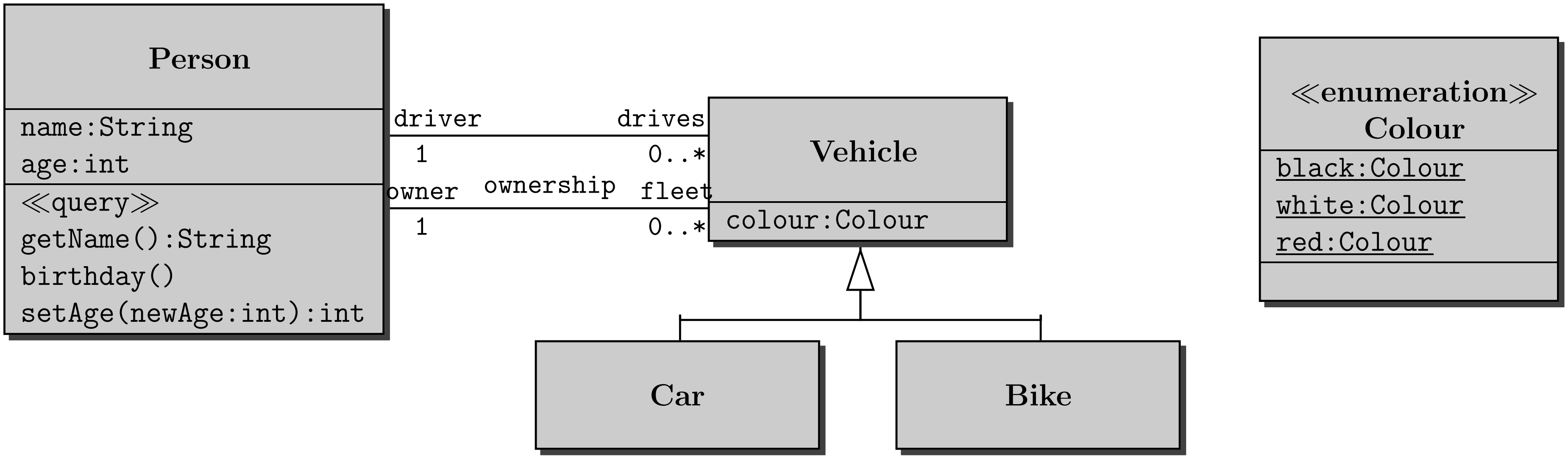
If argument has no collection type: dot notation (like JAVA)

If argument has collection type: arrow notation “–>”

Collection type has large number of predefined properties:

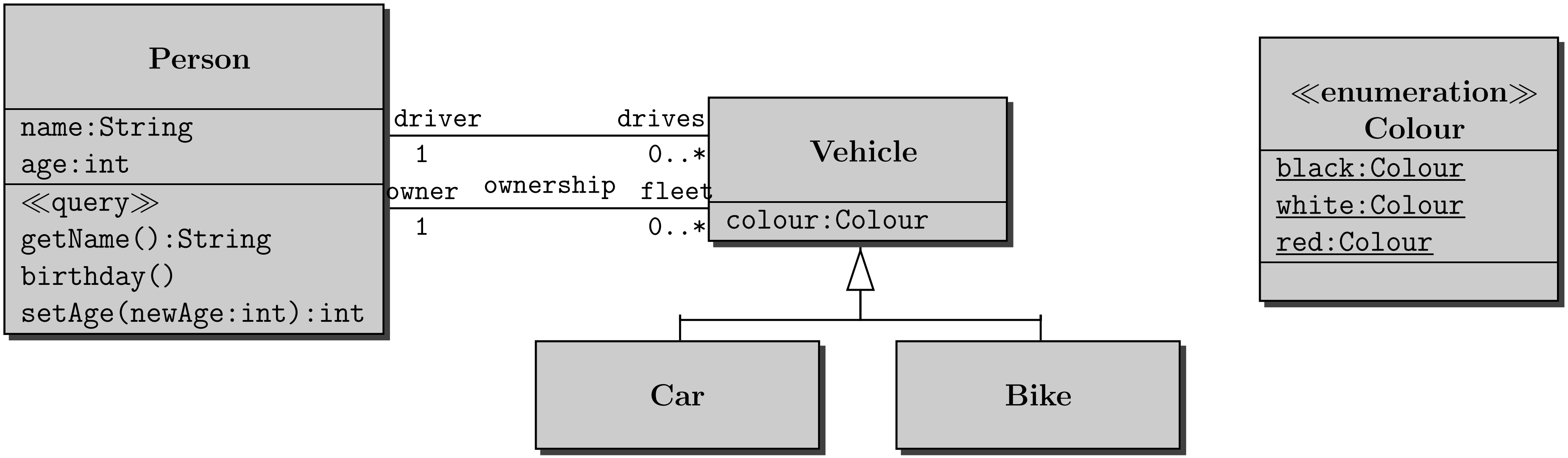
$\mathrm{includes}$, $\mathrm{intersection}$, forAll, **etc.**

User-Defined Operations within Constraints



Only «query» operations allowed to occur within OCL expressions

User-Defined Operations within Constraints



Only ≪query≫ operations allowed to occur within OCL expressions

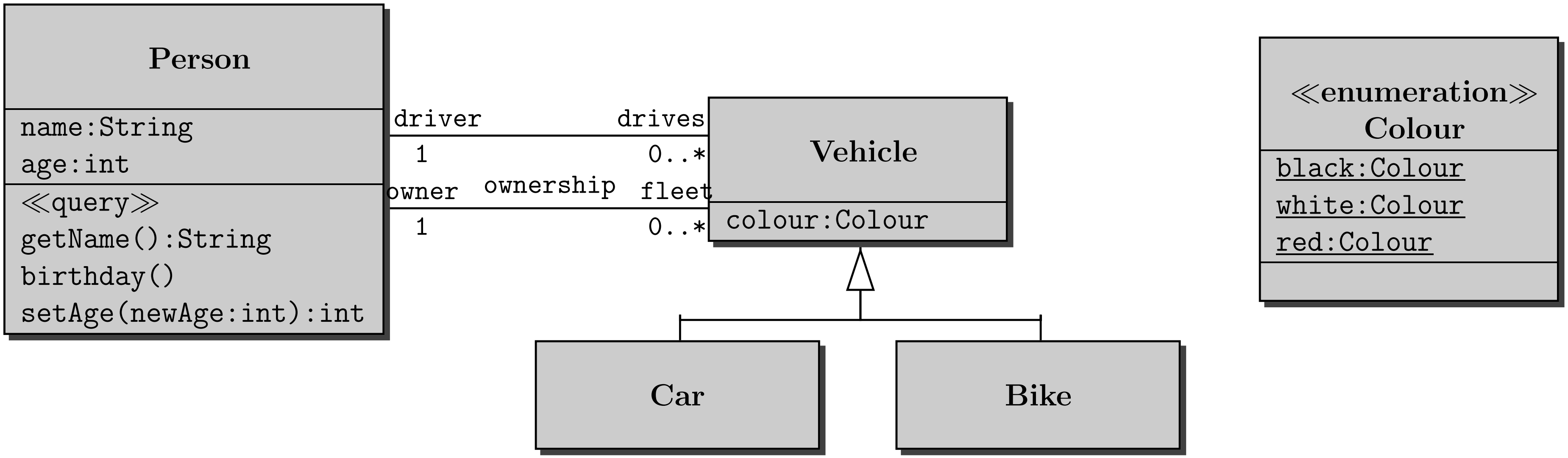
context **Person**

  inv **self.name = self.getName()**

Beware: parameterless properties with brackets, eg:

Set$\{1, 2, 3\}$ **–>** **sum()**

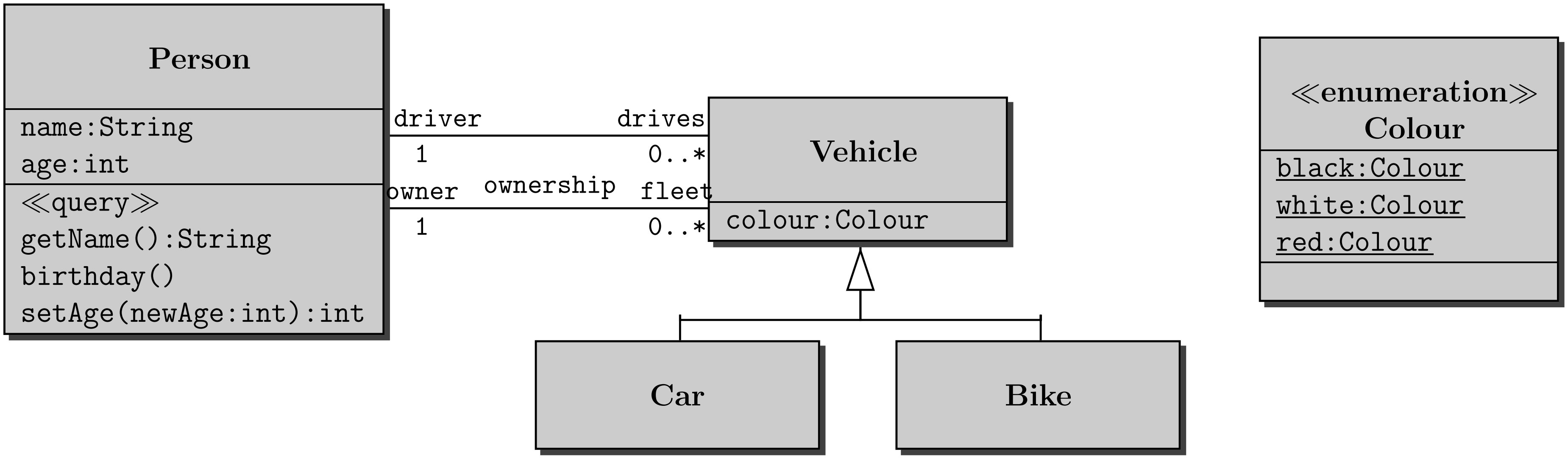
Constraints that use Associations



context **Vehicle**

inv **owner <> driver** **- - 'self' implicit!**

Constraints that use Associations



context **Vehicle**

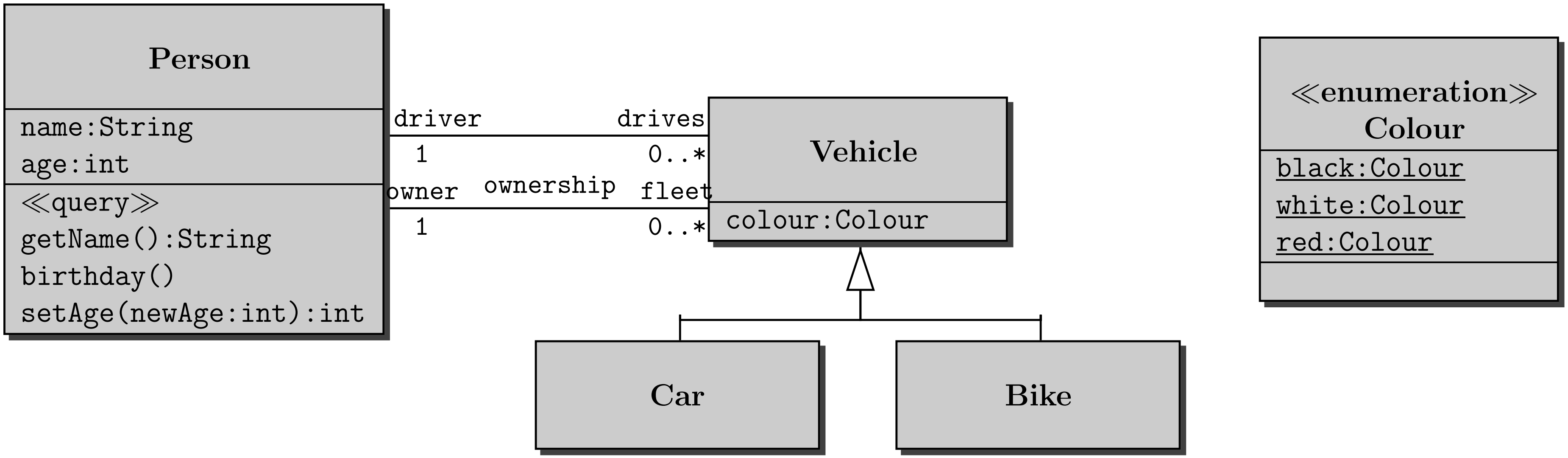
inv **owner <> driver** **- - 'self' implicit!**

context **Person**

inv **fleet –>intersection(drives) –>isEmpty()**

inv **self.fleet –>intersection(self.drives) –>isEmpty()**

Notational Variants of Collection Properties



context **Person** - - all constraints are equivalent

inv **fleet –>collect(v:Vehicle | v.colour) –>size() = 1**

inv **fleet –>collect(v | v.colour) –>size() = 1**

inv **fleet –>collect(colour) –>size() = 1**

inv **fleet.colour –>size() = 1** - - shorthand for 'collect' in Together

The type OclType

What is the type of UML model types (eg, Person**)?**

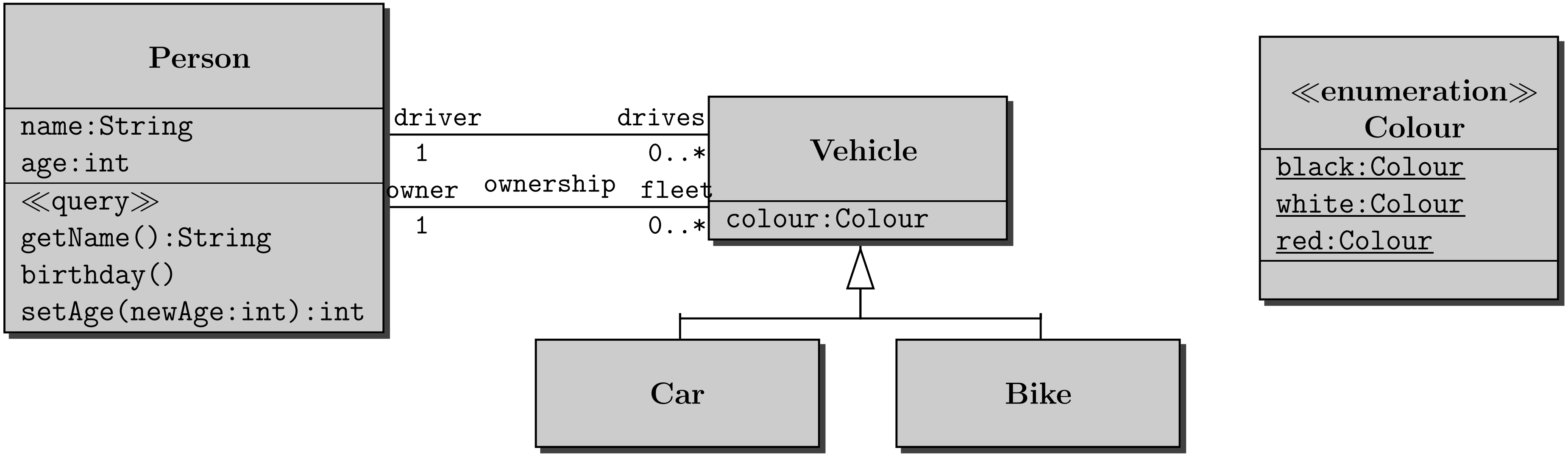
OclType

OclType **is metatype with predefined properties:**

- aType**.name() gives name string of** aType
- **Similar are attributes(), operations(), associationEnds()**
- aType**.allInstances() gives all instances of** aType **in current snapshot**

allInstances **needed to express properties relating to all currently existing objects**

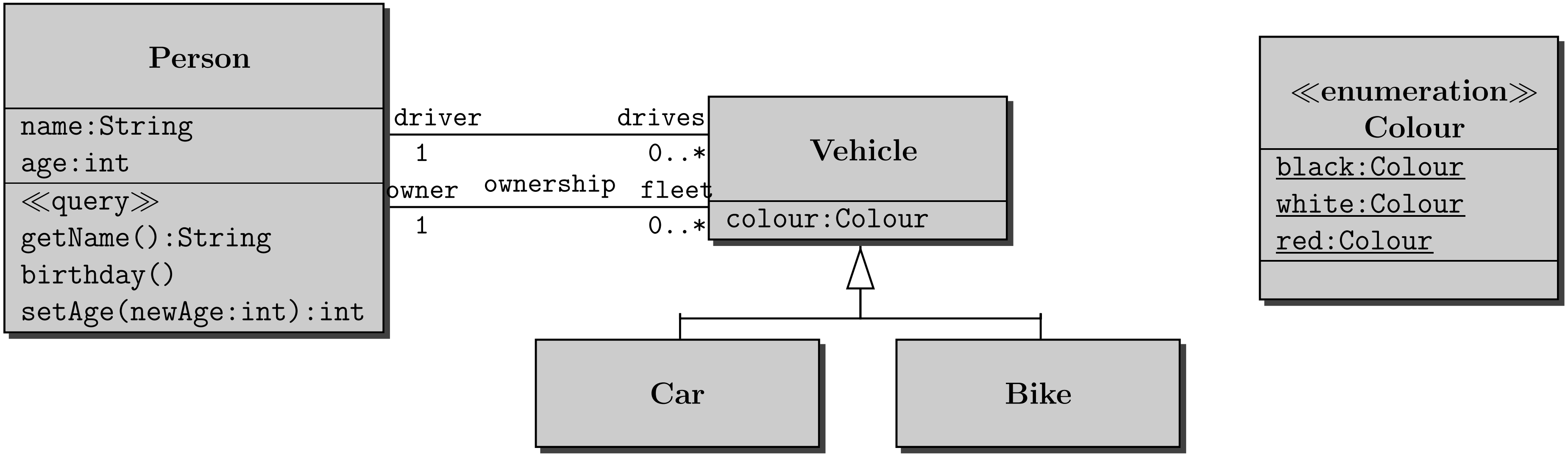
Using allInstances



context **Person**

inv **Person.allInstances –> forAll(p | p.age $\geq$ 0)**

Using allInstances



context **Person**

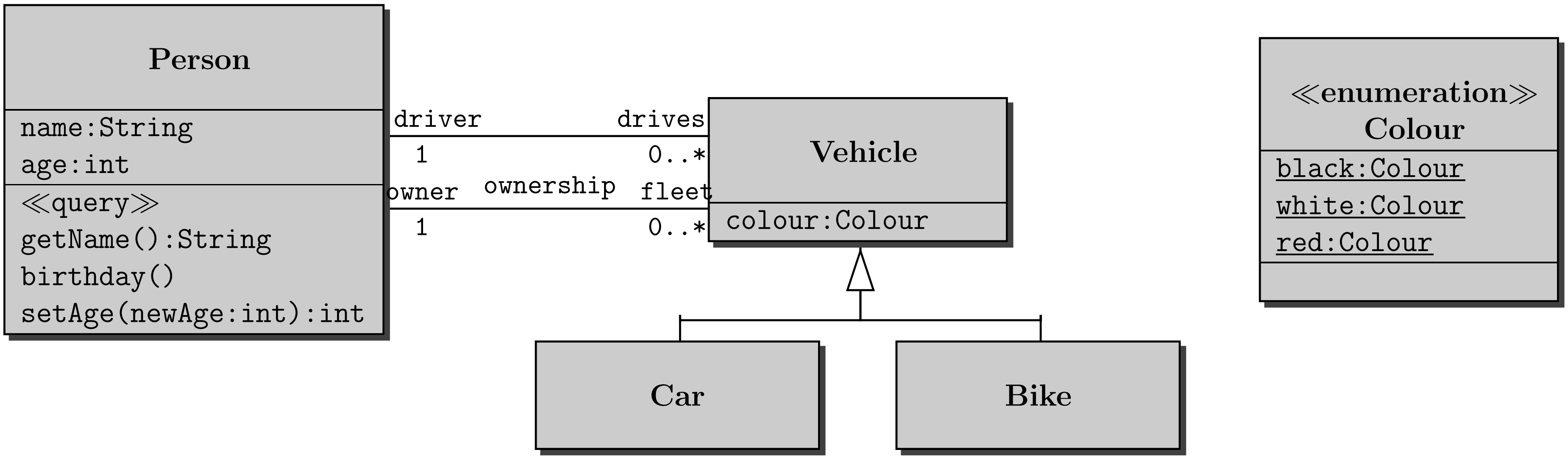
inv **Person.allInstances –> forAll(p | p.age ≥ 0)**

Constraint is independent of model context — equivalent:

context **Vehicle**

inv **Person.allInstances –> forAll(p | p.age ≥ 0)**

Using allInstances



context **Person**

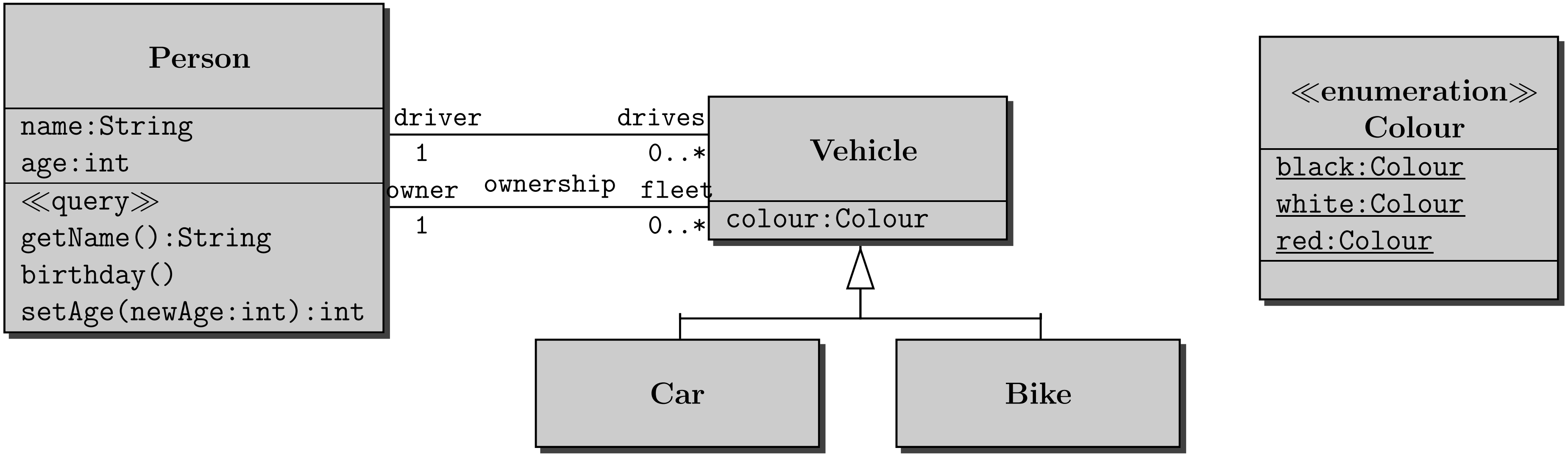
inv **Person.allInstances –> forAll(p | p.age $\geq$ 0)**

Context declaration of invariant has implicit allInstances/forAll:

context **Person** **- - equivalent to constraint above**

inv **self.age $\geq$ 0**

Avoiding allInstances



context **Person**

inv **Person.allInstances –>**

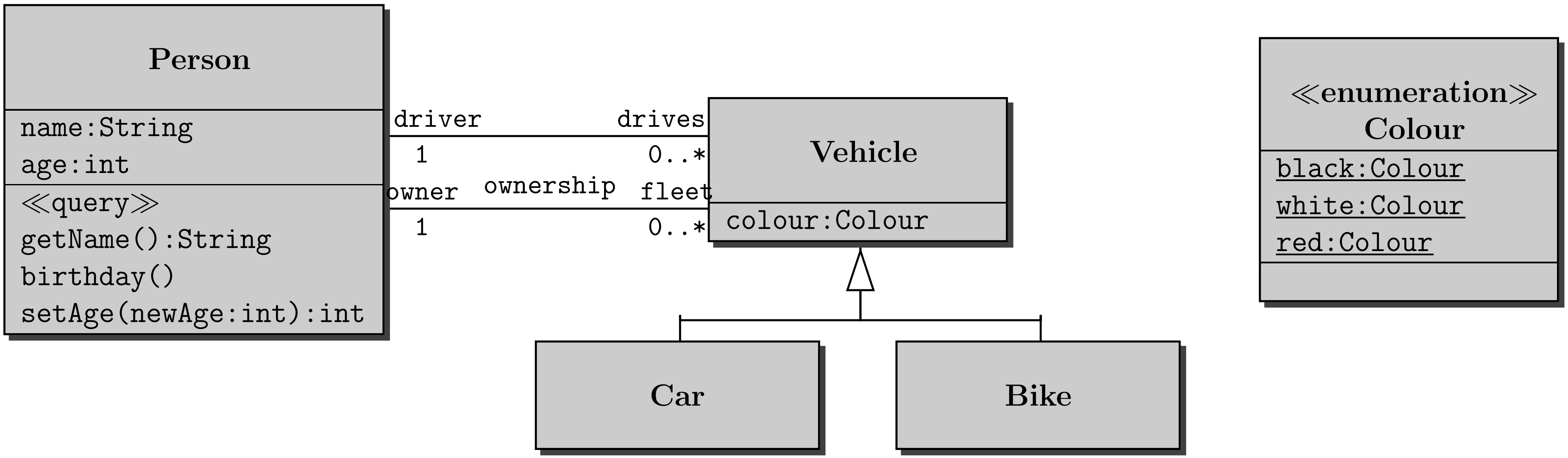
forAll(p1, p2 | p1.name = p2.name implies p1 = p2)

allInstances

. . . tends to make constraint difficult to read

. . . can give rise to unnecessarily difficult verification task

Avoiding allInstances



context **Person**

inv **Person.allInstances –>**

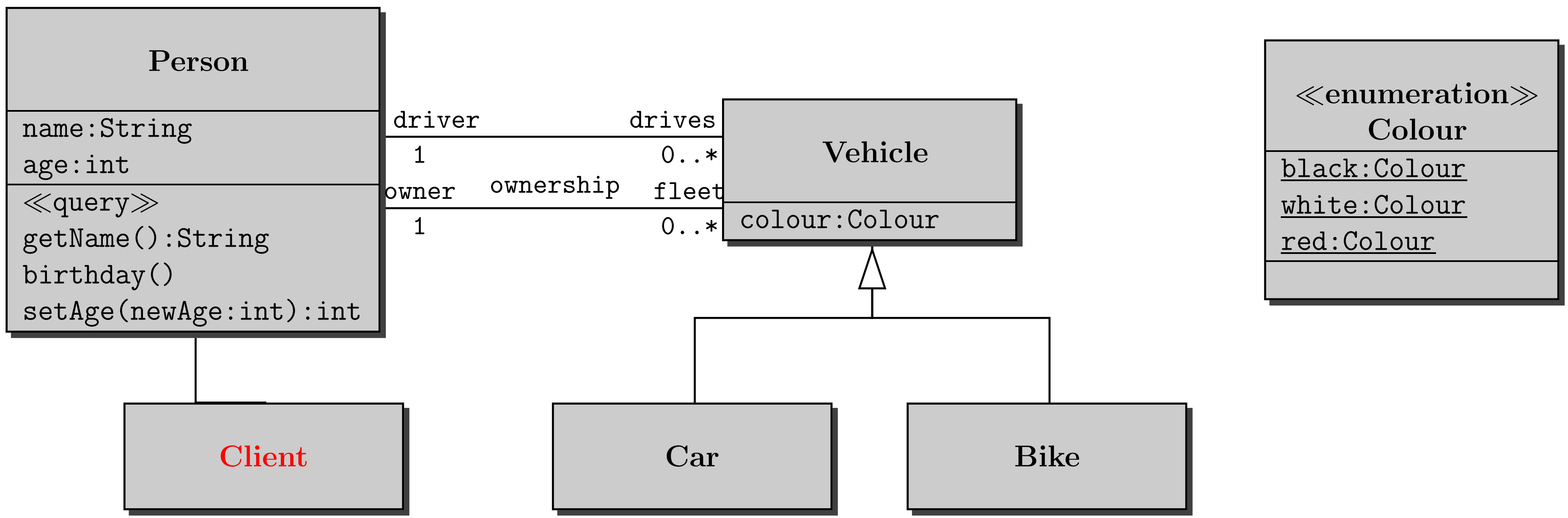
forAll(p1, p2 | p1.name = p2.name implies p1 = p2)

Can be equivalently replaced with: (not in Together!)

context **p1,p2:Person**

inv **p1.name = p2.name implies p1 = p2**

Avoiding allInstances



context **Person**

inv **Person.allInstances –>**

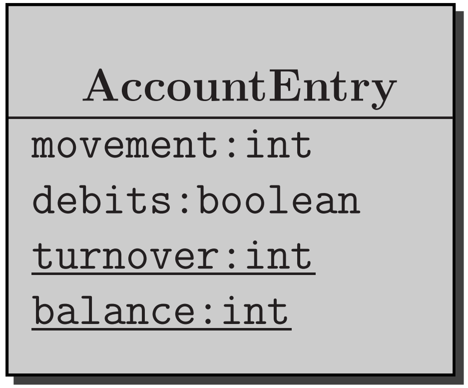
forAll(p1, p2 | p1.name = p2.name implies p1 = p2)

Often, collection of objects available via suitable association:

context **Client**

inv **: person –> forAll(p1, p2 | p1.name = p2.name implies p1 = p2)**

The iterate Property



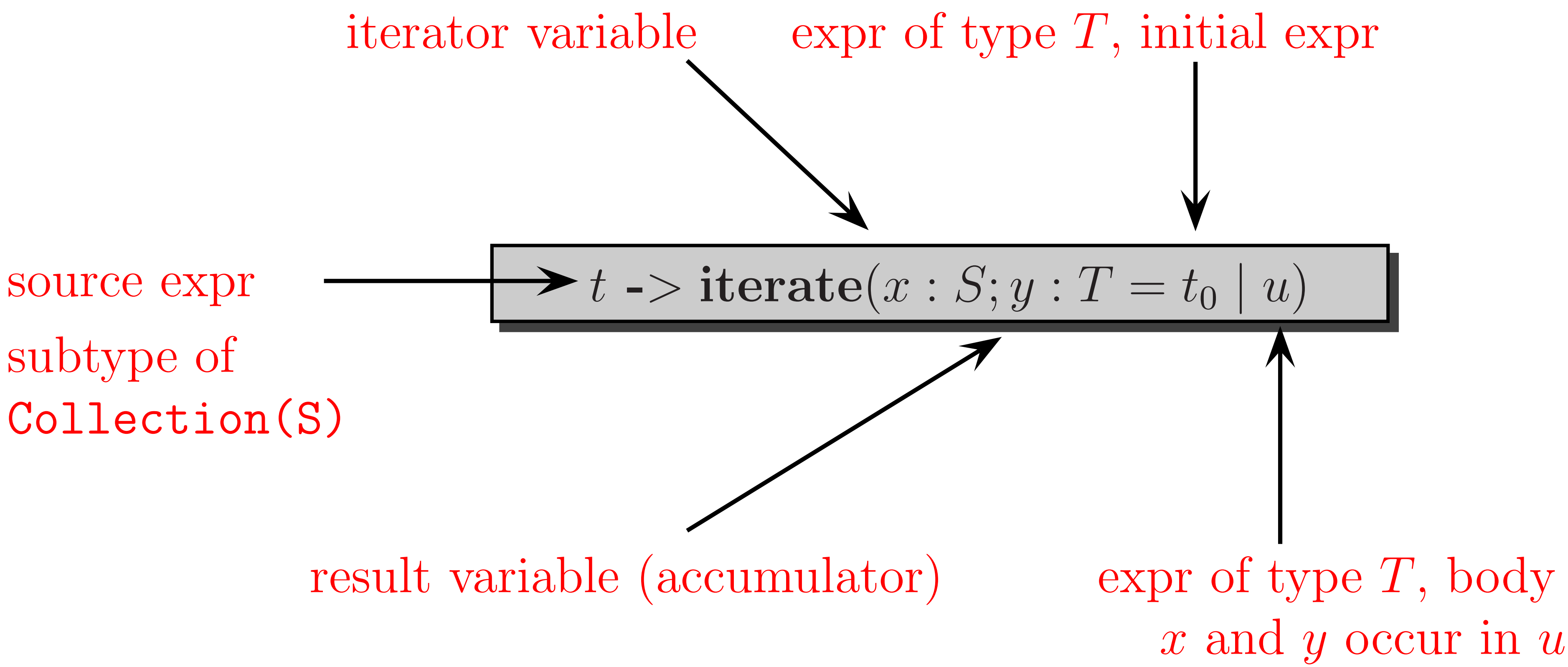
context **AccountEntry**

inv **AccountEntry.allInstances –>**

iterate(a:AccountEntry ; m:Integer=0 | m+a.movement) =

AccountEntry.turnover

Syntax of the *iterate* Property



Java Pseudocode of *iterate*

t –>iterate(x:S; y:T=t_0 | u)

```
S x;
T y = t0;
for (Enumeration e = t.elements(); e.hasMoreElements() ) {
  x = e.nextElement();
  y = u(x,y);
}
```

Type of x and y can be inferred from t and u

OCL's iterate is also similar to the accumulate function of the C++ STL

Quantifiers

t –>iterate(x:S; y:Boolean=true | y and a(x))

. . . **where** $a(x)$ **is an expression of type** $\mathrm{Boolean}$ **(with occurrence of** x**)**

Quantifiers

$$\text{t ->iterate(x:S; y:Boolean=true} \mid \text{y and a(x))}$$

. . . where $a(x)$ is an expression of type $\mathrm{Boolean}$ (with occurrence of x)

Can be equivalently expressed by

$$\text{t ->forAll(x} \mid \text{a(x))}$$

Quantifiers

t –>iterate(x:S; y:Boolean=true | y and a(x))

. . . where $\mathrm{a(x)}$ **is an expression of type** $\mathrm{Boolean}$ **(with occurrence of** x**)**

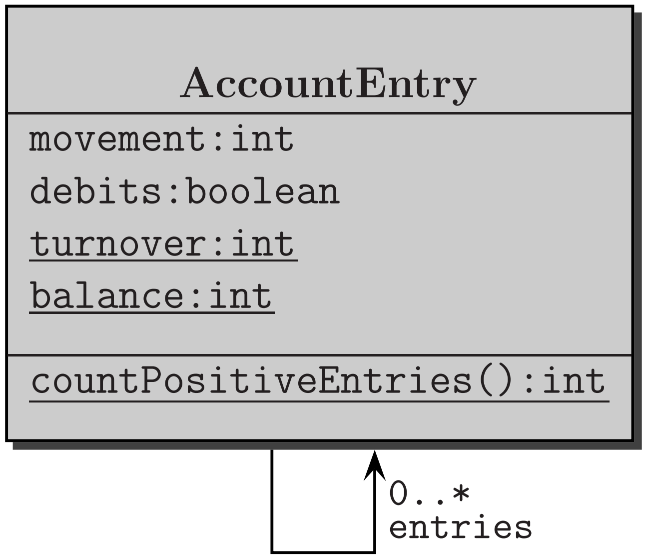
Can be equivalently expressed by

t –>forAll(x | a(x))

Similar:

t –>exists(x | a)

Selecting Elements

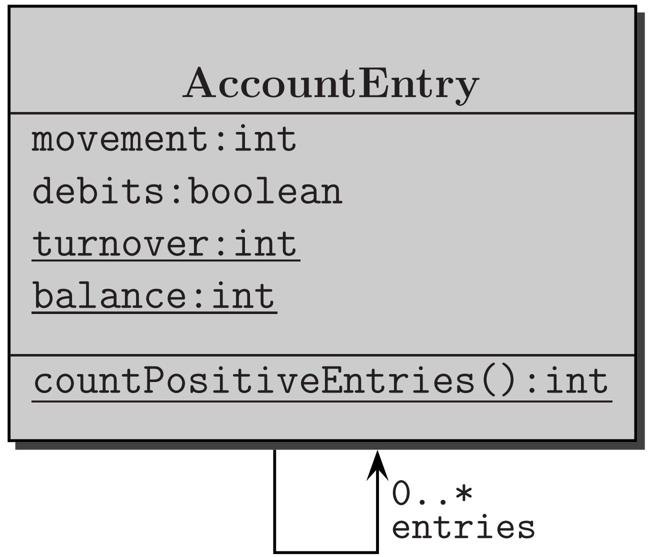


context **AccountEntry::countPositiveEntries():int**

pre **: true**

post **: result = AccountEntry.allInstances –>**
select(e | not e.debits) –> size()

Selecting Elements



context **AccountEntry::countPositiveEntries():int**

pre : **true**
post : **result = AccountEntry.allInstances –> select(e | not e.debits) –> size()**

Alternative notation using self-association:

post : **result = entries –> select(not debits) –> size()**

Reducing *select* to *iterate*

Like all other collection properties *select* definable with *iterate*

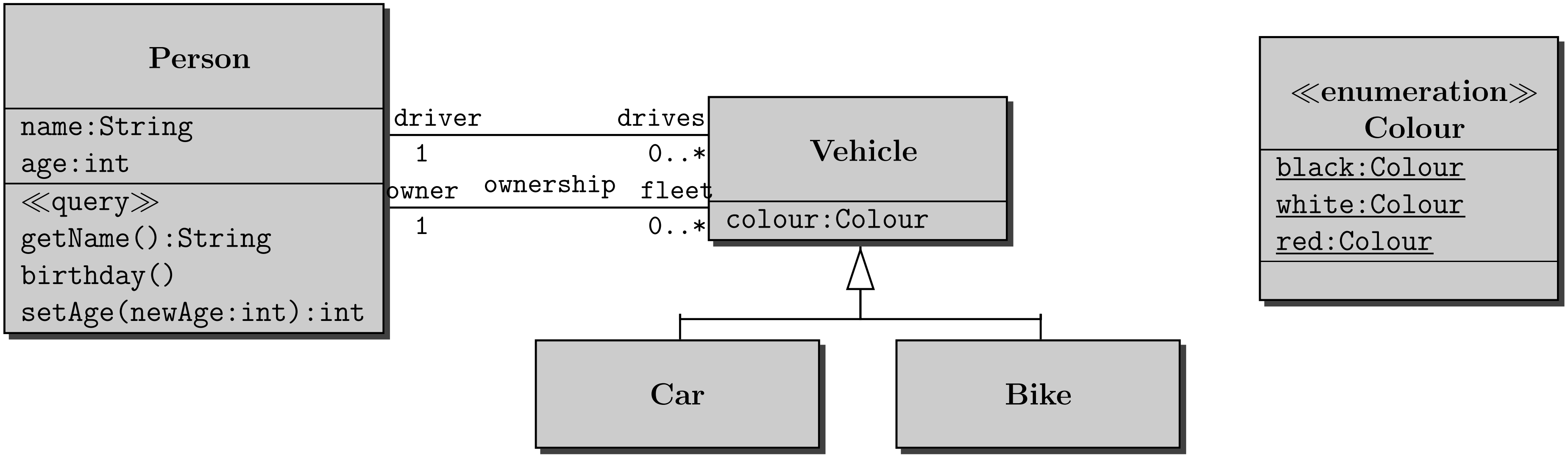
s –> select(x:T | e) =

iterate(x:T; acc: Set(T) = Set{} |

if e then acc –> including(x) else acc)

- s **is of type** $Set(T)$
- e **is an OCL expression of type** Boolean
- including **in turn is definable with** iterate
- **all built-in collection properties definable with** iterate **and** includes

Referring to Previous Values



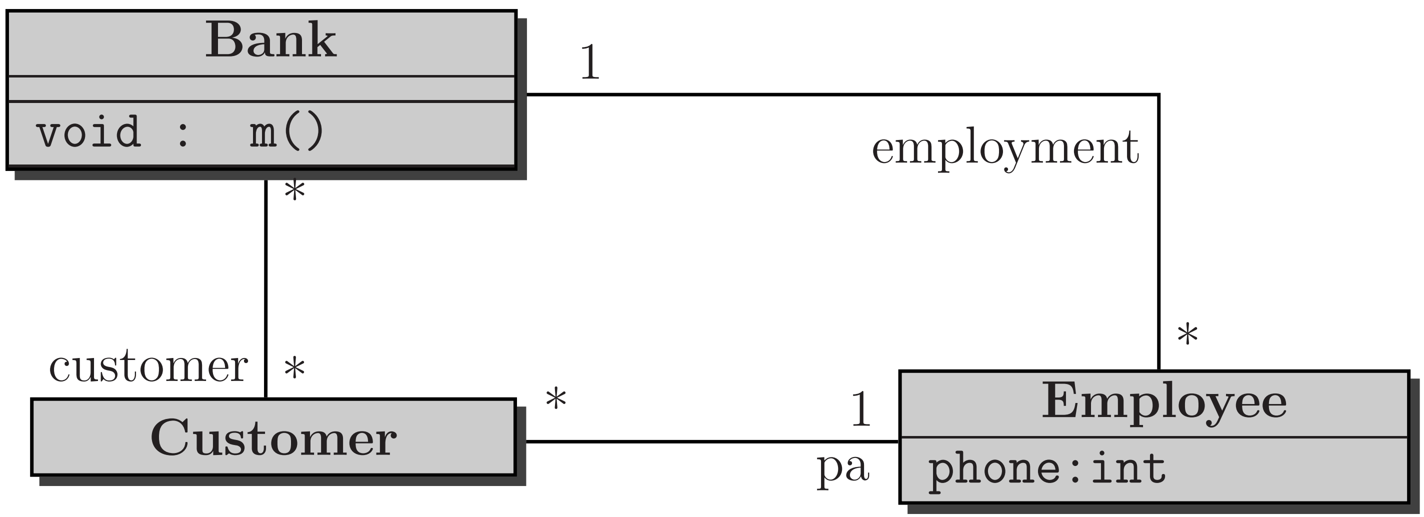
context **Person::birthday()**

pre **age $\geq$ 0**

post **age = age@pre + 1**

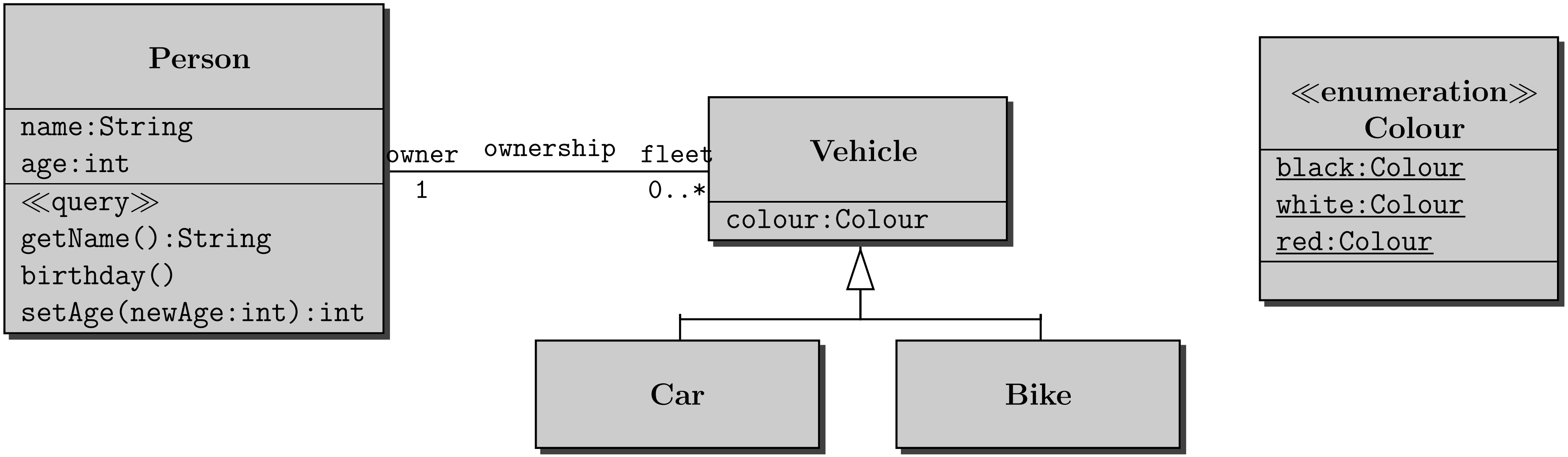
User-defined properties qualified with @pre refer to value in prestate

Multiple Occurrences of @pre



$aCustomer.pa.phone$	**new phone number of current p.a.**
$aCustomer.pa@pre.phone$	**new phone number of previous p.a.**
$aCustomer.pa.phone@pre$	**old phone number of current p.a.**
$aCustomer.pa@pre.phone@pre$	**old phone number of previous p.a.**

A Method Does More Than It Should



context **Person::setAge(newAge: int):int**

pre : **self.age** ≥ 0 **and newAge** ≥ 0

post : **self.age = newAge**

```
int setAge(int newAge) { // correct implementation?!
    name = "Jabberwocky";
    return this.age = newAge;
}
```

The Frame Problem

How to express that nothing else is changed than what is specified?

Known in AI as the Frame Problem

The Frame Problem

How to express that nothing else is changed than what is specified?

Known in AI as the Frame Problem

First Solution

context **Person::setAge(newAge: int):int**

pre : **self.age** ≥ 0 **and newAge** ≥ 0

post : **self.age = newAge and name = name@pre**

Done for all attributes visible for context class: very tedious!

The Frame Problem

How to express that nothing else is changed than what is specified?

Known in AI as the Frame Problem

Second Solution

context **Person::setAge(newAge: int):int**

pre : **self.age** ≥ 0 **and newAge** ≥ 0

post : **self.age = newAge**

modifies: **self.age**

The OCL to FOL compiler creates an efficient representation

KeY extension to OCL, not in the standard

Snapshots and OCL Constraints

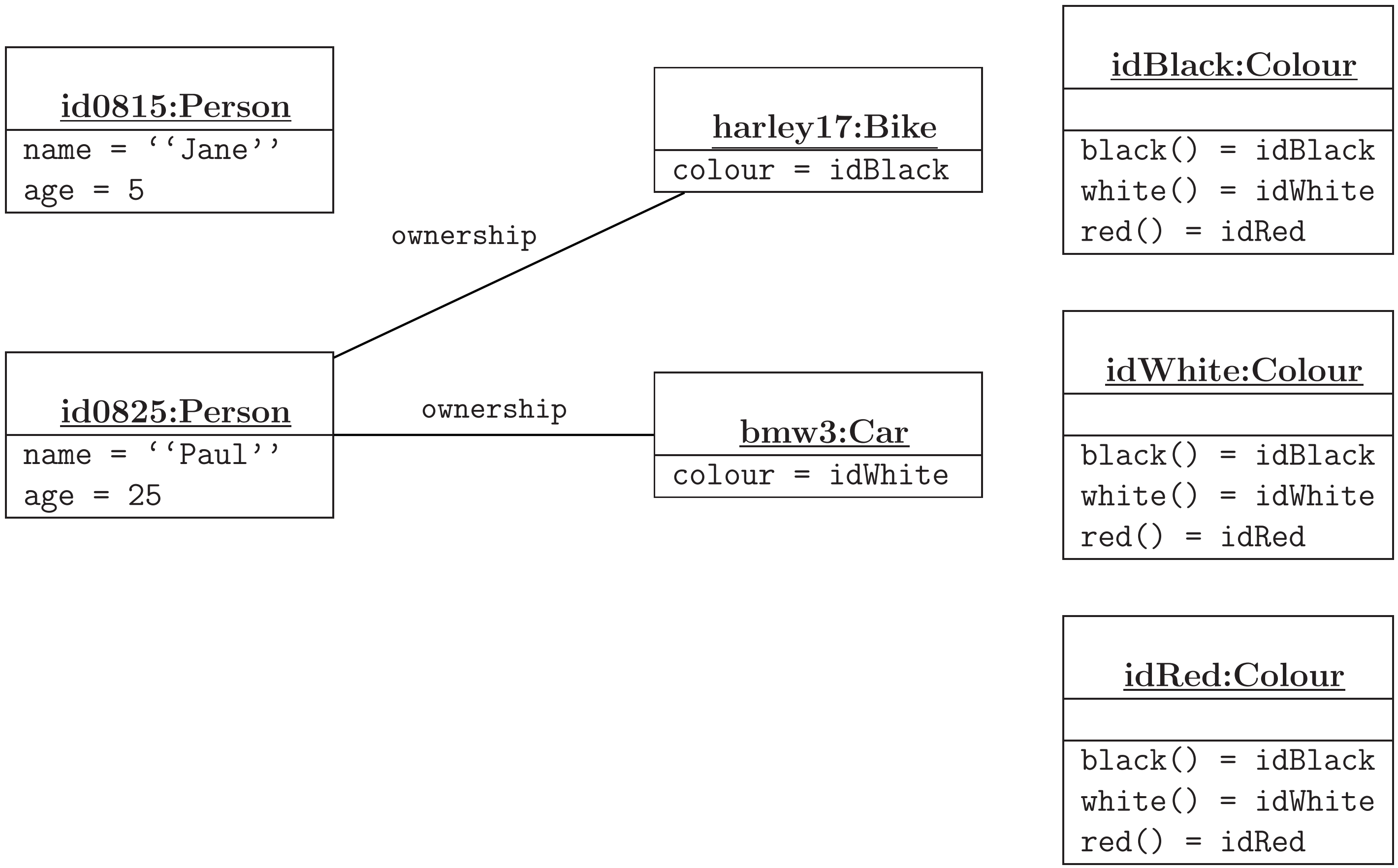
- **OCL constraints evaluated relative to a snapshot I**

  (Recall that snapshot determines an object diagram)
- **OCL expressions have type $\mathrm{Boolean} \Rightarrow$ they are true or false wrt I**
- **OCL constraints restrict legal snapshots of UML diagram**

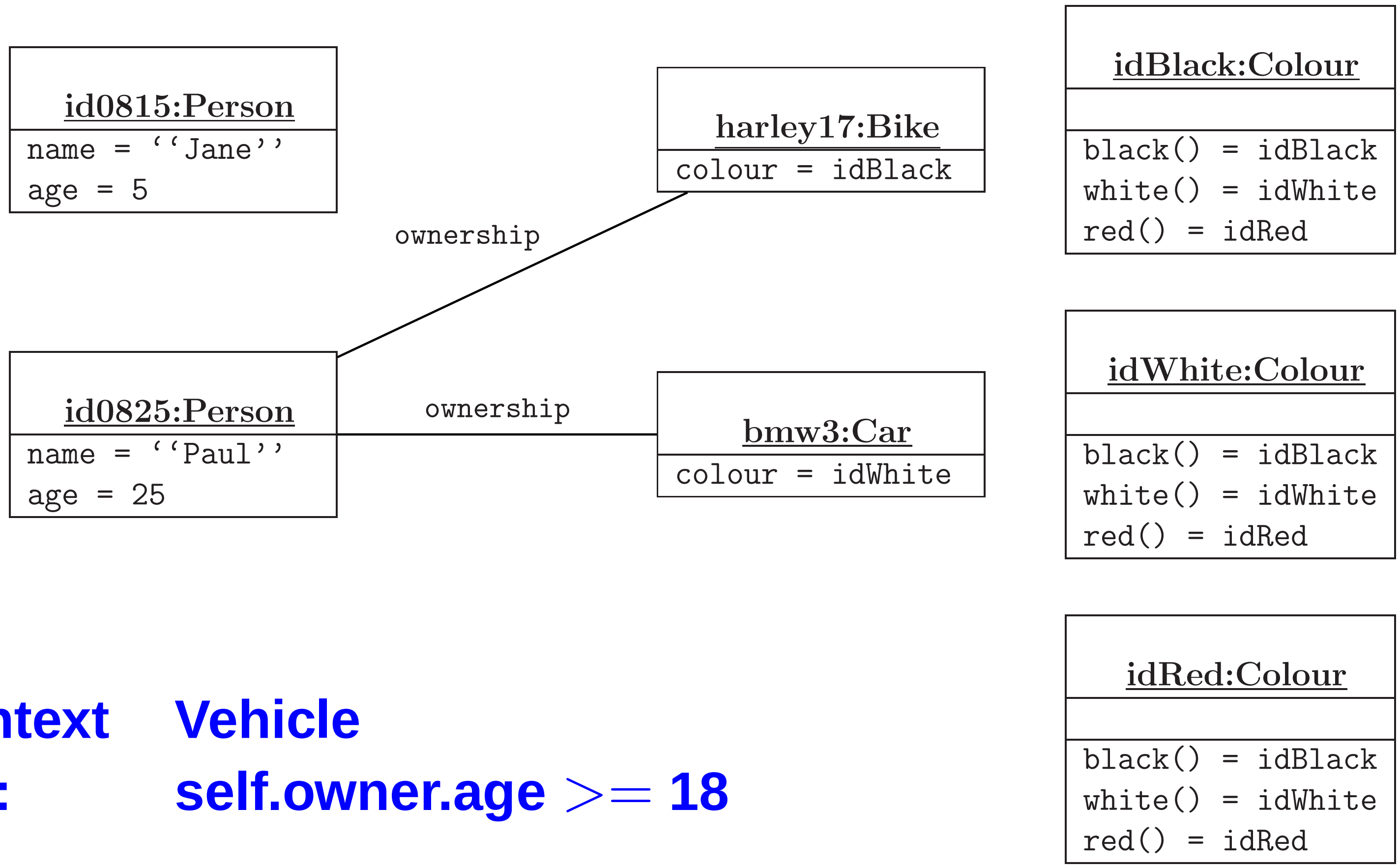
  Possibility to express intended semantics of diagram
- **OCL expressions can be evaluated and checked wrt given snapshot**
- **Don’t give formal semantics of OCL in terms of snapshots**

  Tell later how UML/OCL is translated into FOL/DL

Object Diagrams and OCL Constraints

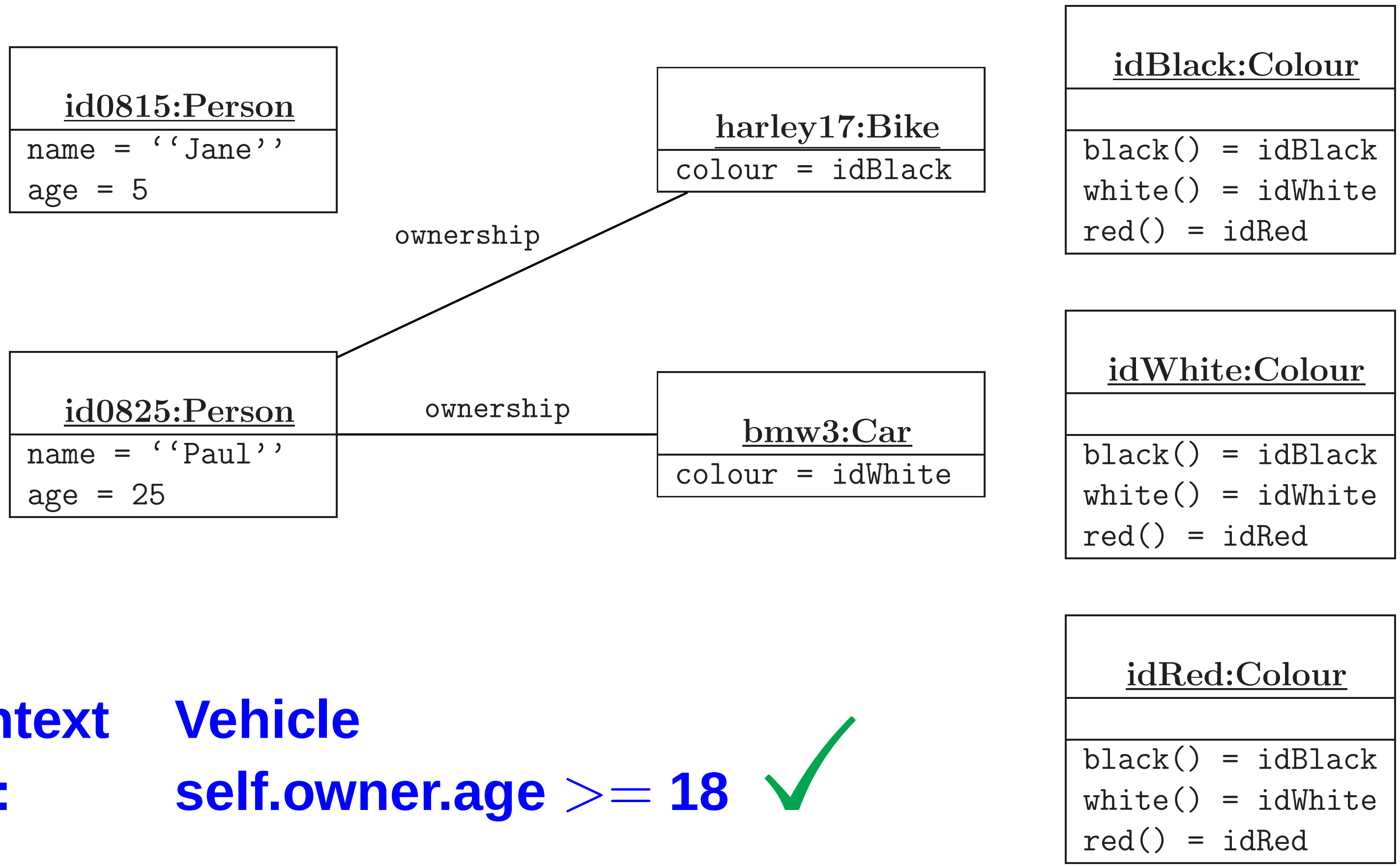


Object Diagrams and OCL Constraints



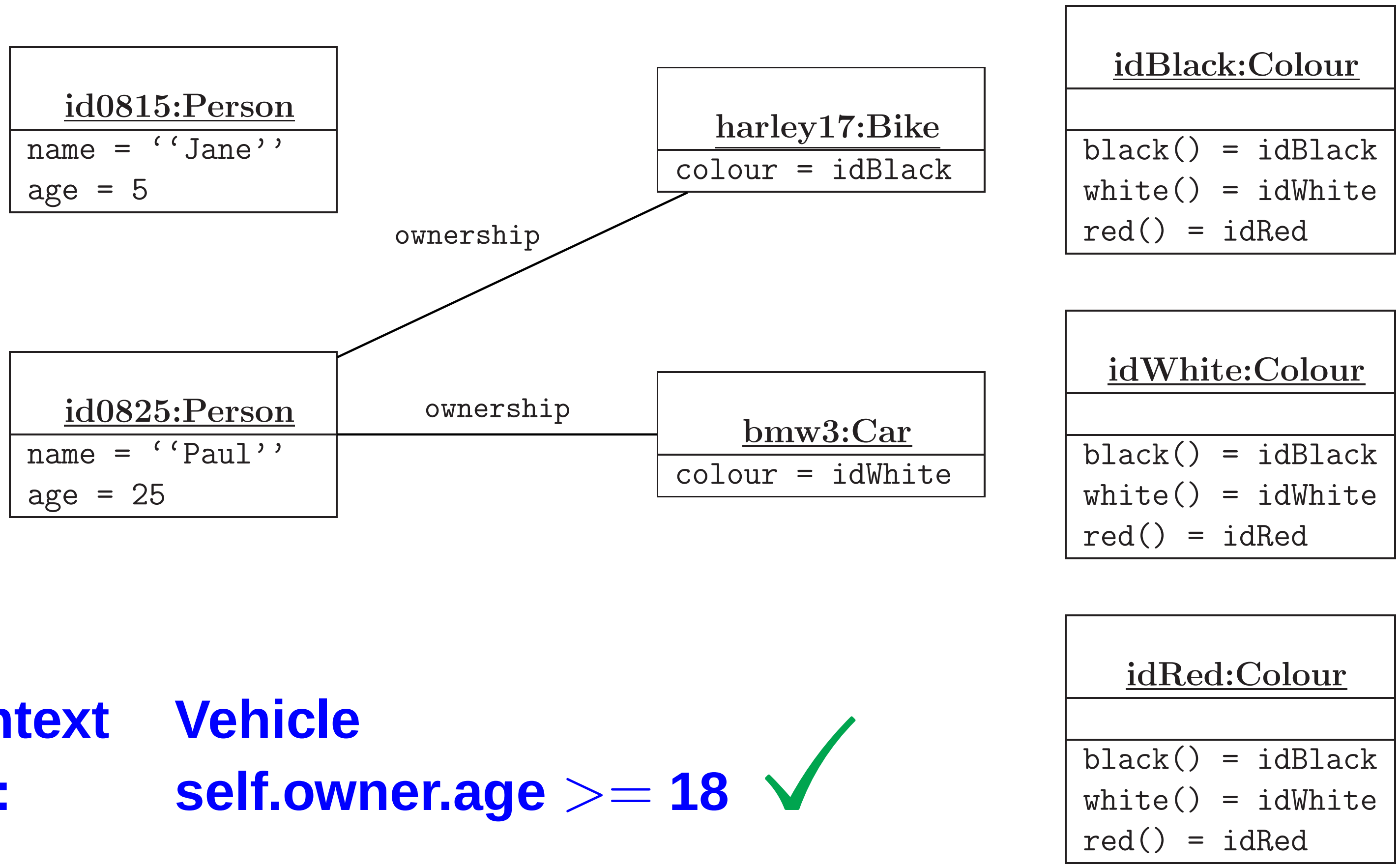
context **Vehicle**
inv: **self.owner.age $>=$ 18**

Object Diagrams and OCL Constraints



context Vehicle
inv: self.owner.age $>=$ 18 ✓

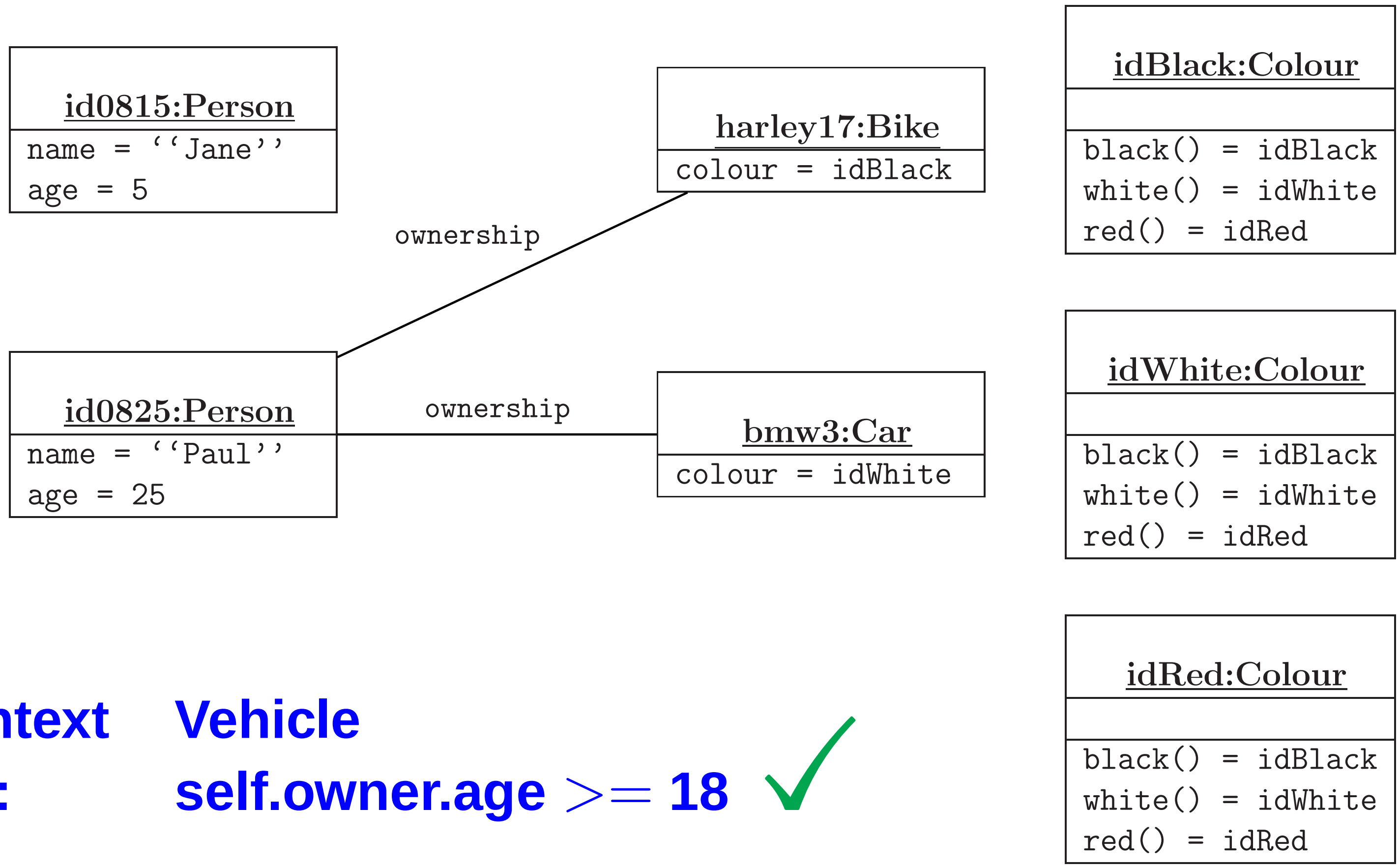
Object Diagrams and OCL Constraints



context **Vehicle**
inv: **self.owner.age $>=$ 18** ✓

context **Person**
inv: **fleet–$>$forAll(colour = Colour.black)**

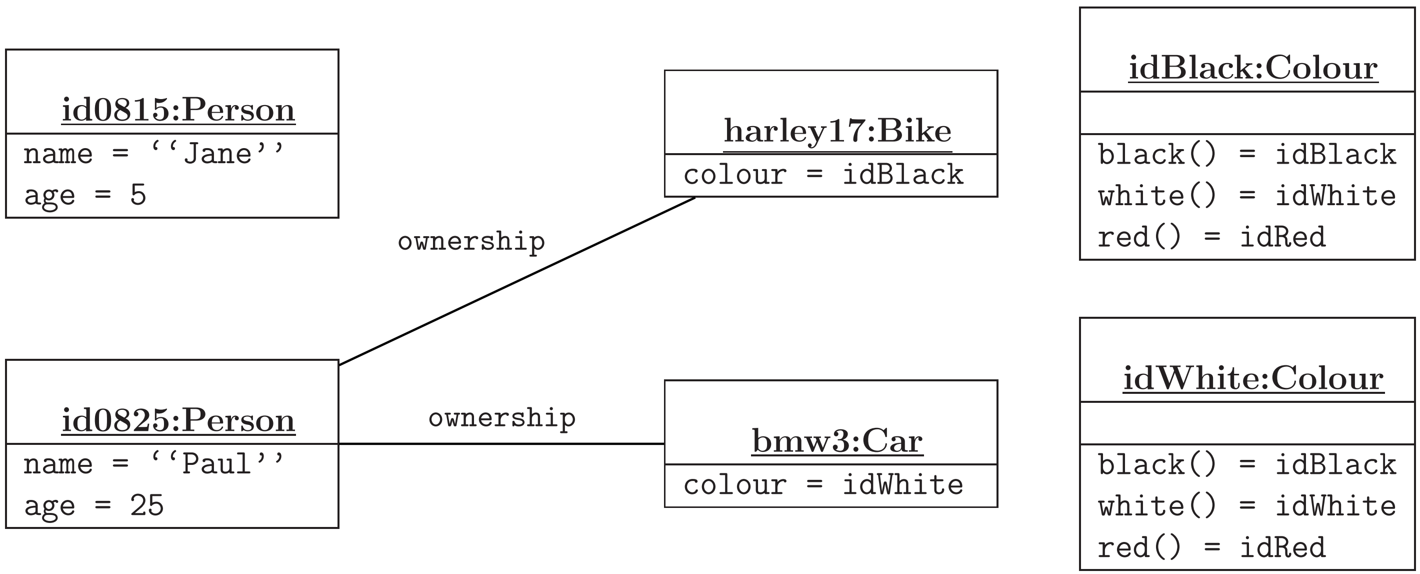
Object Diagrams and OCL Constraints

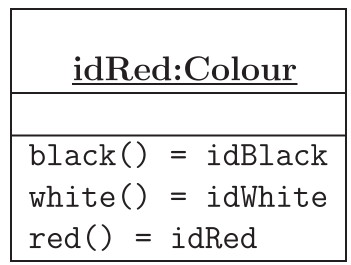


context **Vehicle**
inv: **self.owner.age >= 18** ✓

context **Person**
inv: **fleet–>forAll(colour = Colour.black)** ☒

Object Diagrams and OCL Constraints





context **Vehicle**

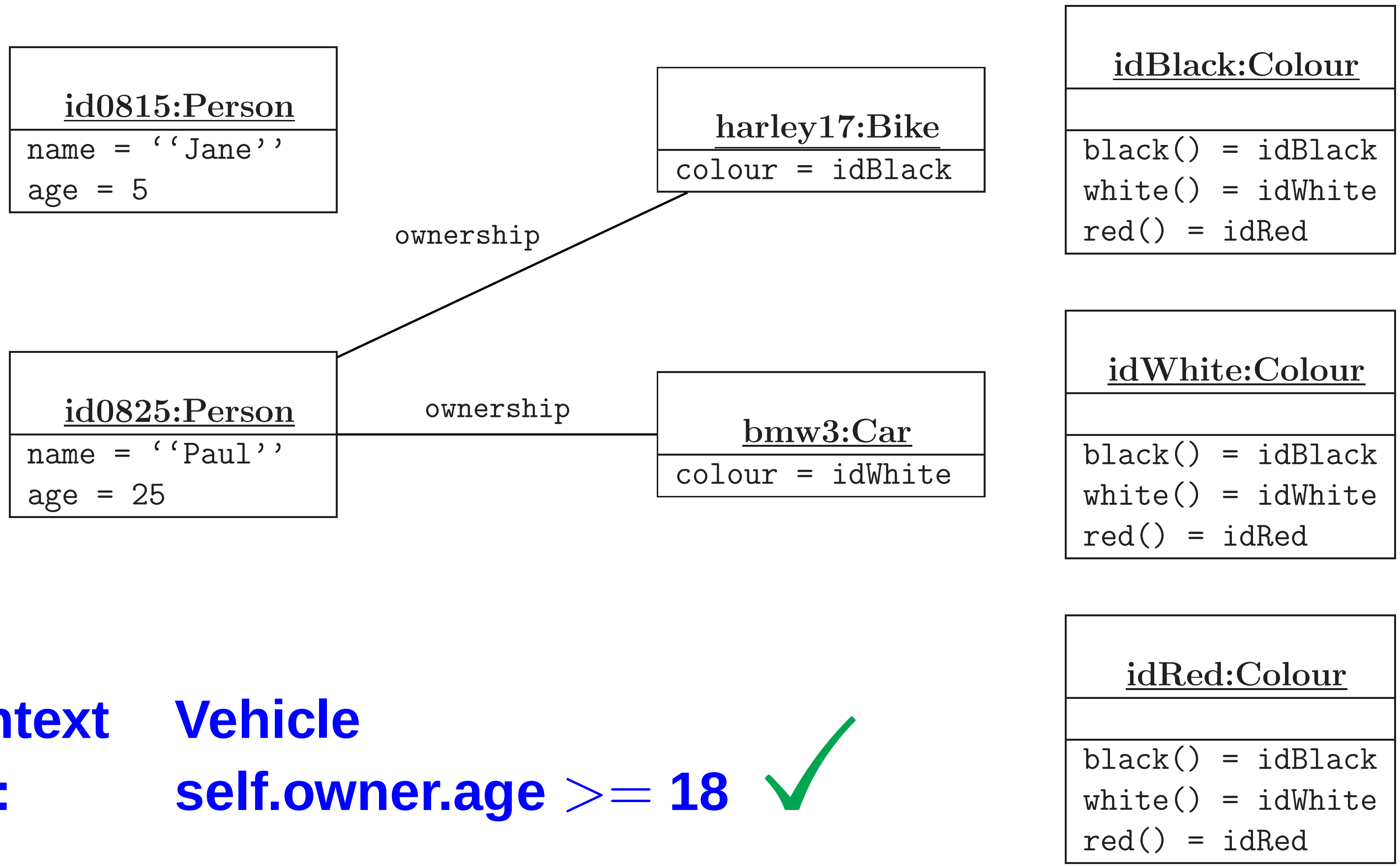
inv: **self.owner.age >= 18** ✓

context **Person**

inv: **fleet–>forAll(colour = Colour.black)** ☒

inv: **fleet–>select(colour = Colour.black) –>size() <= 3**

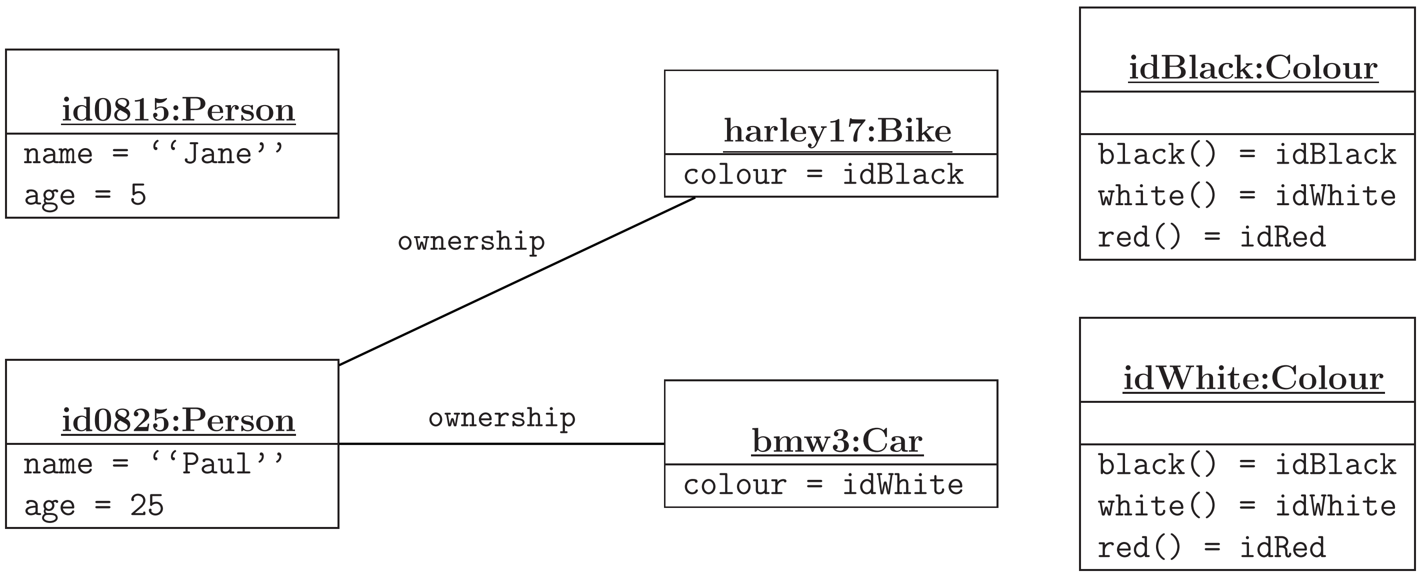
Object Diagrams and OCL Constraints

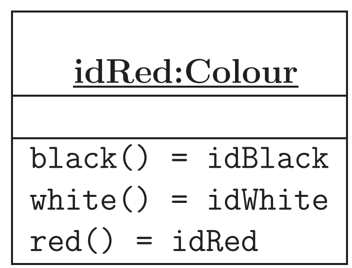


context **Vehicle**
inv: **self.owner.age >= 18** ✓

context **Person**
inv: **fleet–>forAll(colour = Colour.black)** ☒
inv: **fleet–>select(colour = Colour.black) –>size() <= 3** ✓

Object Diagrams and OCL Constraints





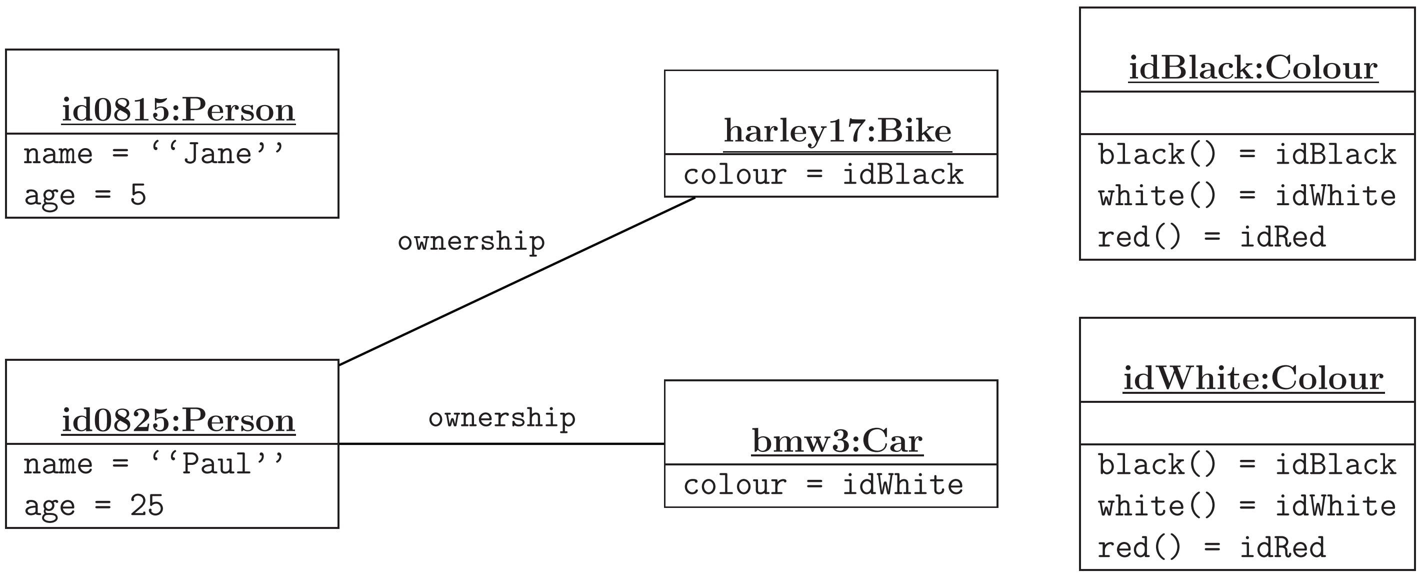
context **Vehicle**
inv: **self.owner.age >= 18** ✓

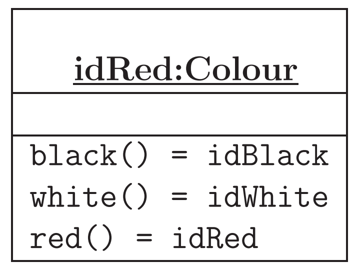
context **Person**
inv: **fleet–>forAll(colour = Colour.black)** ☒

inv: **fleet–>select(colour = Colour.black) –>size() <= 3** ✓

inv: **Car.allInstances –>exists(colour = Colour.red)**

Object Diagrams and OCL Constraints





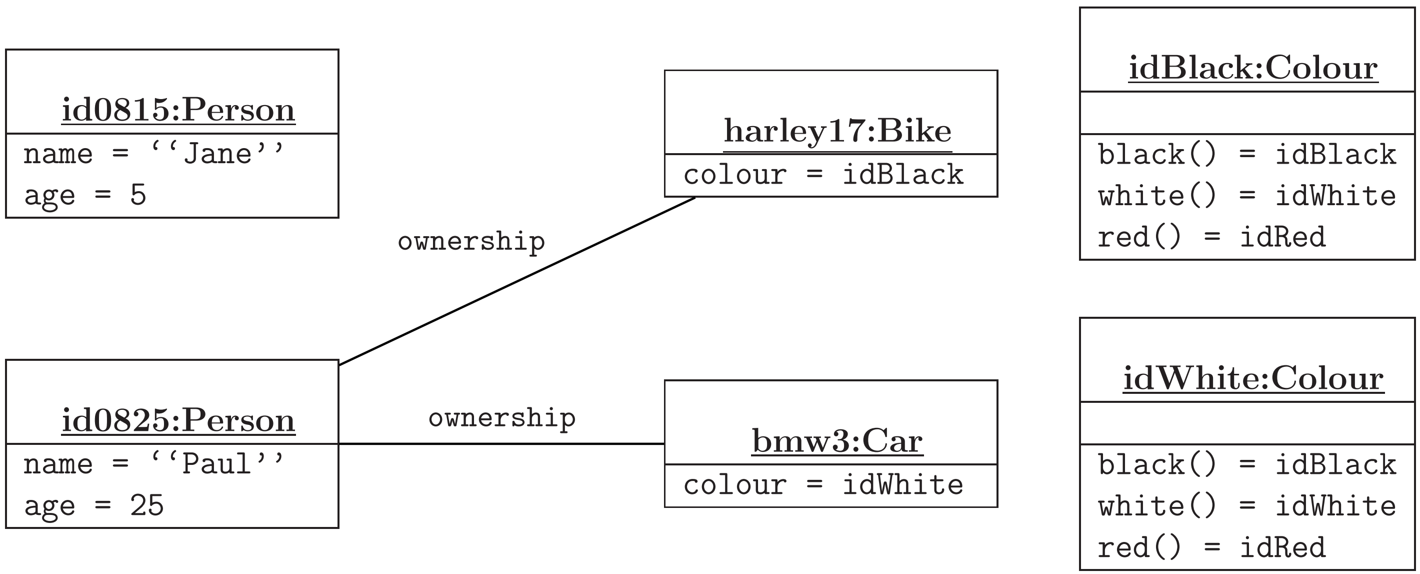
context **Vehicle**
inv: **self.owner.age $>=$ 18** ✓

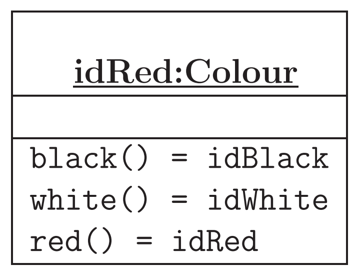
context **Person**
inv: **fleet–$>$forAll(colour = Colour.black)** ☒

inv: **fleet–$>$select(colour = Colour.black) –$>$size() $<$= 3** ✓

inv: **Car.allInstances –$>$exists(colour = Colour.red)** ☒

Object Diagrams and OCL Constraints





context **Person::getName()**
post: **result = name** **?**

context **Person::birthDay()**
pre: **age $\geq$ 0**
post: **age = age@pre + 1** **?**

Why (Formal) Specification?

Importance of Requirements Specification

Advantages of formal requirements spec before implementation:

- **No need to decide on algorithm, but sufficient to describe result**
- **Parts of behaviour can be left open (underspecification)**
- **Possibility of code generation, platform/technology independency model-driven development**
- **Formalisation exhibits bugs & missing requirements in early stage**

Two independent formal models (specification, code):

- **Possibility of formal verification**
- **Find more bugs**
- **More trust in resulting system**