Chapter No.3
"Object Creation"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.3 "Object Creation"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Sergey Akopkokhyants is a software architect with more than 20 years of professional experience in designing and developing client- and server-side applications. He is also a certified Java developer and a project manager. He has general knowledge of many tools, languages, and platforms.

For the past 5 years, Sergey has been responsible for customizing and producing web-oriented applications for wholesale business management solutions projects for several worldwide mobile communication companies. His responsibilities have been architecture design and guidance of client software development using Flex, ActionScript, HTML, JavaScript, TypeScript, Dart, and client-server integration with Java. He is also a founder and active contributor in several open source projects on GitHub, such as Dart Web Toolkit (DWT) and Angular Dart UI.

Sergey is passionate about web design and development and likes sharing his expertise with others, helping them develop their skills and increase their productivity. He has also reviewed the books Learning Dart and Dart Cookbook, both by Packt Publishing.

I would like to thank my wife, Lada, and my parents, Alexander and Tamara, for their constant and invaluable support.

I also wish to express my deep gratitude and appreciation to Jeffrey Johnson for his valuable feedback.

For More Information:
www.packtpub.com/application-development/mastering-dart
Mastering Dart

It's only been a few years since the Dart programming language was born and it has become eminent and well-known not only within Google, but also in the wider community of developers. It has grown into a powerful tool to help developers produce efficient and consistent source code for web clients and standalone and server-side programs. The Dart development team and independent contributors have created some good documentation, so getting started with programming using Dart isn't that hard. You can program within different development environments such as the Eclipse-based Dart Editor; IDE plugins for IntelliJ IDEA and WebStorm; and text editor plugins for Sublime Text 3, Emacs, and Vim.

Actual development on Dart is quite similar to Java, C#, or any other object-oriented languages, so you can have your first short application up and running in a short amount of time. To go further, you can use the tutorials, code labs, and examples on the official Dart website (https://www.dartlang.org/). If you want to improve your level of expertise, you can read through the set of books on Dart that have been published in the last couple of years. You should always bear in mind that creating high-level, secure, and internationally compliant code is more complex than the application created in the beginning.

This book is designed to help you make the most effective use of the Dart programming language, its SDK libraries, and a rich set of publicly available packages. Feel free to refer to any chapter that you need to get more in-depth knowledge about. If you feel you have missed something major, go back and read the earlier chapters. The chapters in the book are arranged in such a way that you can easily learn more in a step by step manner.

I enjoy working with the Dart programming language and am really happy to share my enthusiasm and expertise with you to help you make use of the language in a more effective and comfortable manner.

What This Book Covers

Chapter 1, Beyond Dart's Basics, helps you take a step further to understand the concept of programming in the Dart language. It shows the best practices to use functions and closures and dives into the world of classes and mixins.

Chapter 2, Advanced Techniques and Reflection, lets you get a firm grasp on how to use generics, errors and exceptions, and annotations, and it takes you through Mirrors of reflection.

Chapter 3, Object Creation, helps you understand how objects can be created and will help you make a right choice that will be useful in different business cases.

For More Information:  
www.packtpub.com/application-development/mastering-dart
Chapter 4, Asynchronous Programming, explores advanced technologies to organize asynchronous code execution. It shows you the best practices of Future, zones, and isolates.

Chapter 5, The Stream Framework, shows you how Dart manages streams. It teaches you how the single-subscription and broadcast streams can be used in different cases and scenarios.

Chapter 6, The Collection Framework, introduces you to the Dart collection frameworks. It shows which data structure of a collection is appropriate in specific cases based on patterns and performance consideration.

Chapter 7, Dart and JavaScript Interoperation, shows you how to use Dart and JavaScript together to build web applications. It lists problems and shows solutions you can use to communicate between Dart and JavaScript and the existing JavaScript program.

Chapter 8, Internalization and Localization, explains how the i18n and l10n access can be embedded into your code to help you design and develop web applications that enable easy localization for different cultures, regions, and languages.

Chapter 9, Client-to-server Communication, helps you organize and create client-to-server communications. It has the answer on presumably the most important questions about the right choice of libraries and packages in specific scenarios.

Chapter 10, Advanced Storage, explains how can store data locally on a client, break the storage limits, and prevent security issues in your web applications. It will again touch upon cookies and demonstrate how to use Web Storage and the elaborate, more powerful, and more useful IndexedDB to store a large amount of data in the web browser.

Chapter 11, Supporting Other HTML5 Features, introduces you to different HTML5 features such as notifications, native drag-and-drop, geolocation, and canvas.

Chapter 12, Security Aspects, helps you understand the different aspects of security in web applications.

For More Information:
www.packtpub.com/application-development/mastering-dart
In this chapter, we will talk about the creation of objects. We will see how and when to create them using best practices from the programming world, and then find a place for them to be accommodated in Dart. The different techniques covered here will help us to make correct choices that will be useful in different business cases.

In this chapter, we will cover the following topics:

- A generative constructor
- A constructor with optional parameters
- A named constructor
- A redirecting constructor
- A private constructor
- A factory constructor
- A constant constructor
- Initialization of variables
- Syntactic sugar

Creating an object

A class is a blueprint for objects. The process of creating objects from a class is called instantiation. An object can be instantiated with a new statement from a class or through reflection. It must be instantiated before it is used.

A class contains a constructor method that is invoked to create objects from the class. It always has the same name as the class. Dart defines two types of constructors: generative and factory constructors.
A generative constructor

A **generative** constructor consists of a constructor name, a constructor parameter list, either a redirect clause or an initializer list, and an optional body. Dart always calls the generative constructor first when the class is being instantiated, as shown in the following code:

```dart
class SomeClass {
  // Default constructor
  SomeClass();
}
main() {
  var some = new SomeClass();
}
```

If the constructor is not defined, Dart creates an implicit one for us as follows:

```dart
class AnyClass {
  // implicit constructor
}
main() {
  var some = new AnyClass();
}
```

The main purpose of a generative constructor is to safely initialize the instance of a class. This initialization takes place inside a class and ensures that the instantiating object is always in a valid state.

A constructor with optional parameters

A constructor is a method of a class and has parameters to specify the initial state or other important information about the class. There are required and optional parameters in a constructor. The optional parameters can either be a set of named parameters or a list of positional parameters. Dart doesn't support method overload; hence, the ability to have optional parameters can be very handy.

Dart does not allow you to combine named and positional optional parameters.

For More Information:

www.packtpub.com/application-development/mastering-dart
Let's take a look at the following Car class constructor with optional positional parameters:

```dart
// Class Car
class Car {
  String color;
  int weight;

  Car([this.color = 'white', this.weight = 1000]);
}
```

We can omit one or all the parameters to create an object of the class as follows:

```dart
import 'car_optional_parameters.dart';

main() {
  var car = new Car('blue');
  var car2 = new Car();
}
```

Let's take a look at the following Car class constructor that uses optional named parameters:

```dart
// Class Car
class Car {
  String color;
  int weight;

  Car({this.color:'white', this.weight:1000});
}
```

In the following main code, I used the named parameters to create an instance of the object:

```dart
import 'car_named_parameters.dart';

main() {
  var car = new Car(weight:750, color:'blue');
}
```

So, which kind of optional parameters are better? I recommend the use of named parameters due to the following reasons:

- Here, you need not remember the place of the parameters
- It gives you a better explanation of what the parameters do

For More Information:
www.packtpub.com/application-development/mastering-dart
A named constructor

Let's say we want to create a `Collection` class to store all our data in one place. We can do it as follows:

```dart
library collection;

// Collection class
class Collection {
  // We save data here
  List _data;

  // Default constructor
  Collection() {
    _data = new List();
  }

  // Add new item
  add(item) {
    _data.add(item);
  }

  // ...
}
```

Somewhere in the main method, we create the instance of a class and add data to the collection:

```dart
import 'collection.dart';

var data = [1, 2, 3];

main() {
  var collection = new Collection();
  //
  data.forEach((item) {
    collection.add(item);
  });
}
```
Any chance that my collection will be initialized in this way in the future is high. So, it would be nice to have an initialization method in my Collection class to add a list of data. One of the solutions is to create a constructor with named parameters to manage our initialization. The code creates a new collection from the optional parameter value if specified, as shown in the following code:

```dart
library collection;

// Collection class
class Collection {
  // We save data here
  List _data;

  // Default constructor with optional [values] or [item].
  Collection({Iterable values:null, String item:null}) {
    if (item != null) {
      _data = new List();
      add(item);
    } else {
      _data = values != null ? new List.from(values) : new List();
    }
  }

  // Add new item
  add(item) {
    _data.add(item);
  }

  // ...
}
```

This solution has a right to live, but only if the number of parameters is small. Otherwise, a simple task initialization of variables results in very complicated code. I have specified two options for the named parameters with a really tangled logic. It is difficult to convey the meaning of what values means. A better way is to use named constructors, as shown in the following code:

```dart
library collection;

// Collection class
class Collection {
  // We save data here
  List _data;

  // Named constructor
  Collection({Iterable values, String item}) {
    if (item != null) {
      _data = new List();
      add(item);
    } else {
      _data = values != null ? new List.from(values) : new List();
    }
  }

  // Add new item
  add(item) {
    _data.add(item);
  }

  // ...
}
```
// Default constructor
Collection() {
    _data = new List();
}

// Create collection from [values]
Collection.fromList(Iterable values) {
    _data = values == null ?
        new List() :
        new List.from(values);
}

// Create collection from [item]
Collection.fromItem(String item) {
    _data = new List();
    if (item != null) {
        add(item);
    }
    // ...
}

The constructor is referred to as named because it has a readable and intuitive way of creating objects of a class. There are constructors named Collection.fromList and Collection.fromItem in our code. Our class may have a number of named constructors to do the simple task of class instantiation, depending on the type of parameters. However, bear in mind that any superclass named constructor is not inherited by a subclass.

The named constructors provide intuitive and safer construction operations.

A redirecting constructor
Constructors with optional parameters and named constructors help us to improve the usability and readability of our code. However, sometimes we need a little bit more. Suppose we want to add values of the map to our collection, we can do this by simply adding the Collection.fromMap named constructor as shown in the following code:

    //...
    // Create collection from [values]
    Collection.fromList(Iterable values) {
The preceding method is not suitable because the two named constructors have similar code. We can correct it by using a special form of a generative constructor (redirecting constructor), as shown in the following code:

```dart
// Create collection from values of [map]
Collection.fromMap(Map map) {
  _data = map == null ?
    new List() :
    new List.from(map.values);
}
```

The redirecting constructor calls another generative constructor and passes values of a map or an empty list. I like this approach as it is compact, less error-prone, and does not contain similar code anymore.

The redirecting constructor cannot have a body and initialization list.

### A private constructor

I want to mention a couple of things about the private constructor before we continue with our journey. A private constructor is a special generative constructor that prevents a class from being explicitly instantiated by its callers. It is usually used in the following cases:

- For a singleton class
- In the factory method
Object Creation

- When the utility class contains static methods only
- For a constant class

Let's define a Task class as follows:

```dart
class Task {
  int id;

  Task._();

  Task._internal();
}
```

In the preceding code, there are private named constructors, Task._ and Task._internal, in the Task class. Also, Dart does not allow you to create an instance of Task as no public constructors are available, as shown in the following screenshot:

```
1 main() {
  The class 'Task' does not have a default constructor
  Task task = new Task();
}
```

The private constructors are used to prevent creating instances of a class.

A factory constructor

A **factory** constructor can only create instances of a class or inherited classes. It is a static method of a class that has the same name as the class and is marked with the factory constructor.

```
The factory constructor cannot access the class members.
```

For More Information:
www.packtpub.com/application-development/mastering-dart
The factory method design pattern

Let's imagine that we are creating a framework and it's time to log the information about different operations of the class methods. The following simple Log class can work for us:

```dart
library log;

// Log information
abstract class Log {
  debug(String message);
  // ...
}
```

We can print the log information to the console with the ConsoleLog class as follows:

```dart
library log.console;
import 'log.dart';

// Write log to console
class ConsoleLog implements Log {
  debug(String message) {
    // ...
  }
  // ...
}
```

We can also save the log messages in a file with the FileLog class as follows:

```dart
library log.file;
import 'log.dart';

// Write log to file
class FileLog implements Log {
  debug(String message) {
    // ...
  }
  // ...
}
```
Now, we can use the `log` variable to print the debug information to the console in the `Adventure` class, as follows:

```dart
import 'log.dart';
import 'console_log.dart';

// Adventure class with log
class Adventure {
  static Log log = new ConsoleLog();

  walkMethod() {
    log.debug("entering log");
    // ...
  }
}
```

We created an instance of `ConsoleLog` and used it to print all our messages to the console. There are plenty of classes that will support logging, and we will include code similar to the preceding one in each of them. I can imagine what will happen when we decide to change `ConsoleLog` to `FileLog`. A simple operation can turn into a nightmare, because it might take a lot of time and resources to make the changes. At the same time, we must avoid altering our classes.

The solution is to use the factory constructor to replace the type of code with subclasses, as shown in the following code:

```dart
library log;

part 'factory_console_log.dart';

// Log information
abstract class Log {

  factory Log() {
    return new ConsoleLog();
  }

  debug(String message);
  // ...
}
```
The factory constructor of the Log class is a static method and every time it creates a new instance of ConsoleLog. The updated version of the Adventure class looks like the following code:

```dart
import 'factory_log.dart';

// Adventure class with log
class Adventure {
  static Log log = new Log();

  walkMethod() {
    log.debug("entering log");
    // ...
  }
}
```

Now, we are not referring to ConsoleLog as an implementation of Log, but only using the factory constructor. All the changes from ConsoleLog to FileLog will happen in one place, that is, inside the factory constructor. However, it would be nice to use different implementations that are appropriate in specific scenarios without altering the Log class as well. This can be done by adding a conditional statement in the factory constructor and instantiating different subclasses, as follows:

```dart
library log;

part 'factory_console_log.dart';
part 'factory_file_log.dart';

// Log information
abstract class Log {
  static bool useConsoleLog = false;

  factory Log() {
    return useConsoleLog ?
      new ConsoleLog() :
      new FileLog();
  }

  debug(String message);
  // ...
}
```

For More Information:  
www.packtpub.com/application-development/mastering-dart
Object Creation

The `useConsoleLog` static variable can be changed programatically at any point of time to give us a chance to change the logging direction. As a result, we don't change the `Log` class at all.

In our example, the factory constructor is an implementation of the factory method design pattern. It makes a design more customizable and only a little more complicated.

It is always better to use a factory constructor instead of a generative constructor in the following cases:

- It is not required to always return a new instance of a class
- It is required to instantiate any subtype of the return type
- It is essential to reduce the verbosity of creating parameterized type instances of a class

**The singleton design pattern**

If we want to keep unique information about a user somewhere in our imaginable framework, then we would have to create a `Configuration` class for that purpose, as shown in the following code:

```dart
library configuration;

// Class configuration
class Configuration {
  // It always keep our Configuration
  static final Configuration configuration = new Configuration._();

  // Database name
  String dbName;

  // Private default constructor
  Configuration._();
}
```

The `Configuration` class has a `dbName` variable to keep the database's name and probably a number of other properties and methods as well. It has a private default constructor, so the class cannot be instantiated from other classes. A static variable configuration is final and will be initialized only once. All looks good and the standards of implementing the `singleton` design pattern are followed.

For More Information:

We have only one instance of a `Configuration` class at a point of time and that's the main purpose of the singleton pattern. One disadvantage here is the time of initialization of the configuration variable. This only happens when our program starts. It is better to use the Lazy initialization when it calls the configuration variable for the first time. The following factory constructor comes in handy here:

```dart
library configuration;

// Class configuration
class Configuration {
  // It always keep our [Configuration]
  static Configuration _configuration;

  // Factory constructor
  factory Configuration() {
    if (_configuration == null) {
      _configuration = new Configuration._();
    }
    return _configuration;
  }

  // Database name
  String dbName;

  // Private default constructor
  Configuration._();
}
```

For now, when we refer to the `Configuration` class for the first time, Dart will call the factory constructor. It will check whether the private variable configuration was initialized before, create a new instance of the `Configuration` class if necessary, and only then return an instance of our class. The following are the changes in the framework of the code:

```dart
import 'factory_configuration.dart';

main() {
  // Access to database name
  new Configuration().dbName = 'Oracle';
  // ...
  print('Database name is ${new Configuration().dbName}');
}
```

For More Information:
www.packtpub.com/application-development/mastering-dart
We always get the same instance of a Configuration class when we call the factory method in this solution. Factory constructors can be widely used in the implementation of the flyweight pattern and object pooling.

### A constant constructor

Let's assume we have the following Request class in our imaginary framework:

```dart
library request;

// Request class
class Request {
  static const int AWAIT = 0;
  static const int IN_PROGRESS = 1;
  static const int SUCCESS = 2;
  static const int FAULT = 3;

  // Result of request
  int result = AWAIT;

  // Send request with optional [status]
  void send({status:IN_PROGRESS}) {
    // ...
  }
}
```

The `result` variable keeps the status of the last request as an integer value. The Result class has constants to keep all the possible values of the status in one place so that we can always refer to them. This makes the code better in readability and safer too. This technique is called the enumerated pattern and is widely used. However, it has a problem when it comes to safety as any other integer value could be assigned to the `result` variable. This problem makes a class very fragile. Enumerated types would help us to solve this problem, but unfortunately they do not exist in Dart. The solution to this is that we create an enumerated type ourselves with the help of constant constructors as it can be used to create a compile-time constant.

We can create an abstract Enum class with the entered parameter, as follows:

```dart
library enumerated_type;

abstract class Enum<T> {
  // The value
  final T value;

  // Get the value
  T getValue();
}
```

For More Information:

www.packtpub.com/application-development/mastering-dart
// Create new instance of [T] with [value]
const Enum(this.value);

// Print out enum info
String toString() {
  return "${runtimeType.toString()}." +
      "${value == null ? 'null' : value.toString()}";
}

I intentionally used a generic class in the preceding code as I don't know which
type of enumeration we will create. For example, to create an enumerated type
of RequestStatus based on the integer values, we can create a concrete class
as follows:

```dart
import 'enum.dart';

// Enumerated type Status of Request
class RequestStatus<int> extends Enum {
  static const RequestStatus AWAIT = const RequestStatus(0);
  static const RequestStatus IN_PROGRESS = const RequestStatus(1);
  static const RequestStatus SUCCESS = const RequestStatus(2);
  static const RequestStatus FAULT = const RequestStatus(3);

  const RequestStatus(int value) : super(value);
}
```

The RequestStatus class extends the Enum class and defines the request statuses as
static constant members of the class. To instantiate the RequestStatus class object,
we use `const` instead of the `new` instantiation expression.

![A constant constructor creates compile-time immutable instances of a class.]

Let's go back to the Request class and modify it with RequestStatus, as shown in
the following code:

```dart
library request;

import 'request_status.dart';

// Request class with enum
class Request {
```
// Result of request
var result = RequestStatus.AWAIT;

// Send request with optional [status]
void send({status:RequestStatus.IN_PROGRESS}) {
  // ...
}

In the preceding code, we used the enumerated type across the whole class. Finally, here is main method in which we use the Request class:

import 'request_with_enum.dart';
import 'request_status.dart';

void main() {
  Request request = new Request();
  // ...
  request.send(status:RequestStatus.SUCCESS);
  // ...
  RequestStatus status = request.result;
  //
  switch (status) {
    case RequestStatus.AWAIT:
      print('Result is $status');
      // ...
      break;
  }
}

As you can see, the compile-time constants have a wide variety of uses such as default values of variables and constants, default values in method signatures, switch cases, annotations, and enumerators. Moreover, the code uses them to have a better performance and translates them into optimized JavaScript code.

Use cases of the constant constructor have the following restrictions:

- A constant constructor doesn't have a body to prevent any changes of the class state
- All the variables in a class that have a constant constructor must be final as their binding is fixed upon initialization
- The variables from the initialization list of a constant constructor must be initialized with compile-time constants

For More Information:  
www.packtpub.com/application-development/mastering-dart
Initializing variables

Variables reflect the state of a class instance. There are two types of variables, namely, class and instance variables. Class variables are static in Dart. The static variables of a class share information between all instances of the same class. A class provides instance variables when each instance of a class should maintain information separately from others. As we mentioned, the main purpose of a constructor is to initialize the instance of a class in a safe manner. In other words, we initialize instance variables. Let’s discuss when and how we should initialize them.

Uninitialized variables in Dart have the value `null` so we should initialize them before using them. The initialization of variables may happen in several places. They are as follows:

- We can assign any value to a variable at the place of declaration
- A variable can be initialized in the body of a constructor
- A variable can be initialized over a constructor parameter
- Initialization can happen in the initialization list of a constructor

Where is the best place to initialize the variables? Noninitialized variables generate null reference runtime exceptions when we try to use them, as shown in the following code:

```dart
class First {
  bool isActive;

  doSomething() {
    if (isActive) {
      // ...
    }
  }
}

void main() {
  First first = new First();
  first.doSomething();
}
```

For More Information:
www.packtpub.com/application-development/mastering-dart
Object Creation

The runtime exception will terminate the program execution and display the following error because the isActive variable of the First class was not initialized:

Unhandled exception:
type 'Null' is not a subtype of type 'bool' of 'boolean expression'.
#0 First.doSomething (file:///… / no_initialized.dart:5:9)
#1 main (file:///…/ no_initialized.dart:13:19)
...

The variable must be initialized during the declaration if we are not planning to do it in other places.

Now, let's move ahead. The First and Second classes are similar to each other with only a few differences, as shown in the following code:

```dart
class First {
  bool isActive;

  First(bool isActive) {
    this.isActive = isActive;
  }
}

class Second {
  bool isActive;

  Second(this.isActive);
}
```

In the First class, we initialize a variable in the body of a constructor; otherwise, the Second class initializes a variable via a constructor parameter. Which one is right? The desire to use the Second class is obvious because it is compact.

It is always preferred to use the compact code to initialize the variables via constructor parameters than in a body of the constructor.

A variable marked final is a read-only variable that must be initialized during the instantiation of a class. This means all the final variables must be initialized:

- At the place of declaration
- Over a constructor parameter
- In the initialization list of a constructor

For More Information:
www.packtpub.com/application-development/mastering-dart
In the following code, we initialize the `isActive` variable at the place of declaration:

```dart
class First {
  final bool isActive = false;

  First();
}
```

In the `Second` class, we initialize the `isActive` variable via a parameter of the constructor, as follows:

```dart
class Second {
  final bool isActive;

  Second(this.isActive);
}
```

If the `isActive` final variable is indirectly dependent on the parameter of the constructor, we use the initializer list, as shown in the following code, as Dart does not allow us to initialize the final variable in a body of the constructor:

```dart
class Third {
  final bool isActive;

  Third(value) :
    this.isActive = value != null;
}
```

The last place to initialize the final variables is in the constructor initializer list.

### Syntactic sugar

We talked a lot about the usability of the code, but I could not resist the desire to mention a couple of things about **syntactic sugar**—a syntax that is designed to make a code easier to read or express.

---

For More Information:

**Object Creation**

**Method call**

Dart is an object-oriented programming language by definition. However, sometimes we need a piece of the functional language to be present in our estate. To help us with this, Dart has an interesting feature that may change the behavior of any class instance like a function, as shown in the following code:

```dart
import 'dart:async';

class Request {
  send() {
    print("Request sent");
  }
}

main() {
  Request request = new Request();
  Duration duration = new Duration(milliseconds: 1000);
  Timer timer = new Timer(duration, (Timer timer) {
    request.send();
  });
}
```

We have a `timer` function that invokes a callback function and sends a request to the server periodically to help organize pull requests, as follows:

```
Request sent
Request sent
...
```

The `call` method added to the class helps Dart to emulate instances of the `Request` class as functions, as shown in the following code:

```dart
import 'dart:async';

class Request {
  send() {
    print("Request sent");
  }
  
call(Timer timer) {
    send();
  }
}

main() {
  Duration duration = new Duration(milliseconds: 1000);
  Timer timer = new Timer.periodic(duration, new Request());
}
```

For More Information:

www.packtpub.com/application-development/mastering-dart
So now we invoke the `send` method from the `call` method of the `Request` class. The result of the execution is similar to the result from the preceding example:

```
Request sent
Request sent
...
```

The `call` method allows Dart to execute a class instance as a function.

## Cascade method invocation

Dart has quite a large number of innovations to create an application comfortably, but I will mention the one that can help us to write compact code. It is a cascade method invocation. Let’s take a look at the ordinary `SomeClass` class in the following code:

```dart
library some_class;

class SomeClass {
  String name;
  int id;
}
```

Also, we can see the absolutely ordinary object creation:

```dart
import 'some_class.dart';

void main() {
  SomeClass some = new SomeClass();
  some.name = 'John';
  some.id = 1;
}
```

We created an instance of a class and initialized the instance variables. The preceding code is very simple. A more elegant and compact version of code uses the cascade method invocation, as follows:

```dart
import 'some_class.dart';

void main() {
  SomeClass some = new SomeClass()
    ..name = 'John'
    ..id = 1;
}
```
Object Creation

In the first line of the main method, we create an instance of the SomeClass class. At the same time, Dart creates a scope of the some variable and invokes all methods located in that scope. The result would be similar to the one that was in the previous code snippet.

Use the cascade method invocation to make the code less verbose to do multiple operations on the members of an object.

Summary

In this chapter, you learned that a constructor is a method that is invoked to create an object from a class. There are generative and factory constructors. The main purpose of a generative constructor is to initialize the instance of a class in a safe manner. The constructor can have required or optional parameters. The optional parameters enable the supply of arguments for only a few parameters from the list of optional parameters.

A constructor can be named. The named constructors provide intuitive and safer construction operations, because named constructors have similar code and some of them can be translated into redirecting constructors. A redirecting constructor calls another constructor that makes the code compact.

Dart supports private constructors. If a class has only private constructors, Dart cannot create an instance of a class. Private constructors are usually used in classes that contain static members, which are only useful in combination with the factory constructor. A factory constructor usually implements the popular factory method and singleton design patterns.

No language can exist without constant variables or constants. Constants play a significant role in programming with Dart and constant constructors can create compile-time constants as instances of a class.

The initialization of variables can happen in several places. Some variables must be initialized during declaration; others can be initialized via constructor parameters. Final variables must be initialized during the instantiation of class. This can happen during a declaration, via constructor parameters or in the constructor initializer list.

This syntax in Dart very often provides a form of syntactic sugar. One of those places is a method call that allows you to execute a class instance as a function. Another one is the cascade method invocation that makes the code less verbose.

In the next chapter, we will discuss advanced technologies to organize asynchronous code execution and learn the best practices to use Futures, Zones, and Isolates in different cases.
Where to buy this book

You can buy Mastering Dart from the Packt Publishing website: 

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.