CodeQL documentation CodeQL resources

Types

QL is a statically typed language, so each variable must have a declared type.

A type is a set of values. For example, the type `int` is the set of integers. Note that a value can belong to more than one of these sets, which means that it can have more than one type.

The kinds of types in QL are primitive types, classes, character types, class domain types, algebraic datatypes, type unions, and database types.

Primitive types

These types are built in to QL and are always available in the global namespace, independent of the database that you are querying.

1. **boolean**: This type contains the values `true` and `false`.
2. **float**: This type contains 64-bit floating point numbers, such as `6.28` and `-0.618`.
3. **int**: This type contains 32-bit two's complement integers, such as `-1` and `42`.
4. **string**: This type contains finite strings of 16-bit characters.
5. **date**: This type contains dates (and optionally times).

QL has a range of built-in operations defined on primitive types. These are available by using dispatch on expressions of the appropriate type. For example, `1.toString()` is the string representation of the integer constant `1`. For a full list of built-in operations available in QL, see the section on built-ins in the QL language specification.

Classes

You can define your own types in QL. One way to do this is to define a **class**.

Classes provide an easy way to reuse and structure code. For example, you can:

- Group together related values.
- Define member predicates on those values.
- Define subclasses that override member predicates.

A class in QL doesn't "create" a new object, it just represents a logical property. A value is in a particular class if it satisfies that logical property.

Defining a class

To define a class, you write:

1. The keyword `class`.
2. The name of the class. This is an identifier starting with an uppercase letter.
3. The supertypes that the class is derived from via *extends* and/or *instanceof*
4. The body of the class, enclosed in braces.

For example:

```
class OneTwoThree extends int {
  OneTwoThree() { // characteristic predicate
    this = 1 or this = 2 or this = 3
  }

  string getAString() { // member predicate
    result = "One, two or three: " + this.toString()
  }

  predicate isEven() { // member predicate
    this = 2
  }
}
```

This defines a class `OneTwoThree`, which contains the values `1`, `2`, and `3`. The characteristic predicate captures the logical property of "being one of the integers 1, 2, or 3."

`OneTwoThree` extends `int`, that is, it is a subtype of `int`. A class in QL must always have at least one supertype. Supertypes that are referenced with the *extends* keyword are called the **base types** of the class. The values of a class are contained within the intersection of the supertypes (that is, they are in the class domain type). A class inherits all member predicates from its base types.

A class can extend multiple types. For more information, see "Multiple inheritance." Classes can also specialise other types without extending the class interface via *instanceof*, see "Non-extending subtypes.".

To be valid, a class:

- Must not extend itself.
- Must not extend a final class.
- Must not extend types that are incompatible. For more information, see "Type compatibility."

You can also annotate a class. See the list of annotations available for classes.

Class bodies

The body of a class can contain:

- A characteristic predicate declaration.
- Any number of member predicate declarations.
- Any number of field declarations.

When you define a class, that class also inherits all non-private member predicates and fields from its supertypes. You can override those predicates and fields to give them a more specific definition.

Characteristic predicates

These are predicates defined inside the body of a class. They are logical properties that use the variable `this` to restrict the possible values in the class.

Member predicates

These are predicates that only apply to members of a particular class. You can call a member predicate on a value. For example, you can use the member predicate from the above class:

```
1.(OneTwoThree).getAString()
```

This call returns the result `"One, two or three: 1"`.

The expression `(OneTwoThree)` is a cast. It ensures that `1` has type `OneTwoThree` instead of just `int`. Therefore, it has access to the member predicate `getAString()`.

Member predicates are especially useful because you can chain them together. For example, you can use `toUpperCase()`, a built-in function defined for `string`:

```
1.(OneTwoThree).getAString().toUpperCase()
```

This call returns `"ONE, TWO OR THREE: 1"`.

> Note
>
> Characteristic predicates and member predicates often use the variable `this`. This variable always refers to a member of the class—in this case a value belonging to the class `OneTwoThree`. In the characteristic predicate, the variable `this` constrains the values that are in the class. In a member predicate, `this` acts in the same way as any other argument to the predicate.

Fields

These are variables declared in the body of a class. A class can have any number of field declarations (that is, variable declarations) within its body. You can use these variables in predicate declarations inside the class. Much like the variable `this`, fields must be constrained in the characteristic predicate.

For example:

```
class SmallInt extends int {
  SmallInt() { this = [1 .. 10] }
}

class DivisibleInt extends SmallInt {
  SmallInt divisor;   // declaration of the field `divisor`
  DivisibleInt() { this % divisor = 0 }

  SmallInt getADivisor() { result = divisor }
}

from DivisibleInt i
select i, i.getADivisor()
```

In this example, the declaration `SmallInt divisor` introduces a field `divisor`, constrains it in the characteristic predicate, and then uses it in the declaration of the member predicate `getADivisor`. This is similar to introducing variables in a select clause by declaring them in the `from` part.

You can also annotate predicates and fields. See the list of annotations that are available.

Concrete classes

The classes in the above examples are all **concrete** classes. They are defined by restricting the values in a larger type. The values in a concrete class are precisely those values in the intersection of the supertypes that also satisfy the characteristic predicate of the class.

Abstract classes

A class annotated with `abstract`, known as an **abstract** class, is also a restriction of the values in a larger type. However, an abstract class is defined as the union of its subclasses. In particular, for a value to be in an abstract class, it must satisfy the characteristic predicate of the class itself **and** the characteristic predicate of a subclass.

An abstract class is useful if you want to group multiple existing classes together under a common name. You can then define member predicates on all those classes. You can also extend predefined abstract classes: for example, if you import a library that contains an abstract class, you can add more subclasses to it.

Example

If you are writing a security query, you may be interested in identifying all expressions that can be interpreted as SQL queries. You can use the following abstract class to describe these expressions:

```
abstract class SqlExpr extends Expr {
  ...
}
```

Now define various subclasses—one for each kind of database management system. For example, you can define a subclass `class PostgresSqlExpr extends SqlExpr`, which contains expressions passed to some Postgres API that performs a database query. You can define similar subclasses for MySQL and other database management systems.

The abstract class `SqlExpr` refers to all of those different expressions. If you want to add support for another database system later on, you can simply add a new subclass to `SqlExpr`; there is no need to update the queries that rely on it.

> **Important**
>
> You must take care when you add a new subclass to an existing abstract class. Adding a subclass is not an isolated change, it also extends the abstract class since that is a union of its subclasses.

Overriding member predicates

If a class inherits a member predicate from a supertype, you can **override** the inherited definition. You do this by defining a member predicate with the same name and arity as the inherited predicate, and by adding the `override` annotation. This is useful if you want to refine the predicate to give a more specific result for the values in the subclass.

For example, extending the class from the first example:

```
class OneTwo extends OneTwoThree {
  OneTwo() {
    this = 1 or this = 2
  }

  override string getAString() {
    result = "One or two: " + this.toString()
  }
}
```

The member predicate `getAString()` overrides the original definition of `getAString()` from `OneTwoThree`.

Now, consider the following query:

```
from OneTwoThree o
select o, o.getAString()
```

The query uses the "most specific" definition(s) of the predicate `getAString()`, so the results look like this:

o	`getAString()` result
1	One or two: 1
2	One or two: 2
3	One, two or three: 3

In QL, unlike other object-oriented languages, different subtypes of the same types don't need to be disjoint. For example, you could define another subclass of `OneTwoThree`, which overlaps with `OneTwo`:

```
class TwoThree extends OneTwoThree {
  TwoThree() {
    this = 2 or this = 3
  }

  override string getAString() {
    result = "Two or three: " + this.toString()
  }
}
```

Now the value 2 is included in both class types `OneTwo` and `TwoThree`. Both of these classes override the original definition of `getAString()`. There are two new "most specific" definitions, so running the above query gives the following results:

o	`getAString()` result
1	One or two: 1
2	One or two: 2
2	Two or three: 2

o	`getAString()` result
3	Two or three: 3

Multiple inheritance

A class can extend multiple types. In that case, it inherits from all those types.

For example, using the definitions from the above section:

```
class Two extends OneTwo, TwoThree {}
```

Any value in the class `Two` must satisfy the logical property represented by `OneTwo`, **and** the logical property represented by `TwoThree`. Here the class `Two` contains one value, namely 2.

It inherits member predicates from `OneTwo` and `TwoThree`. It also (indirectly) inherits from `OneTwoThree` and `int`.

> **Note**
>
> If a subclass inherits multiple definitions for the same predicate name, then it must override those definitions to avoid ambiguity. Super expressions are often useful in this situation.

Non-extending subtypes

Besides extending base types, classes can also declare `instanceof` relationships with other types. Declaring a class as `instanceof Foo` is roughly equivalent to saying `this instanceof Foo` in the characteristic predicate. The main differences are that you can call methods on `Bar` via `super` and you can get better optimisation.

```
class Foo extends int {
  Foo() { this in [1 .. 10] }

  string fooMethod() { result = "foo" }
}

class Bar instanceof Foo {
  string toString() { result = super.fooMethod() }
}
```

In this example, the characteristic predicate from `Foo` also applies to `Bar`. However, `fooMethod` is not exposed in `Bar`, so the query `select any(Bar b).fooMethod()` results in a compile time error. Note from the example that it is still possible to access methods from instanceof supertypes from within the specialising class with the `super` keyword.

Crucially, the instanceof **supertypes** are not **base types**. This means that these supertypes do not participate in overriding, and any fields of such supertypes are not part of the new class. This has implications on method resolution when complex class hierarchies are involved. The following example demonstrates this.

```
class Interface extends int {
  Interface() { this in [1 .. 10] }
  string foo() { result = "" }
}

class Foo extends int {
  Foo() { this in [1 .. 5] }
  string foo() { result = "foo" }
}

class Bar extends Interface instanceof Foo {
  override string foo() { result = "bar" }
}
```

Here, the method `Bar::foo` does not override `Foo::foo`. Instead, it overrides only `Interface::foo`. This means that `select any(Foo f).foo()` yields `foo`. Had `Bar` been defined as `extends Foo`, then `select any(Foo f).foo()` would yield `bar`.

Character types and class domain types

You can't refer to these types directly, but each class in QL implicitly defines a character type and a class domain type. (These are rather more subtle concepts and don't appear very often in practical query writing.)

The **character type** of a QL class is the set of values satisfying the characteristic predicate of the class. It is a subset of the domain type. For concrete classes, a value belongs to the class if, and only if, it is in the character type. For abstract classes, a value must also belong to at least one of the subclasses, in addition to being in the character type.

The **domain type** of a QL class is the intersection of the character types of all its supertypes, that is, a value belongs to the domain type if it belongs to every supertype. It occurs as the type of `this` in the characteristic predicate of a class.

Algebraic datatypes

> **Note**
>
> The syntax for algebraic datatypes is considered experimental and is subject to change. However, they appear in the standard QL libraries so the following sections should help you understand those examples.

An algebraic datatype is another form of user-defined type, declared with the keyword `newtype`.

Algebraic datatypes are used for creating new values that are neither primitive values nor entities from the database. One example is to model flow nodes when analyzing data flow through a program.

An algebraic datatype consists of a number of mutually disjoint *branches*, that each define a branch type. The algebraic datatype itself is the union of all the branch types. A branch can have arguments and a body. A new value of the branch type is produced for each set of values that satisfy the argument types and the body.

A benefit of this is that each branch can have a different structure. For example, if you want to define an “option type” that either holds a value (such as a `Call`) or is empty, you could write this as follows:

```
newtype OptionCall = SomeCall(Call c) or NoCall()
```

This means that for every `Call` in the program, a distinct `SomeCall` value is produced. It also means that a unique `NoCall` value is produced.

Defining an algebraic datatype

To define an algebraic datatype, use the following general syntax:

```
newtype <TypeName> = <branches>
```

The branch definitions have the following form:

```
<BranchName>(<arguments>) { <body> }
```

- The type name and the branch names must be identifiers starting with an uppercase letter. Conventionally, they start with `T`.
- The different branches of an algebraic datatype are separated by `or`.
- The arguments to a branch, if any, are variable declarations separated by commas.
- The body of a branch is a predicate body. You can omit the branch body, in which case it defaults to `any()`. Note that branch bodies are evaluated fully, so they must be finite. They should be kept small for good performance.

For example, the following algebraic datatype has three branches:

```
newtype T =
  Type1(A a, B b) { body(a, b) }
  or
  Type2(C c)
  or
  Type3()
```

Standard pattern for using algebraic datatypes

Algebraic datatypes are different from classes. In particular, algebraic datatypes don’t have a `toString()` member predicate, so you can’t use them in a select clause.

Classes are often used to extend algebraic datatypes (and to provide a `toString()` predicate). In the standard QL language libraries, this is usually done as follows:

- Define a class `A` that extends the algebraic datatype and optionally declares abstract predicates.
- For each branch type, define a class `B` that extends both `A` and the branch type, and provide a definition for any abstract predicates from `A`.
- Annotate the algebraic datatype with private, and leave the classes public.

For example, the following code snippet from the CodeQL data-flow library for C# defines classes for dealing with tainted or untainted values. In this case, it doesn’t make sense for `TaintType` to extend a database type. It is part of the taint analysis, not the underlying program, so it’s helpful to extend a new type (namely `TTaintType`):

```
private newtype TTaintType =
  TExactValue()
  or
  TTaintedValue()

/** Describes how data is tainted. */
class TaintType extends TTaintType {
  string toString() {
    this = TExactValue() and result = "exact"
    or
    this = TTaintedValue() and result = "tainted"
  }
}

/** A taint type where the data is untainted. */
class Untainted extends TaintType, TExactValue {
}

/** A taint type where the data is tainted. */
class Tainted extends TaintType, TTaintedValue {
}
```

Type unions

Type unions are user-defined types that are declared with the keyword `class`. The syntax resembles type aliases, but with two or more type expressions on the right-hand side.

Type unions are used for creating restricted subsets of an existing algebraic datatype, by explicitly selecting a subset of the branches of that datatype and binding them to a new type. Type unions of database types are also supported.

You can use a type union to give a name to a subset of the branches from an algebraic datatype. In some cases, using the type union over the whole algebraic datatype can avoid spurious recursion in predicates. For example, the following construction is legal:

```
newtype InitialValueSource =
  ExplicitInitialization(VarDecl v) { exists(v.getInitializer()) } or
  ParameterPassing(Call c, int pos) { exists(c.getParameter(pos)) } or
  UnknownInitialGarbage(VarDecl v) { not exists(DefiniteInitialization di | v =
target(di)) }

class DefiniteInitialization = ParameterPassing or ExplicitInitialization;

VarDecl target(DefiniteInitialization di) {
  di = ExplicitInitialization(result) or
  exists(Call c, int pos | di = ParameterPassing(c, pos) and
                          result = c.getCallee().getFormalArg(pos))
}
```

However, a similar implementation that restricts `InitialValueSource` in a class extension is not valid. If we had implemented `DefiniteInitialization` as a class extension instead, it would trigger a type test for `InitialValueSource`. This results in an illegal recursion `DefiniteInitialization -> InitialValueSource -> UnknownInitialGarbage -> ¬DefiniteInitialization` since `UnknownInitialGarbage` relies on `DefiniteInitialization`:

```
// THIS WON'T WORK: The implicit type check for InitialValueSource involves an
illegal recursion
```

```
  // DefiniteInitialization -> InitialValueSource -> UnknownInitialGarbage ->
  ¬DefiniteInitialization!
  class DefiniteInitialization extends InitialValueSource {
    DefiniteInitialization() {
      this instanceof ParameterPassing or this instanceof ExplicitInitialization
    }
    // ...
  }
```

Type unions are supported from release 2.2.0 of the CodeQL CLI.

Database types

Database types are defined in the database schema. This means that they depend on the database that you are querying, and vary according to the data you are analyzing.

For example, if you are querying a CodeQL database for a Java project, the database types may include `@ifstmt`, representing an if statement in the Java code, and `@variable`, representing a variable.

Type compatibility

Not all types are compatible. For example, `4 < "five"` doesn't make sense, since you can't compare an `int` to a `string`.

To decide when types are compatible, there are a number of different "type universes" in QL.

The universes in QL are:

- One for each primitive type (except `int` and `float`, which are in the same universe of "numbers").
- One for each database type.
- One for each branch of an algebraic datatype.

For example, when defining a class this leads to the following restrictions:

- A class can't extend multiple primitive types.
- A class can't extend multiple different database types.
- A class can't extend multiple different branches of an algebraic datatype.