Generics

If you look at the API documentation for the basic array type, `List`, you'll see that the type is actually `List<E>`. The <...> notation marks List as a *generic* (or *parameterized*) type—a type that has formal type parameters. By convention, most type variables have single-letter names, such as E, T, S, K, and V.

Why use generics?

Generics are often required for type safety, but they have more benefits than just allowing your code to run:

- Properly specifying generic types results in better generated code.
- You can use generics to reduce code duplication.

If you intend for a list to contain only strings, you can declare it as `List<String>` (read that as "list of string"). That way you, your fellow programmers, and your tools can detect that assigning a non-string to the list is probably a mistake. Here's an example:

```dart
✗ static analysis: failure
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // Error
```

Another reason for using generics is to reduce code duplication. Generics let you share a single interface and implementation between many types, while still taking advantage of static analysis. For example, say you create an interface for caching an object:

```dart
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}
```

You discover that you want a string-specific version of this interface, so you create another interface:

```dart
abstract class StringCache {
  String getByKey(String key);
  void setByKey(String key, String value);
}
```

Later, you decide you want a number-specific version of this interface... You get the idea.

Generic types can save you the trouble of creating all these interfaces. Instead, you can create a single interface that takes a type parameter:

```dart
abstract class Cache<T> {
  T getByKey(String key);
  void setByKey(String key, T value);
}
```

In this code, T is the stand-in type. It's a placeholder that you can think of as a type that a developer will define later.

Using collection literals

List, set, and map literals can be parameterized. Parameterized literals are just like the literals you've already seen, except that you add `<type>` (for lists and sets) or `<keyType, valueType>` (for maps) before the opening bracket. Here is an example of using typed literals:

```dart
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Using parameterized types with constructors

To specify one or more types when using a constructor, put the types in angle brackets (`<...>`) just after the class name. For example:

```dart
var nameSet = Set<String>.from(names);
```

The following code creates a map that has integer keys and values of type View:

```dart
var views = Map<int, View>();
```

Generic collections and the types they contain

Dart generic types are *reified*, which means that they carry their type information around at runtime. For example, you can test the type of a collection:

```dart
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

> ⓘ 提示
>
> In contrast, generics in Java use *erasure*, which means that generic type parameters are removed at runtime. In Java, you can test whether an object is a List, but you can't test whether it's a `List<String>`.

Restricting the parameterized type

When implementing a generic type, you might want to limit the types that can be provided as arguments, so that the argument must be a subtype of a particular type. You can do this using `extends`.

A common use case is ensuring that a type is non-nullable by making it a subtype of `Object` (instead of the default, `Object?`).

```dart
class Foo<T extends Object> {
  // Any type provided to Foo for T must be non-nullable.
}
```

You can use `extends` with other types besides `Object`. Here's an example of extending `SomeBaseClass`, so that members of `SomeBaseClass` can be called on objects of type `T`:

```dart
class Foo<T extends SomeBaseClass> {
  // Implementation goes here...
  String toString() => "Instance of 'Foo<$T>'";
}

class Extender extends SomeBaseClass {...}
```

It's OK to use `SomeBaseClass` or any of its subtypes as the generic argument:

```dart
var someBaseClassFoo = Foo<SomeBaseClass>();
var extenderFoo = Foo<Extender>();
```

It's also OK to specify no generic argument:

```dart
var foo = Foo();
print(foo); // Instance of 'Foo<SomeBaseClass>'
```

Specifying any non-`SomeBaseClass` type results in an error:

```dart
X static analysis: failure
  var foo = Foo<Object>();
```

Using generic methods

Methods and functions also allow type arguments:

```dart
T first<T>(List<T> ts) {
  // Do some initial work or error checking, then...
  T tmp = ts[0];
  // Do some additional checking or processing...
  return tmp;
}
```

Here the generic type parameter on `first` (`<T>`) allows you to use the type argument `T` in several places:

- In the function's return type (`T`).
- In the type of an argument (`List<T>`).
- In the type of a local variable (`T tmp`).