techLog

Explore | Android | Flutter | Software Engineering

support codementor

Dart Generics

Posted on April 30, 2020 in Dart

Introduction

Generics are used to apply stronger type checks at compile time. They enforce type-safety in code. For example, in collections the type-safety is enforced by holding same type of data. Generics help write reusable classes, methods/functions for different data types.

Type Safety: Programming concept that allows a memory block to contain only one type of data.

The concept of Generics in Dart, is similar to Java's generics and C++'s templates.

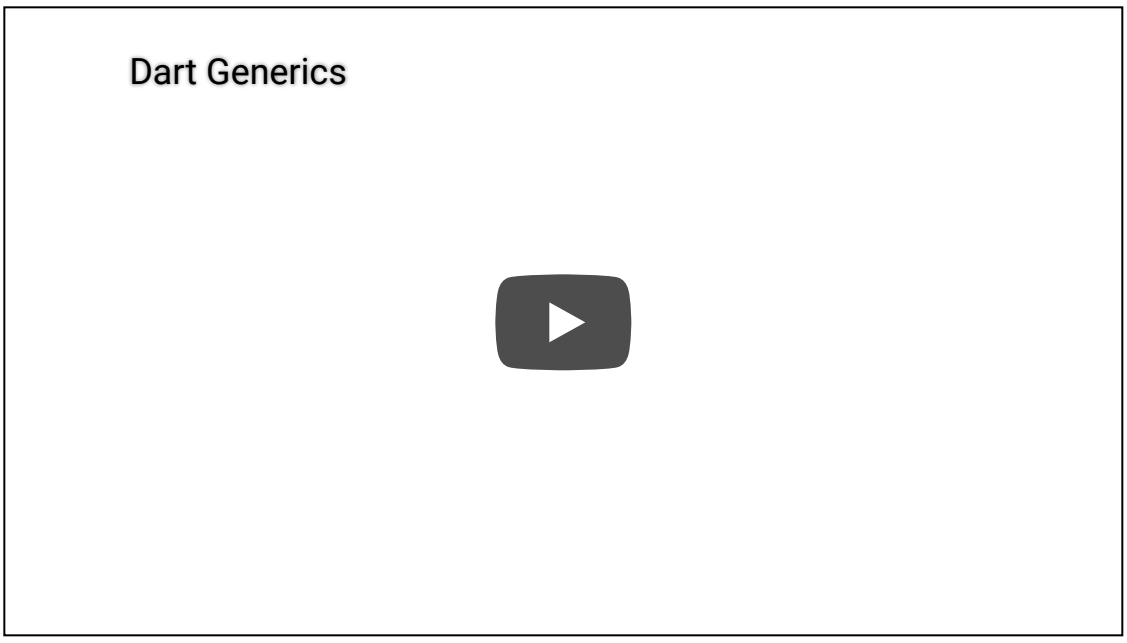
Dart's collection can hold different data types in one collection. It's optional in Dart to mention data type for a value. Usually, the variable's data type is inferred automatically. For example, `var myVar = 5;` will infer `myVar`'s dataType as `int`. The following code is valid in Dart:

```
void main() {
  List items = [1, "Empty", 1.0];
  print(items);
}
```

Output:

```
[1, Empty, 1.0]
```

Check out YouTube Video



Declaring Type-safe Collections

The angular brackets `<>`, with data type enclosed, is used to declare the collection of given data type to ensure type-safety.

Syntax:

```
CollectionType <dataType> identifier = CollectionType <dataType>();
```

Example:

```
List<int> numbers = List<int>();
```

Generics are parameterized and use type variables notations to restrict the type of data. These type variables are represented using single letter names. A few typically used single letter names are:

- **E:** The letter `E` is used to represent the element type in a collection like List.
- **K:** The letter `K` is used to represent the key type in associative collections like Map.
- **V:** The letter `V` is used to represent the value type in associative collections like Map.
- **R:** The letter `R` is used to represent the return type of a method or function.

You can also use a single letter of your choice or a descriptive word for parameter names / generics. Let's explore these two options in the following example.

```
///Example #1: Demonstrating use of single letter and descriptive wor

//A class for grocery product
class Product {
  final int id;
  final double price;
  final String title;
  Product(this.id, this.price, this.title);

  @override
  String toString() {
    return "Price of ${this.title} is \$${this.price}";
  }
}

//A class for product's inventory
class Inventory {
  final int amount;

  Inventory(this.amount);
```

```
  @override
  String toString() {
    return "Inventory amount: $amount";
  }
}

//Custom type variables- Single letter
class Store<P, I> {
  final HashMap<P, I> catalog = HashMap<P, I>();

  List<P> get products => catalog.keys.toList();

  void updateInventory(P product, I inventory) {
    catalog[product] = inventory;
  }

  void printProducts() {
    catalog.keys.forEach(
      (product) => print("Product: $product, " + catalog[product].toS
    );
  }
}

//Custom type variables- Descriptive
class MyStore<MyProduct, MyInventory> {
  final HashMap<MyProduct, MyInventory> catalog =
      HashMap<MyProduct, MyInventory>();

  List<MyProduct> get products => catalog.keys;

  void updateInventory(MyProduct product, MyInventory inventory) {
    catalog[product] = inventory;
  }

  void printProducts() {
    catalog.keys.forEach(
      (product) => print("Product: $product, " + catalog[product].toS
    );
  }
}

//Demonstrating single letter vs descriptive names for generics.
//Both variations have the same results.
void mainCustomParams() {
  Product milk = Product(1, 5.99, "Milk");
  Product bread = Product(2, 4.50, "Bread");

  //Using single letter names for Generics
  Store<Product, Inventory> store1 = Store<Product, Inventory>();
  store1.updateInventory(milk, Inventory(20));
```

```
  store1.updateInventory(bread, Inventory(15));
  store1.printProducts();

  //Using descriptive names for Generics
  MyStore<Product, Inventory> store2 = MyStore<Product, Inventory>();
  store2.updateInventory(milk, Inventory(20));
  store2.updateInventory(bread, Inventory(15));
  store2.printProducts();
}
```

Output:

```
Product: Price of Bread is $4.5, Inventory amount: 15
Product: Price of Milk is $5.99, Inventory amount: 20
Product: Price of Bread is $4.5, Inventory amount: 15
Product: Price of Milk is $5.99, Inventory amount: 20
```

Code Re-use

Generics help to write re-usable code. Generics for classes and methods enable reusing same code for different implementations of data types. Let's explore the details below.

Generics Methods / Functions

Let's define a method `lastItem()` which take the list of products, and returns the last item in the list. In this method, parameter `T` is returned, a list of type `T` elements is passed to method, and `T` type is stored in variable `last`.

In `mainGenericMethods()` method, there are two different data types are using `lastItem()` method to retrieve last item in their respective lists. First list is made up of `Product` items. Second list is made up of `int` data type. The `T lastItem<T>(List<T> products)` method using generic `T` parameter to support multiple data types using the same method. This is one excellent example of writing reusable functions/class methods.

```
//Example #2: Generics methods

//Function's return type (T).
//Function's argument (List<T>).
//Function's local variable (T last).
T lastItem<T>(List<T> products) {
  T last = products.last;
```

```
  return last;
}

mainGenericMethods() {
  Store<Product, Inventory> store = Store<Product, Inventory>();
  Product milk = Product(1, 5.99, "Milk");
  Product bread = Product(2, 4.50, "Bread");
  store.updateInventory(milk, Inventory(20));
  store.updateInventory(bread, Inventory(15));

  //Data type of `Product` is being used
  Product product = lastItem(store.products);
  print("Last item of Product type: ${product}");

  //Demonstrating using another type of data on same `lastItem()` met
  List<int> items = List<int>.from([1, 2, 3, 4, 5]);
  int item = lastItem(items);
  print("Last item of int type: ${item}");
}
```

Output:

```
Last item of Product type: Price of Bread is $4.5
Last item of int type: 5
```

Generics Classes

Generic classes help to restrict the type of values accepted by the class. These supplied values are known as generic parameter(s). In the following example, class `FreshProduce` is restricted to accept only `Product` dataType. It's okay to use `FreshProduce` without `<Product>` parameter, and it will assume it as of type `Product`. However, if any other data type other than allowed type is passed, you'll see the compile time error.

```
///Example #3: Using Generics for classes

//Restricting the type of values that can be supplied to the class.
//FreshProduce class can only accept of Product type when T extends P
class FreshProduce<T extends Product> {
  FreshProduce(int i, double d, String s);

  String toString() {
    return "Instance of Type: ${T}";
  }
```

```
}

mainGenericClass() {
  //Using `Product` parameter accepted by FreshProduce class
  FreshProduce<Product> spinach = FreshProduce<Product>(3, 3.99, "Spi
  print(spinach.toString());

  //Passing
  FreshProduce spinach2 = FreshProduce(3, 3.99, "Spinach");
  print(spinach2.toString());

  //This code will give compile time error complaining that Object is
  //FreshProduce<Object> spinach3 = FreshProduce<Object>(3, 3.99, "Sp
  //print(spinach3.toString());
}
```

Output:

```
Instance of Type: Product
Instance of Type: Product
```

Generic collections

In this section, let's checkout the type-safe implementations for some of the Dart's collection literals:

- List
- Queue
- Set
- Map

List

In this example, a list of `int` datatype is constructed using parameterized constructor - another example of using generics. Additionally, two more items are added to list.

Now, adding a dataType other than `int` will throw a compile time error. Copy this code in Dart Pad , and uncomment `theList.add(4.0);` to see compile time error yourself.

```
void mainList() {
  //using parameterized types with constructors
```

```
  List<int> theList = List<int>.from([1]);
  theList.add(2);
  theList.add(3);

  //Adding double data type will throw compile time error
  //theList.add(4.0);

  //iterate over list and print all items
  print("Printing items in Dart List");
  for (int item in theList) {
    print(item);
  }
}
```

Output:

```
Printing items in Dart List
1
2
3
```

Queue

In this example, a queue of `double` datatype is constructed using parameterized constructor - another example of using generics. Additionally, two more items are added to the queue. Adding a different dataType `String` will throw a compile time error.

```
void mainQueue() {
  //using parameterized types with constructors
  Queue<double> theQueue = Queue<double>.from([1.0]);
  theQueue.add(2.0);
  theQueue.add(3.0);

  //Adding String data type will throw compile time error
  //theQueue.add("4.0");

  print("Printing items in Dart Queue");
  //iterate over queue and print all items
  for (double item in theQueue) {
    print(item);
  }
}
```

Output:

```
Printing items in Dart Queue
1.0
2.0
3.0
```

Set

In this example, a set of `String` datatype is constructed using parameterized constructor - another example of using generics. Additionally, two more items are added to the set. Adding a different dataType `int` will throw a compile time error.

```
void mainSet() {
  Set<String> theSet = Set<String>.from({"1"});
  theSet.add("2");
  theSet.add("3");

  //Adding int data type will throw compile time error
  //theSet.add(3);

  print("Printing items in Dart Set");
  //iterate over set and print all items
  for (String item in theSet) {
    print(item);
  }
}
```

Output:

```
Printing items in Dart Set
1
2
3
```

Map

In this example, a Map of `String` datatype is constructed using parameterized constructor - another example of using generics. One more item is added to the Map. Adding a different dataType `int` will throw a compile time error.

```
void mainMap() {
  Map<int, String> theMap = {1: 'Dart'};
  theMap[2] = 'Flutter';

  //Adding int data type for String value will throw compile time err
  //theMap[3] = 3;

  print("Printing key:value pairs in Dart Map");
  //iterate over map and print all entries
  for (MapEntry mapEntry in theMap.entries) {
    print("${mapEntry.key} : ${mapEntry.value}");
  }
}
```

Output:

```
Printing key:value pairs in Dart Map
1 : Dart
2 : Flutter
```

Summary

In this article, we learned how to use generics in Dart. We saw how generics can be useful in writing type-safe and reusable code.

That's it for this article. Check out the Dart Vocabulary Series for other Dart stuff.

Source Code

Please checkout the source code at Github here

References

1. DartPad
2. Dart Generics

Happy Darting :)

_Liked the article? Couldn't find a topic of your interest? Please leave a comment or reach out at twitter about the topics you would like me to share!

BTW I love cupcakes and coffee both :)_

Follow me at Medium

generics dart cross-platform flutter code-recipes development

Like this article? Share it with your friends!