Fixing common type problems

If you're having problems with type checks, this page can help. To learn more, read about Dart's type system, and see these other resources.

> **Help us improve this page!** If you encounter a warning or error that isn't listed here, please file an issue by clicking the **bug icon** at the top right. Include the **warning or error message** and, if possible, the code for both a small reproducible case and its correct equivalent.

Troubleshooting

No type errors

If you're not seeing expected errors or warnings, make sure that you're using the latest version of Dart and you have properly configured your IDE or editor.

You can also run analysis on your program using the command line with the `dart analyze` command.

To verify that analysis is working as expected, try adding the following code to a Dart file.

```
X static analysis: error/warning
  bool b = [0][0];
```

If properly configured, the analyzer produces the following error:

```
error - A value of type 'int' can't be assigned to a variable of type 'bool'. Try changing the type of
the variable, or casting the right-hand type to 'bool'. - invalid_assignment
```

Static errors and warnings

This section shows how to fix some of the errors and warnings you might see from the analyzer or an IDE.

Static analysis can't catch all errors. For help fixing errors that appear only at runtime, see Runtime errors.

Undefined member

```
error - The <member> '...' isn't defined for the type '...' - undefined_<member>
```

These errors can appear under the following conditions:

- A variable is statically known to be some supertype, but the code assumes a subtype.
- A generic class has a bounded type parameter, but an instance creation expression of the class omits the type argument.

Example 1: A variable is statically known to be some supertype, but the code assumes a subtype

In the following code, the analyzer complains that `context2D` is undefined:

```
X static analysis: error/warning
  var canvas = querySelector('canvas')!;
  canvas.context2D.lineTo(x, y);
```

```
error - The getter 'context2D' isn't defined for the type 'Element'. Try importing the library that
defines 'context2D', correcting the name to the name of an existing getter, or defining a getter or
field named 'context2D'. - undefined_getter
```

Fix: Replace the definition of the member with an explicit type declaration or a downcast

The return type of `querySelector()` is `Element?` (which the `!` converts to `Element`), but the code assumes that it's the subtype `CanvasElement` (which defines `context2D`). The `canvas` field is declared as `var`, which allows Dart to infer `canvas` to be an `Element`.

You can fix this error with an explicit downcast:

✓ static analysis: success

```
var canvas = querySelector('canvas') as CanvasElement;
canvas.context2D.lineTo(x, y);
```

Otherwise, use `dynamic` in situations where you cannot use a single type:

✓ static analysis: success

```
dynamic canvasOrImg = querySelector('canvas, img');
var width = canvasOrImg.width;
```

Example 2: Omitted type parameters default to their type bounds

Consider the following **generic class** with a **bounded type parameter** that extends `Iterable`:

```
class C<T extends Iterable> {
  final T collection;
  C(this.collection);
}
```

The following code creates a new instance of this class (omitting the type argument) and accesses its `collection` member:

✗ static analysis: error/warning

```
var c = C(Iterable.empty()).collection;
c.add(2);
```

```
error - The method 'add' isn't defined for the type 'Iterable'. Try correcting the name to the name of
an existing method, or defining a method named 'add'. - undefined_method
```

While the List type has an `add()` method, Iterable does not.

Fix: Specify type arguments or fix downstream errors

When a generic class is instantiated without explicit type arguments, each type parameter defaults to its type bound (`Iterable` in this example) if one is explicitly given, or `dynamic` otherwise.

You need to approach fixing such errors on a case-by-case basis. It helps to have a good understanding of the original design intent.

Explicitly passing type arguments is an effective way to help identify type errors. For example, if you change the code to specify `List` as a type argument, the analyzer can detect the type mismatch in the constructor argument. Fix the error by providing a constructor argument of the appropriate type, such as a list literal:

✓ static analysis: success

```
var c = C<List>([]).collection;
c.add(2);
```

Invalid method override

```
error - '...'  isn't a valid override of '...' - invalid_override
```

These errors typically occur when a subclass tightens up a method's parameter types by specifying a subclass of the original class.

> **Note:** This issue can also occur when a generic subclass neglects to specify a type. For more information, see Missing type arguments.

Example

In the following example, the parameters to the `add()` method are of type `int`, a subtype of `num`, which is the parameter type used in the parent class.

```
X static analysis: error/warning
abstract class NumberAdder {
  num add(num a, num b);
}

class MyAdder extends NumberAdder {
  @override
  num add(int a, int b) => a + b;
}
```

```
error - 'MyAdder.add' ('num Function(int, int)') isn't a valid override of 'NumberAdder.add' ('num
Function(num, num)'). - invalid_override
```

Consider the following scenario where floating point values are passed to an `MyAdder`:

```
X runtime: error
NumberAdder adder = MyAdder();
adder.add(1.2, 3.4);
```

If the override were allowed, the code would raise an error at runtime.

Fix: Widen the method's parameter types

The subclass's method should accept every object that the superclass's method takes.

Fix the example by widening the types in the subclass:

```
✓ static analysis: success
abstract class NumberAdder {
  num add(num a, num b);
}

class MyAdder extends NumberAdder {
  @override
  num add(num a, num b) => a + b;
}
```

For more information, see Use proper input parameter types when overriding methods.

> **Note:** If you have a valid reason to use a subtype, you can use the covariant keyword.

Missing type arguments

```
error - '...'  isn't a valid override of '...' - invalid_override
```

Example

In the following example, `Subclass` extends `Superclass<T>` but doesn't specify a type argument. The analyzer infers `Subclass<dynamic>`, which results in an invalid override error on `method(int)`.

✗ static analysis: error/warning

```
class Superclass<T> {
  void method(T param) { ... }
}

class Subclass extends Superclass {
  @override
  void method(int param) { ... }
}
```

```
error - 'Subclass.method' ('void Function(int)') isn't a valid override of 'Superclass.method' ('void
Function(dynamic)'). - invalid_override
```

Fix: Specify type arguments for the generic subclass

When a generic subclass neglects to specify a type argument, the analyzer infers the `dynamic` type. This is likely to cause errors.

You can fix the example by specifying the type on the subclass:

✓ static analysis: success

```
class Superclass<T> {
  void method(T param) { ... }
}

class Subclass extends Superclass<int> {
  @override
  void method(int param) { ... }
}
```

Consider using the analyzer in *strict raw types* mode, which ensures that your code specifies generic type arguments. Here's an example of enabling strict raw types in your project's `analysis_options.yaml` file:

```
analyzer:
  language:
    strict-raw-types: true
```

To learn more about customizing the analyzer's behavior, see Customizing static analysis.

Unexpected collection element type

```
error - A value of type '...' can't be assigned to a variable of type '...' - invalid_assignment
```

This sometimes happens when you create a simple dynamic collection and the analyzer infers the type in a way you didn't expect. When you later add values of a different type, the analyzer reports an issue.

Example

The following code initializes a map with several (`String`, `int`) pairs. The analyzer infers that map to be of type `<String, int>` but the code seems to assume either `<String, dynamic>` or `<String, num>`. When the code adds a (`String`, `double`) pair, the analyzer complains:

✗ static analysis: error/warning

```
// Inferred as Map<String, int>
var map = {'a': 1, 'b': 2, 'c': 3};
map['d'] = 1.5;
```

```
error - A value of type 'double' can't be assigned to a variable of type 'int'. Try changing the type
of the variable, or casting the right-hand type to 'int'. - invalid_assignment
```

Fix: Specify the type explicitly

The example can be fixed by explicitly defining the map's type to be `<String, num>`.

✓ static analysis: success

```
var map = <String, num>{'a': 1, 'b': 2, 'c': 3};
map['d'] = 1.5;
```

Alternatively, if you want this map to accept any value, specify the type as `<String, dynamic>`.

Constructor initialization list super() call

```
error - The superclass call must be last in an initializer list: 'super(...)'. -
invalid_super_invocation
```

This error occurs when the `super()` call is not last in a constructor's initialization list.

Example

✗ static analysis: error/warning

```
HoneyBadger(Eats food, String name)
    : super(food),
      _name = name { ... }
```

```
error - The superclass call must be last in an initializer list: 'super(food)'. -
invalid_super_invocation
```

Fix: Put the `super()` call last

The compiler can generate simpler code if it relies on the `super()` call appearing last.

Fix this error by moving the `super()` call:

✓ static analysis: success

```
HoneyBadger(Eats food, String name)
    : _name = name,
      super(food) { ... }
```

The argument type … can't be assigned to the parameter type …

```
error - The argument type '...' can't be assigned to the parameter type '...'. -
argument_type_not_assignable
```

In Dart 1.x `dynamic` was both a top type (supertype of all types) and a bottom type (subtype of all types) depending on the context. This meant it was valid to assign, for example, a function with a parameter of type `String` to a place that expected a function type with a parameter of `dynamic`.

However, in Dart 2 using a parameter type other than `dynamic` (or another *top* type, such as `Object?`) results in a compile-time error.

Example

✗ static analysis: error/warning

```
void filterValues(bool Function(dynamic) filter) {}
filterValues((String x) => x.contains('Hello'));
```

```
error - The argument type 'bool Function(String)' can't be assigned to the parameter type 'bool
Function(dynamic)'. - argument_type_not_assignable
```

Fix: Add type parameters *or* cast from dynamic explicitly

When possible, avoid this error by adding type parameters:

✓ static analysis: success

```
void filterValues<T>(bool Function(T) filter) {}
filterValues<String>((x) => x.contains('Hello'));
```

Otherwise use casting:

✓ static analysis: success

```
void filterValues(bool Function(dynamic) filter) {}
filterValues((x) => (x as String).contains('Hello'));
```

Runtime errors

Dart enforces a sound type system. Roughly, this means you can't get into a situation where the value stored in a variable is different from the variable's static type. Like most modern statically typed languages, Dart accomplishes this with a combination of static (compile-time) and dynamic (runtime) checking.

For example, the following type error is detected at compile-time:

✗ static analysis: error/warning

```
List<int> numbers = [1, 2, 3];
List<String> string = numbers;
```

Since neither `List<int>` nor `List<String>` is a subtype of the other, Dart rules this out statically. You can see other examples of these static analysis errors in Unexpected collection element type.

The errors discussed in the remainder of this section are reported at runtime.

Invalid casts

To ensure type safety, Dart needs to insert *runtime* checks in some cases. Consider the following `assumeStrings` method:

✓ static analysis: success

```
void assumeStrings(dynamic objects) {
  List<String> strings = objects; // Runtime downcast check
  String string = strings[0]; // Expect a String value
}
```

The assignment to `strings` is *downcasting* the `dynamic` to `List<String>` implicitly (as if you wrote `as List<String>`), so if the value you pass in `objects` at runtime is a `List<String>`, then the cast succeeds.

Otherwise, the cast will fail at runtime:

✗ runtime: error

```
assumeStrings(<int>[1, 2, 3]);
```

```
Exception: type 'List<int>' is not a subtype of type 'List<String>'
```

Fix: Tighten or correct types

Sometimes, lack of a type, especially with empty collections, means that a `<dynamic>` collection is created, instead of the typed one you intended. Adding an explicit type argument can help:

✓ runtime: success

```
var list = <String>[];
list.add('a string');
list.add('another');
assumeStrings(list);
```

You can also more precisely type the local variable, and let inference help:

✓ runtime: success

```
List<String> list = [];
list.add('a string');
list.add('another');
assumeStrings(list);
```

In cases where you are working with a collection that you don't create, such as from JSON or an external data source, you can use the cast() method provided by `Iterable` implementations, such as `List`.

Here's an example of the preferred solution: tightening the object's type.

✓ runtime: success

```
Map<String, dynamic> json = fetchFromExternalSource();
var names = json['names'] as List;
assumeStrings(names.cast<String>());
```

Appendix

The covariant keyword

Some (rarely used) coding patterns rely on tightening a type by overriding a parameter's type with a subtype, which is invalid. In this case, you can use the `covariant` keyword to tell the analyzer that you are doing this intentionally. This removes the static error and instead checks for an invalid argument type at runtime.

The following shows how you might use `covariant`:

```
✓ static analysis: success
  class Animal {
    void chase(Animal x) { ... }
  }

  class Mouse extends Animal { ... }

  class Cat extends Animal {
    @override
    void chase(covariant Mouse x) { ... }
  }
```

Although this example shows using `covariant` in the subtype, the `covariant` keyword can be placed in either the superclass or the subclass method. Usually the superclass method is the best place to put it. The `covariant` keyword applies to a single parameter and is also supported on setters and fields.