Day 1
Introduction To Dart

MARLON I. TAYAG

What is Dart?

Dart is an object-oriented language with C-style syntax which can optionally trans compile into JavaScript. It supports a varied range of programming aids like interfaces, classes, collections, generics, and optional typing.

History of Dart

Dart was unveiled at the GOTO conference in Aarhus, Denmark, October 10–12, 2011. The project was founded by Lars Bak and Kasper Lund. Dart 1.0 was released on November 14th, 2013. In August 2018, Dart 2.0 was released, with language changes including a sound type system.

Dart 2.4.0

Goal of Dart

Help developers write complex, high fidelity client apps for the modern mobile and web applications.

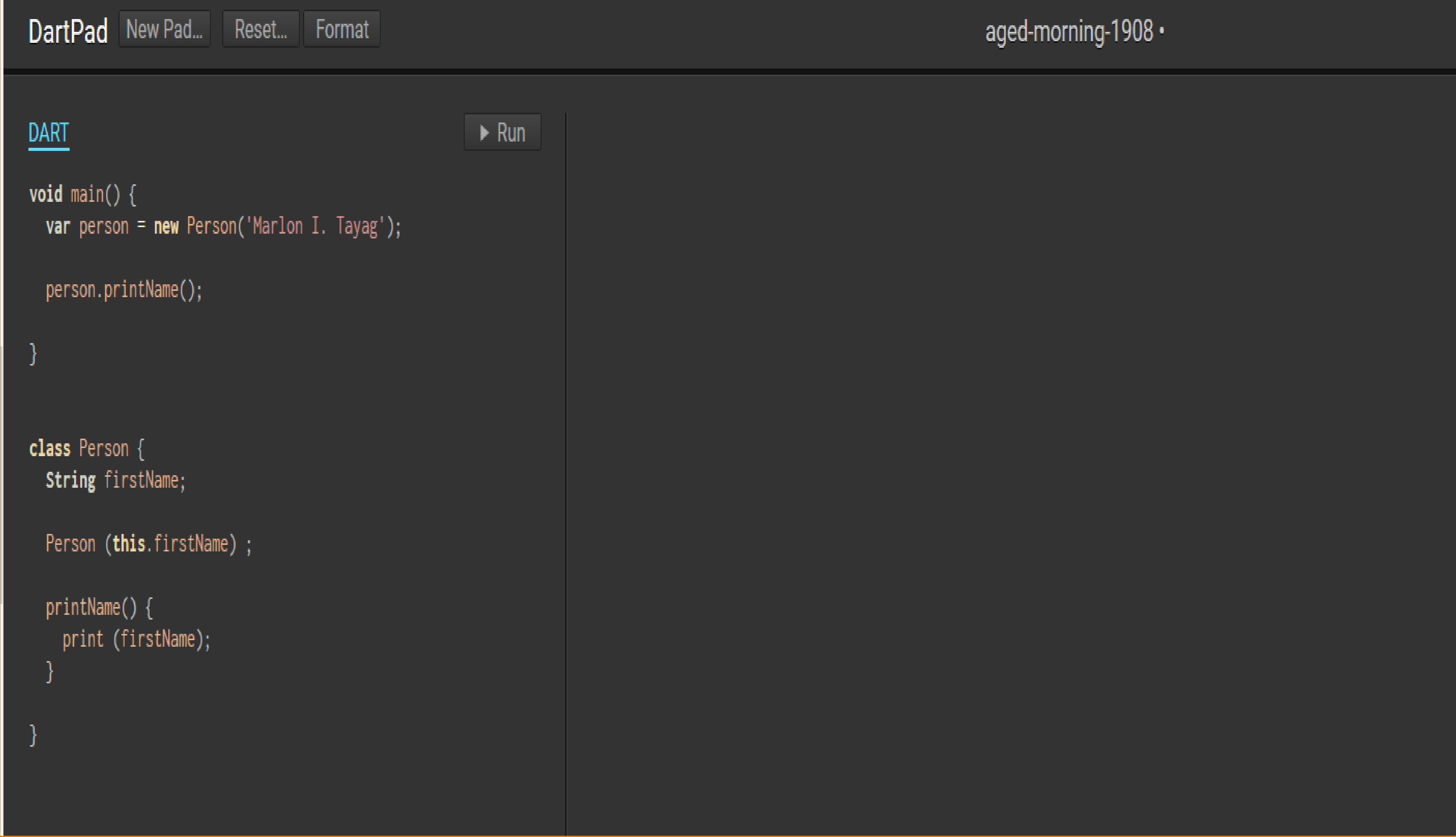
Native Mobile Apps

Google has introduced Flutter for native mobile app development on both Android and iOS. Flutter is a mobile app SDK, complete with framework, widgets, and tools, that gives developers a way to build and deploy mobile apps, written in Dart.

DartPad Editor

The Dart team created DartPad at the start of 2015, to provide an easier way to start using Dart. It is a fully online editor from which users can experiment with Dart application programming interfaces (APIs), and run Dart code. It provides syntax highlighting, code analysis, code completion, documentation, and HTML and CSS editing

https://dartpad.dartlang.org/



```
void main() {
  var person = new Person('Marlon I. Tayag');

  person.printName();

}


class Person {
  String firstName;

  Person (this.firstName) ;

  printName() {
    print (firstName);
  }

}
```

Language Syntax

```
main() {
  var d = "Dart";
  String w = "World";
  print("Hello ${d} ${w}");
}
```

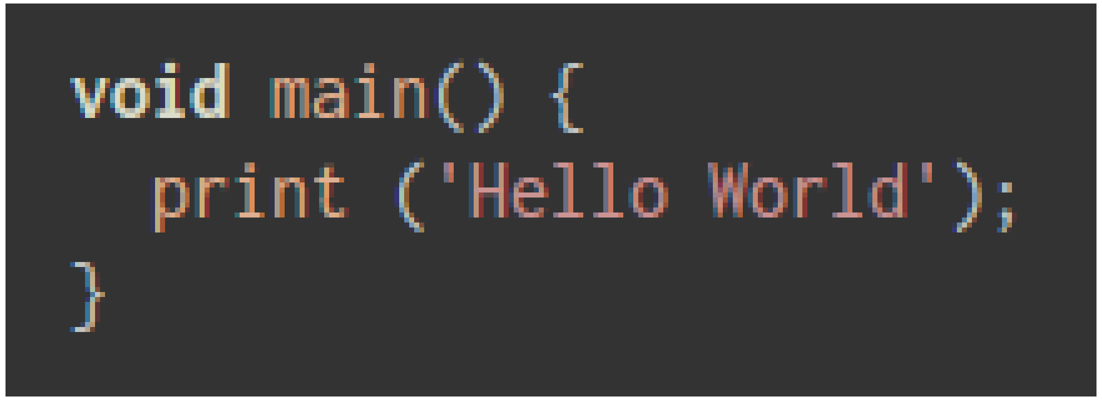
Single entry-point function main() executes when script is fully loaded

Optional typing (no type specified)

Type annotation (String type specified)

Uses string interpolation to output "Hello Dart World" to browser console or stdout

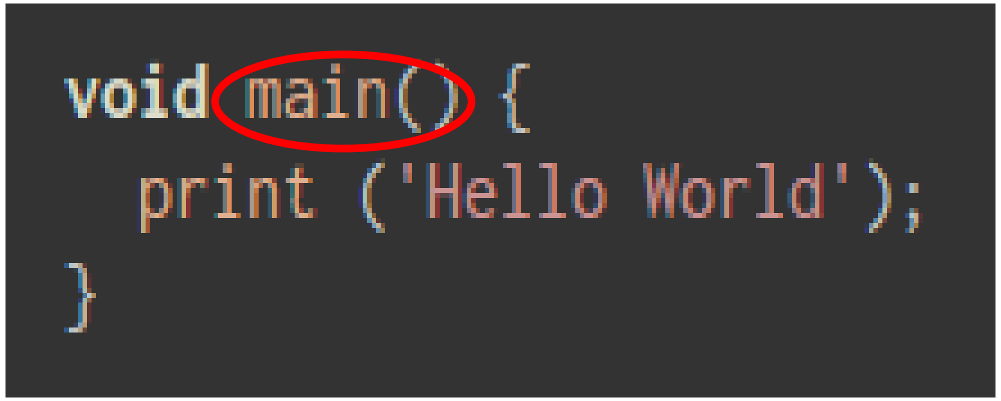
Hello Word



```
void main() {
  print ('Hello World');
}
```

main()

This line declares a function called main(). In short, they are a block of statements that the computer executes when the function is “called.” They are like a page in an instruction manual that can be turned to anytime. main() is always the entry point of a Dart program—the place where the program begins execution. main() is automatically called (executed).



```
void main() {
  print ('Hello World');
}
```

Identifiers in Dart

Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are :

Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.

1. Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
2. Identifiers cannot be keywords.
3. They must be unique.
4. Identifiers are case-sensitive.
5. Identifiers cannot contain spaces.

Valid identifiers	Invalid identifiers
firstName	Var
first_name	first name
num1	first-name
$result	1number

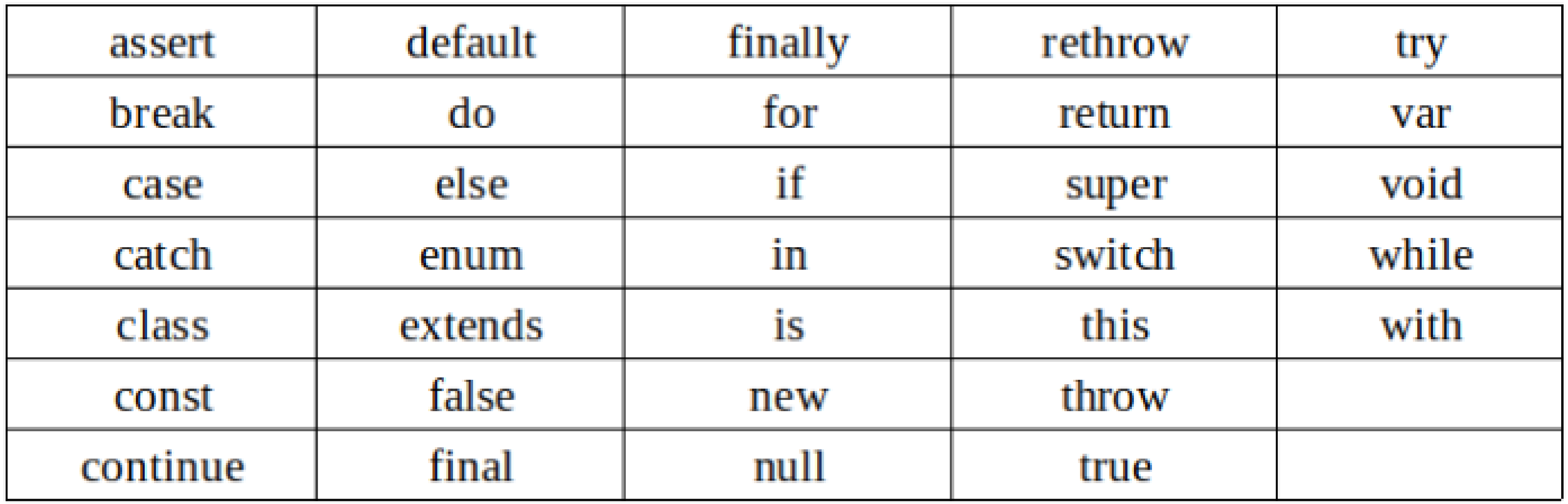
Keywords in Dart

Keyword is a word that is reserved by a program because the word has a special meaning. Keywords can be commands or parameters. Every programming language has a set of keywords that cannot be used as variable names. Keywords are sometimes called *reserved names* :

1. Reserved Words
2. Contextual Keywords
3. Built-in Identifiers

Reserved Words

Following are the reserved words in dart. These reserved words cannot be used as identifiers.



assert	default	finally	rethrow	try
break	do	for	return	var
case	else	if	super	void
catch	enum	in	switch	while
class	extends	is	this	with
const	false	new	throw	
continue	final	null	true	

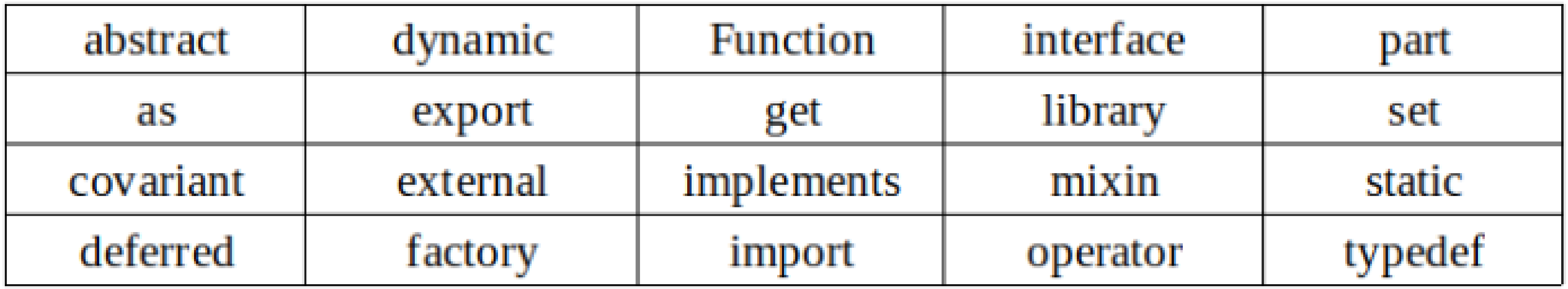
Contextual Keywords

These keywords have meaning only in specific places. In other places they are valid identifiers.

async	show
hide	sync
on	

Built-in Identifiers

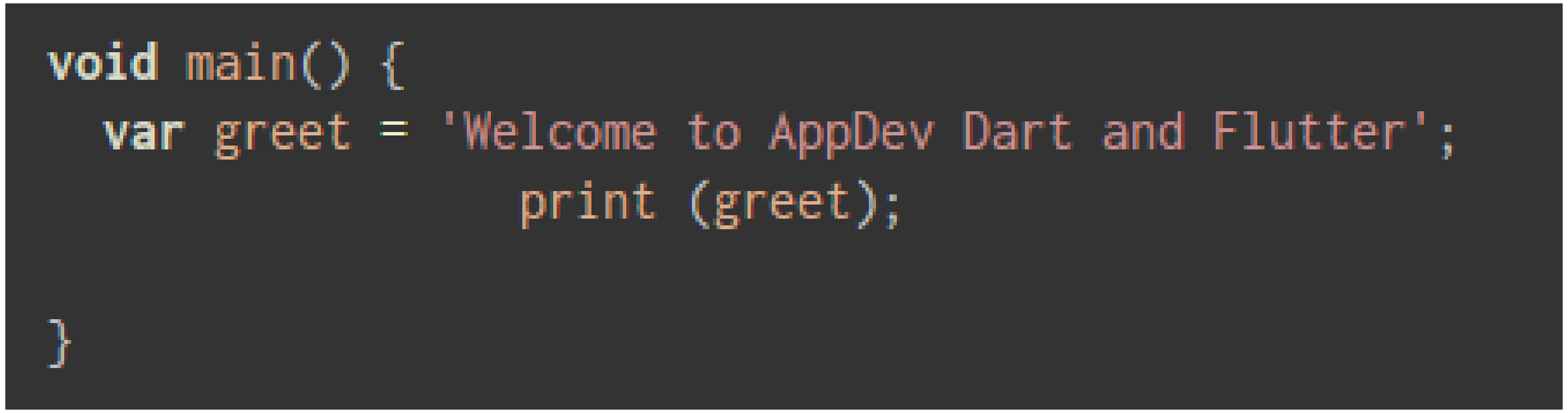
These keywords are valid identifiers in most places, but they can’t be used as class or type names, or as import prefixes.



abstract	dynamic	Function	interface	part
as	export	get	library	set
covariant	external	implements	mixin	static
deferred	factory	import	operator	typedef

Whitespace and Line Breaks

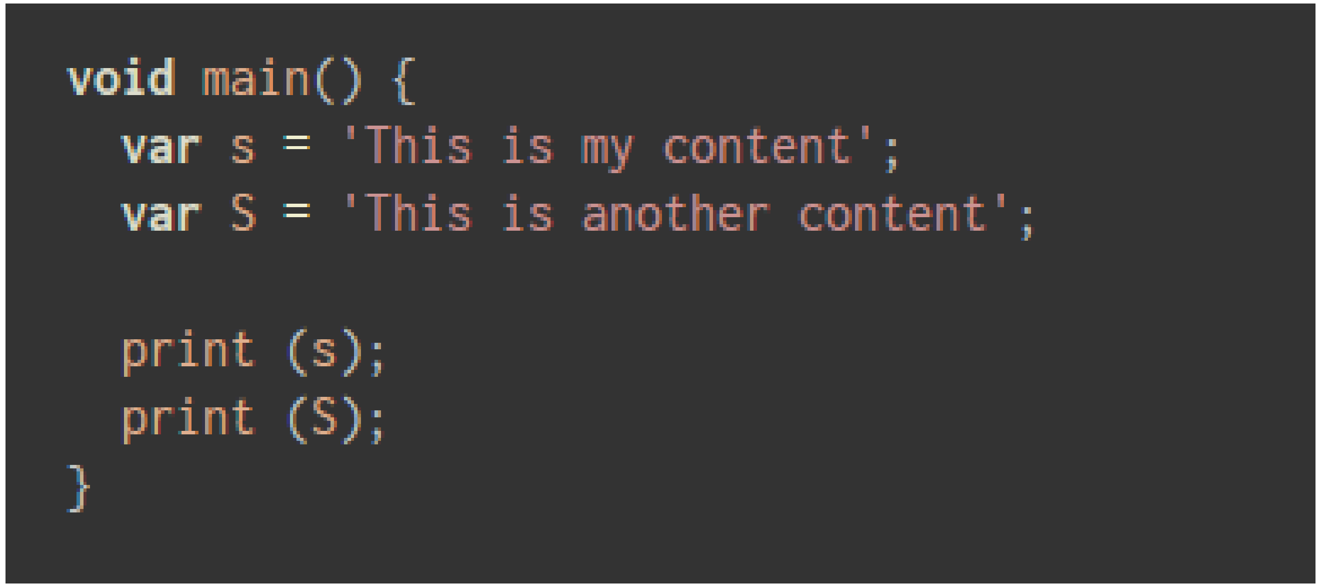
Dart ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand. But it is better to follow coding and alignment rules



```
void main() {
  var greet = 'Welcome to AppDev Dart and Flutter';
                print (greet);

}
```

Dart is Case-sensitive

Dart is case-sensitive. This means that Dart differentiates between uppercase and lowercase characters.

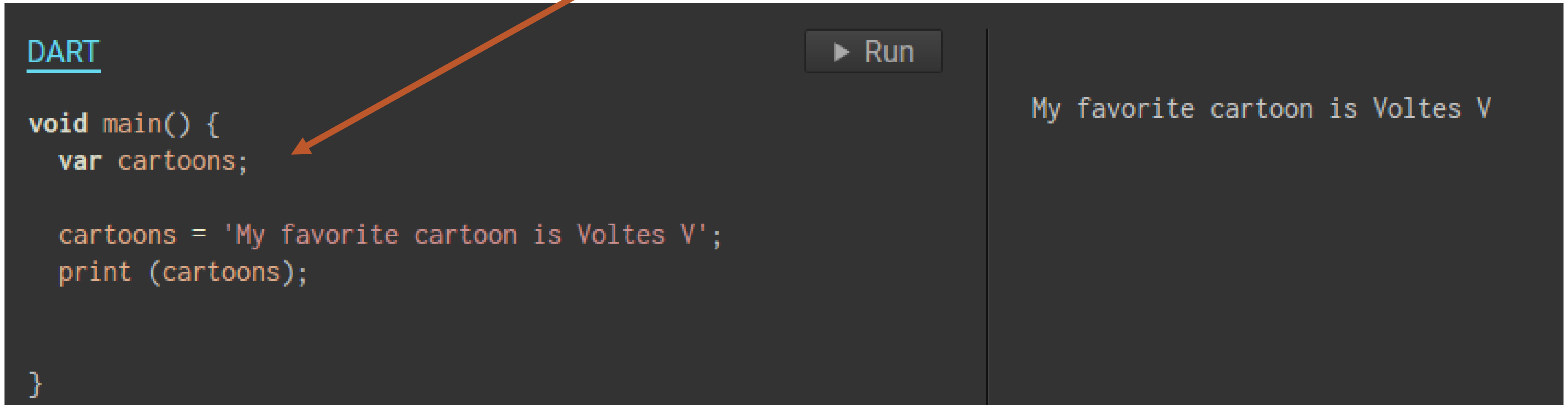


```
void main() {
  var s = 'This is my content';
  var S = 'This is another content';

  print (s);
  print (S);
}
```

Statements end with a Semicolon

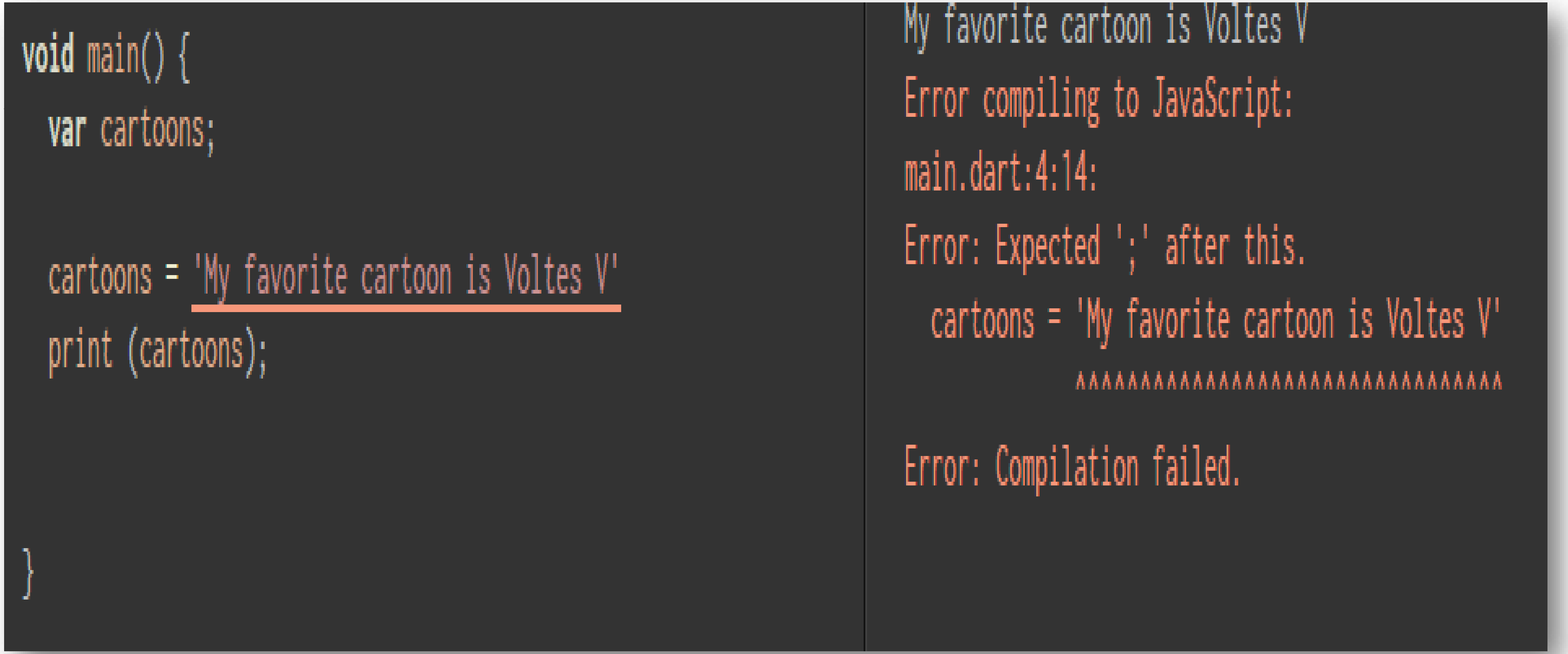
Each line of instruction is called a statement. Each dart statement must end with a semicolon (;). A single line can contain multiple statements. However, these statements must be separated by a semicolon.

semicolon



```
void main() {
  var cartoons;

  cartoons = 'My favorite cartoon is Voltes V';
  print (cartoons);


}
```

```
My favorite cartoon is Voltes V
```



```
void main() {
  var cartoons;

  cartoons = 'My favorite cartoon is Voltes V'
  print (cartoons);


}
```

```
My favorite cartoon is Voltes V
Error compiling to JavaScript:
main.dart:4:14:
Error: Expected ';' after this.
  cartoons = 'My favorite cartoon is Voltes V'
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Error: Compilation failed.
```

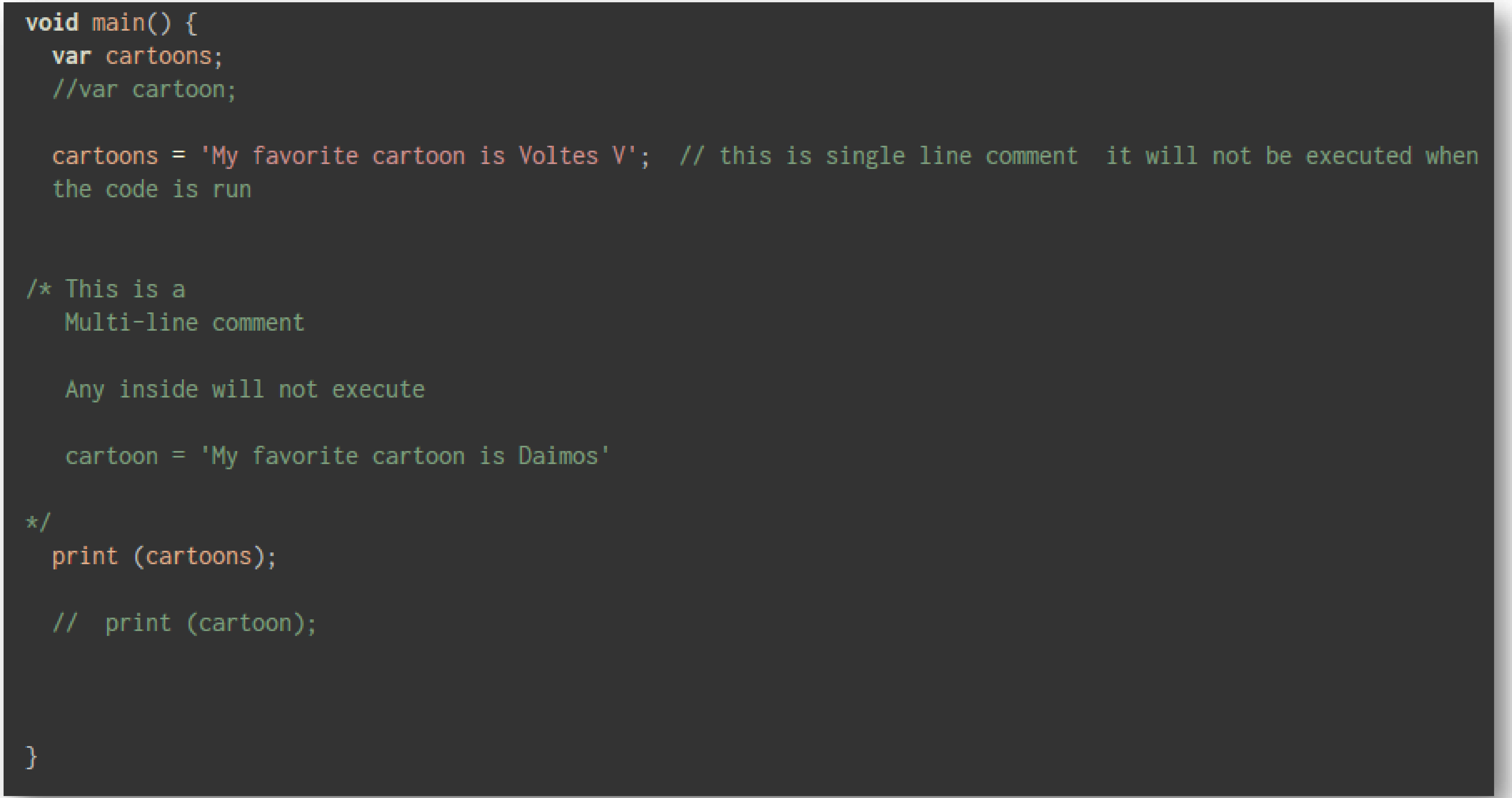
Comments in Dart

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like author of the code, hints about a function/ construct etc. Comments are ignored by the compiler.

Dart supports the following types of comments :

Single-line comments (//) – Any text between a "//" and the end of a line is treated as a comment

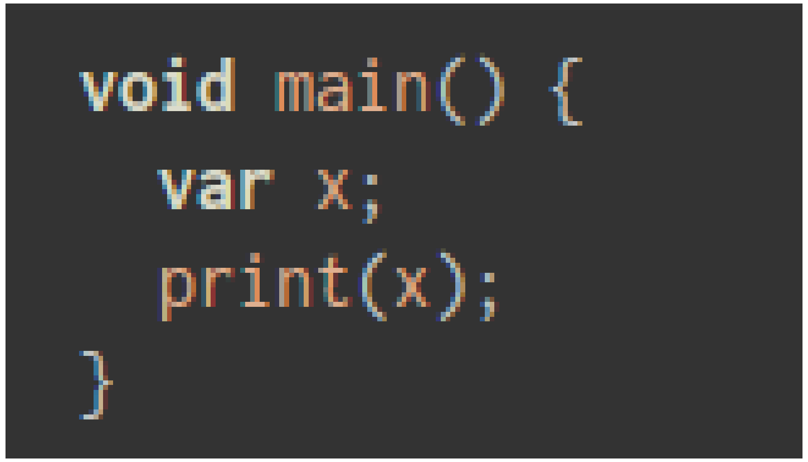
Multi-line comments (/* */) – These comments may span multiple lines.



```
void main() {
  var cartoons;
  //var cartoon;

  cartoons = 'My favorite cartoon is Voltes V';  // this is single line comment  it will not be executed when the code is run



/* This is a
   Multi-line comment

   Any inside will not execute

   cartoon = 'My favorite cartoon is Daimos'

*/
  print (cartoons);

  //  print (cartoon);



}
```

Variables

Variables are a basic building block of most modern programs. In short, a variable is an alias for a computer memory location, at which something of interest will be stored. Whatever is stored there is said to be the variable's value.



```
void main() {
  var x;
  print(x);
}
```

Variables

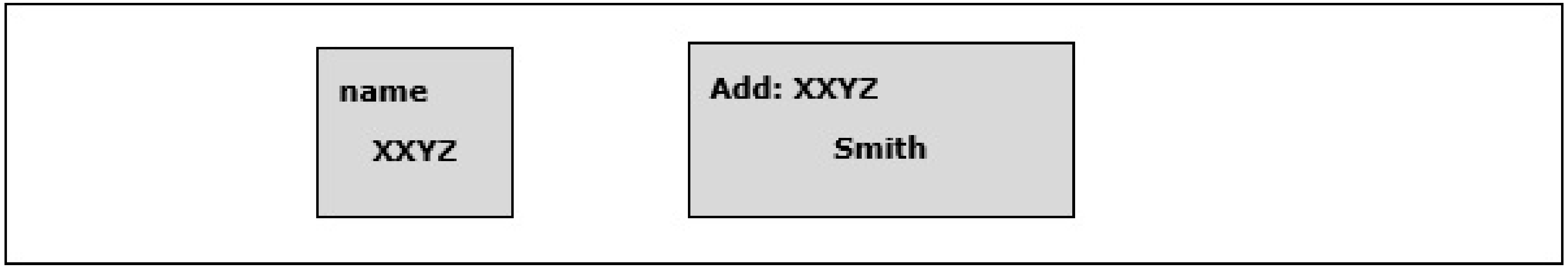
A variable is "a named space in the memory" that stores values. In other words, it acts a container for values in a program. Variable names are called identifiers. Following are the naming rules for an identifier:

- Identifiers cannot be keywords.
- Identifiers can contain alphabets and numbers.
- Identifiers cannot contain spaces and special characters, except the underscore (_) and the dollar ($) sign.
- Variable names cannot begin with a number.

Type Syntax

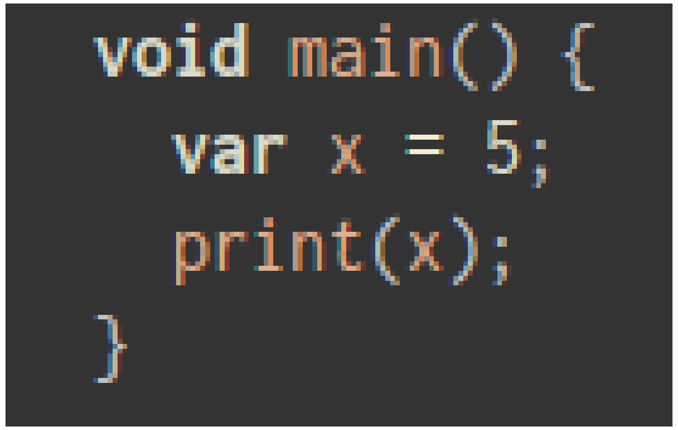
A variable must be declared before it is used. Dart uses the var keyword to achieve the same. The syntax for declaring a variable is as given below:

```
var name = 'Smith';
```



Initialize Variable

As null is not a very useful value for your variable, try initializing it with a numeric value, like so:



```
void main() {
  var x = 5;
  print(x);
}
```

Dart Programming - Data Types

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

The Dart language supports the following types:

- Numbers
- Strings
- Booleans
- Lists
- Maps

Numbers

Numbers in Dart are used to represent numeric literals. The Number Dart come in two flavours:

Integer − Integer values represent non-fractional values, i.e., numeric values without a decimal point. For example, the value "10" is an integer. Integer literals are represented using the **int** keyword.

Double − Dart also supports fractional numeric values i.e. values with decimal points. The Double data type in Dart represents a 64-bit (double-precision) floating-point number. For example, the value "10.10". The keyword **double** is used to represent floating point literals.

Strings

Strings represent a sequence of characters. For instance, if you were to store some data like name, address etc. the string data type should be used.

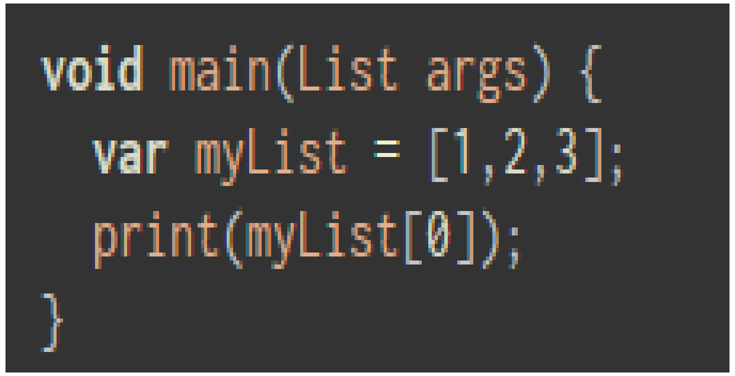
The keyword **String** is used to represent string literals. String values are embedded in either single or double quotes.

Boolean

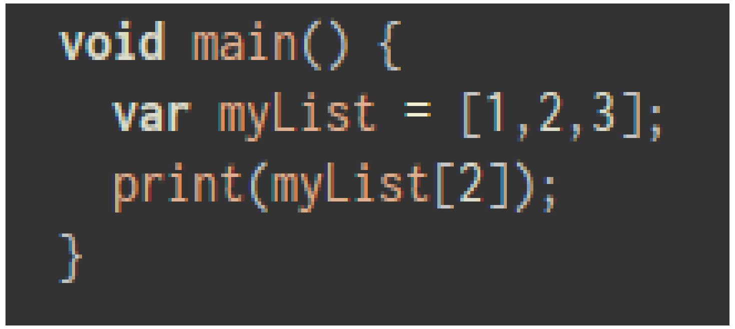
The Boolean data type represents Boolean values true and false.
Dart uses the **bool** keyword to represent a Boolean value.

List and Map

The data types list and map are used to represent a collection of objects. A **List** is an ordered group of objects. The List data type in Dart is synonymous to the concept of an array in other programming languages. The **Map** data type represents a set of values as key-value pairs. The **dart: core** library enables creation and manipulation of these collections through the predefined List and Map classes respectively.



```
void main(List args) {
  var myList = [1,2,3];
  print(myList[0]);
}
```

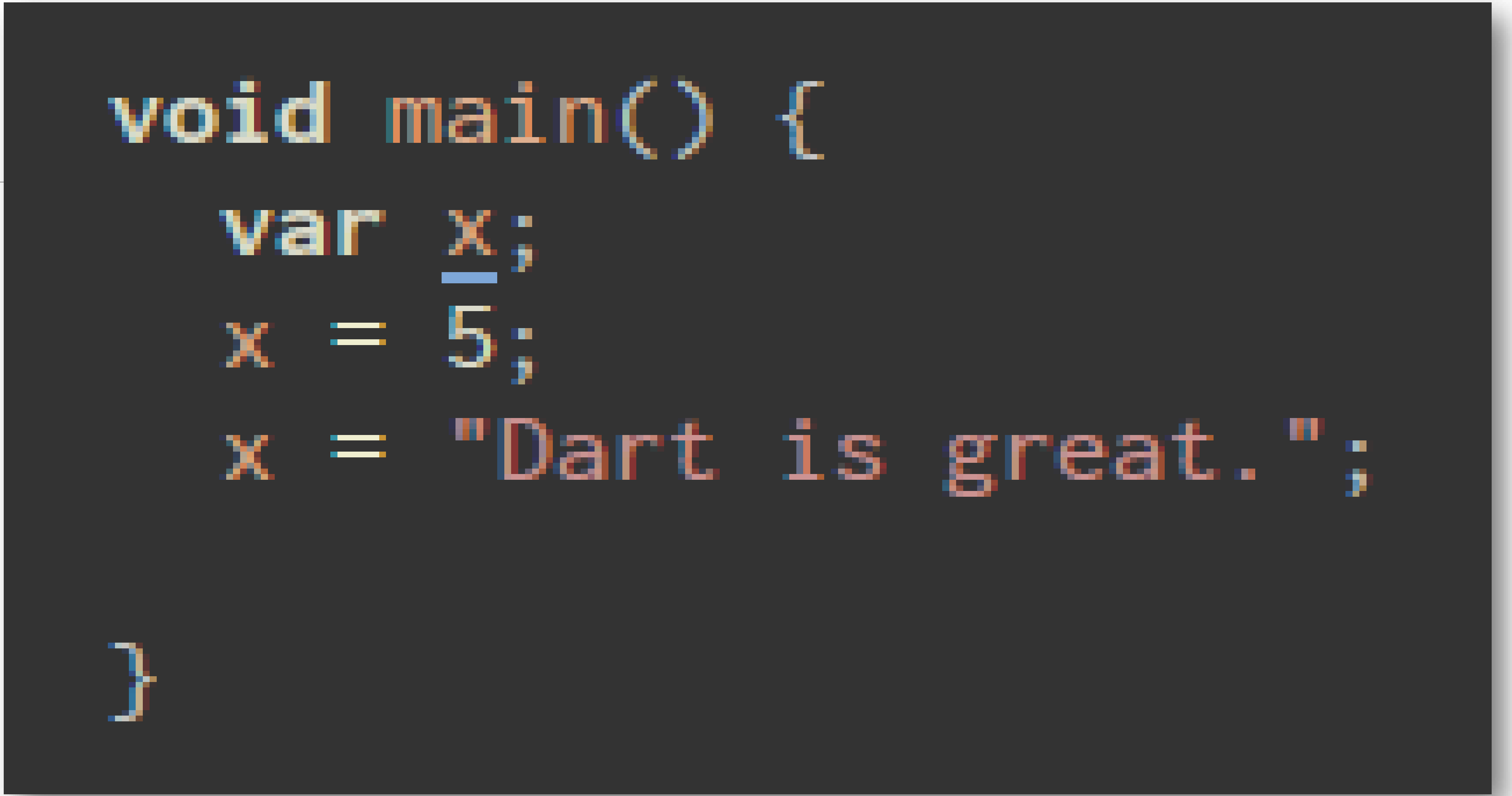


```
void main() {
  var myList = [1,2,3];
  print(myList[2]);
}
```

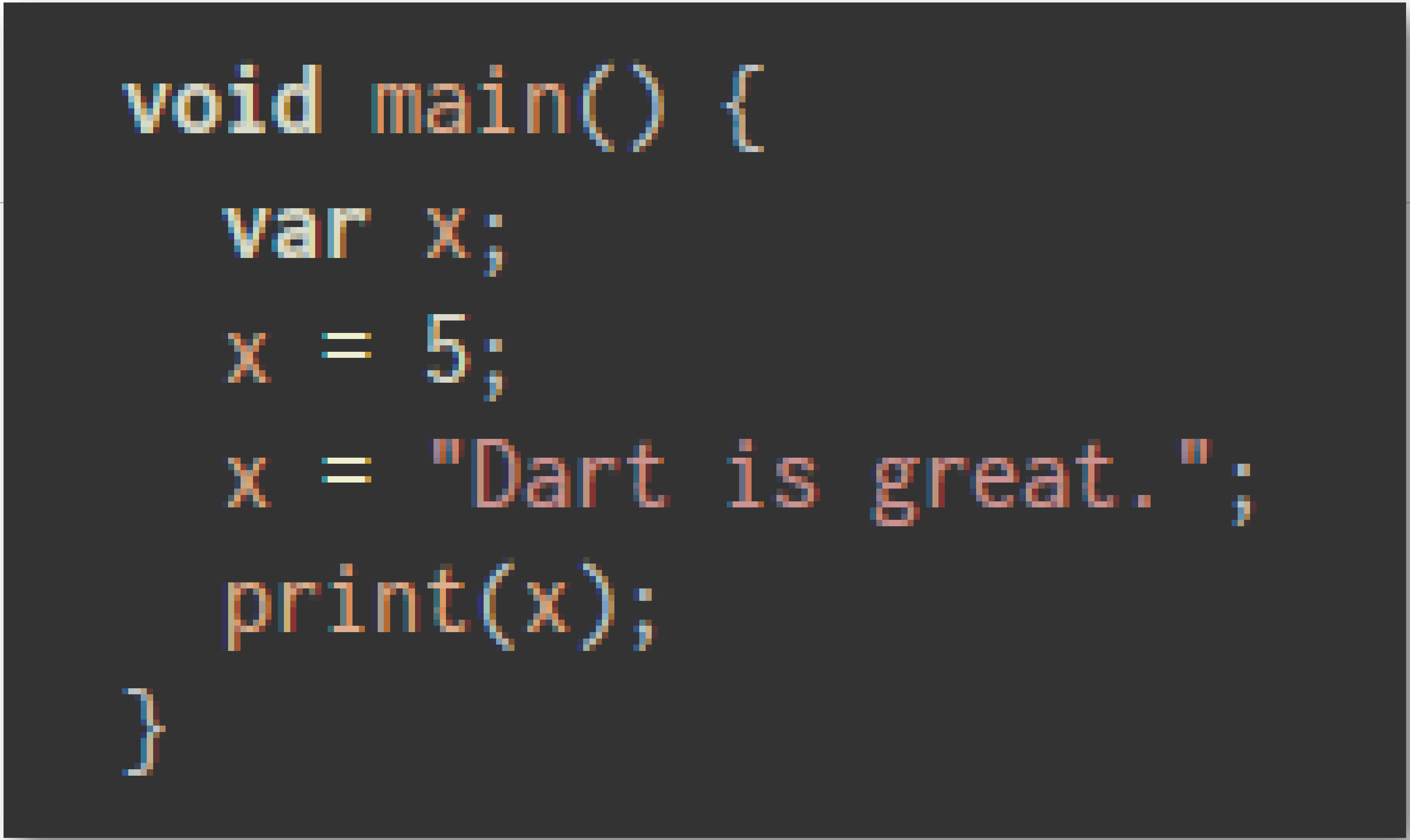
Inferred Types

```
var x;
```

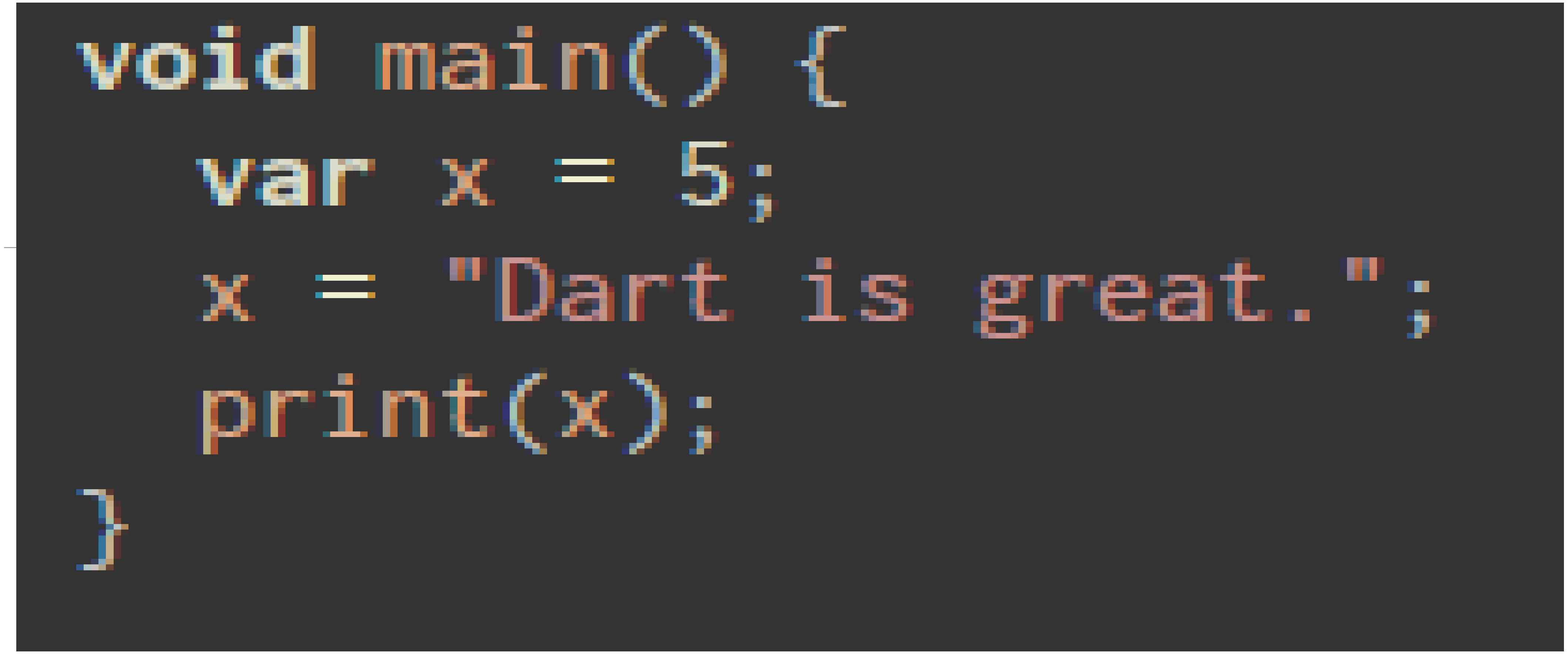
When you declare a variable using **var**, Dart tries to infer the variable's type. If you don't initialize the new variable with a value in the same statement as the declaration, Dart has nothing on which to base an inference, so the variable is considered to be of type dynamic. As a **dynamic variable**, x is able to accept values of any valid type.



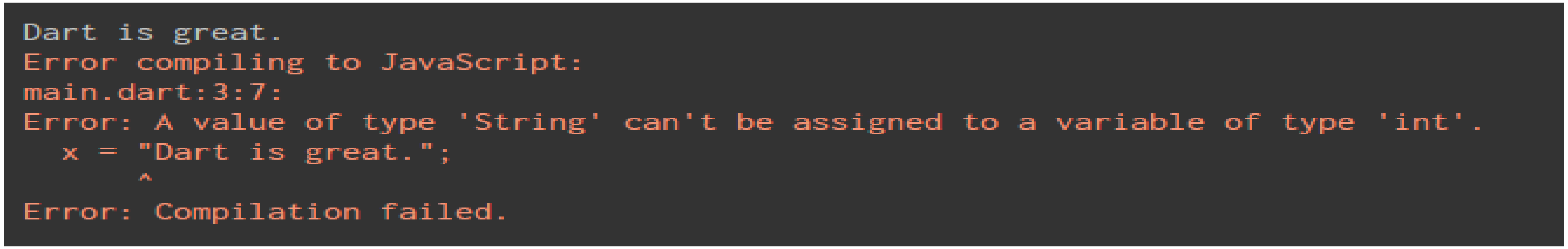
```
void main() {
  var x;
  x = 5;
  x = "Dart is great.";

}
```



```
void main() {
  var x;
  x = 5;
  x = "Dart is great.";
  print(x);
}
```



```
void main() {
  var x = 5;
  x = "Dart is great.";
  print(x);
}
```



```
Dart is great.
Error compiling to JavaScript:
main.dart:3:7:
Error: A value of type 'String' can't be assigned to a variable of type 'int'.
  x = "Dart is great.";
      ^
Error: Compilation failed.
```

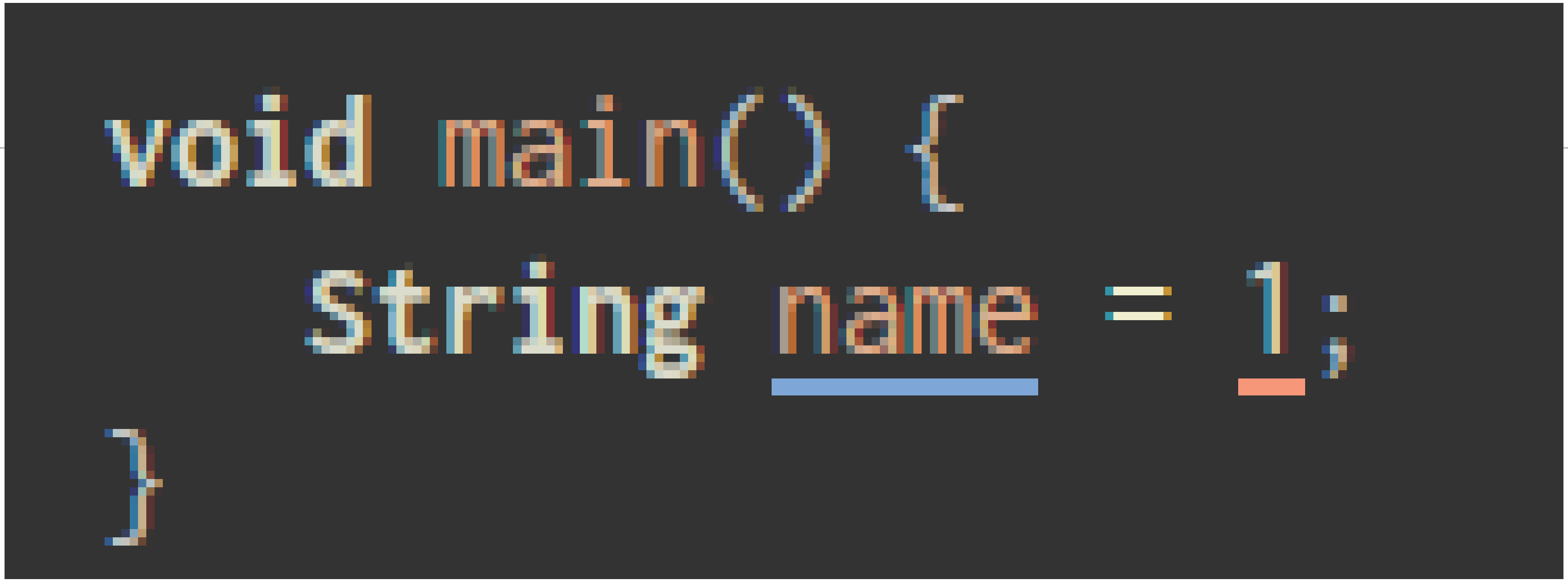
Type-Checking

Dart supports **type-checking** by prefixing the variable name with the data type. Type-checking ensures that a variable holds only data specific to a data type. The syntax for the same is given below:

```
void main() {
   String name = ‘Skee’;
   int num = 10;
}
```

Errors Stop Execution

Note that if you run the code with mismatched types, execution will fail. You should always eliminate all reported errors before attempting to run your code.

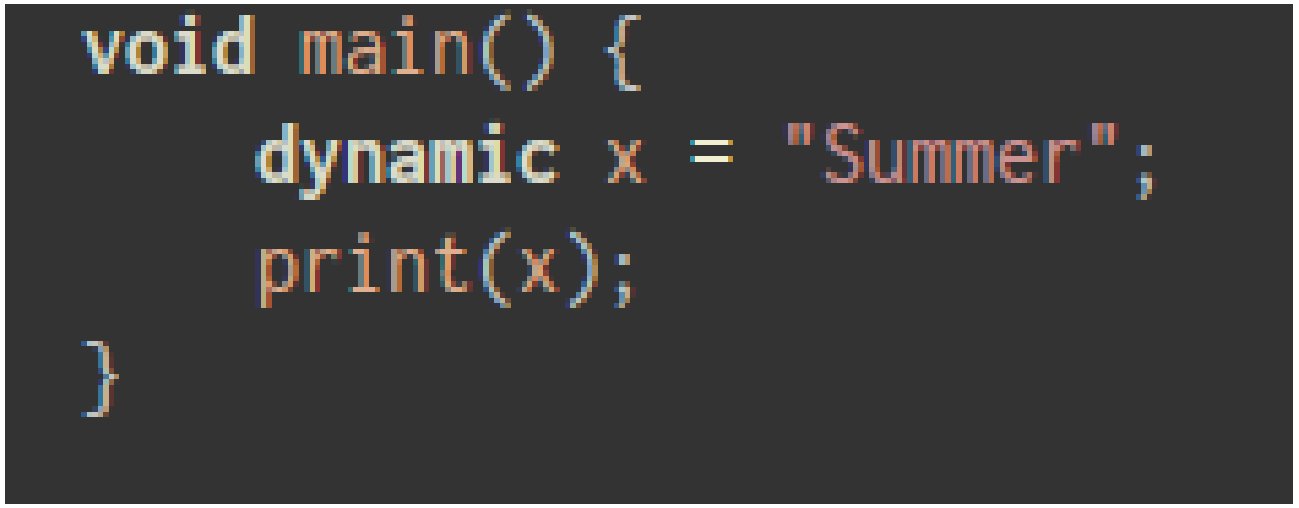


```
void main() {
    String name = 1;
}
```

What is the problem?

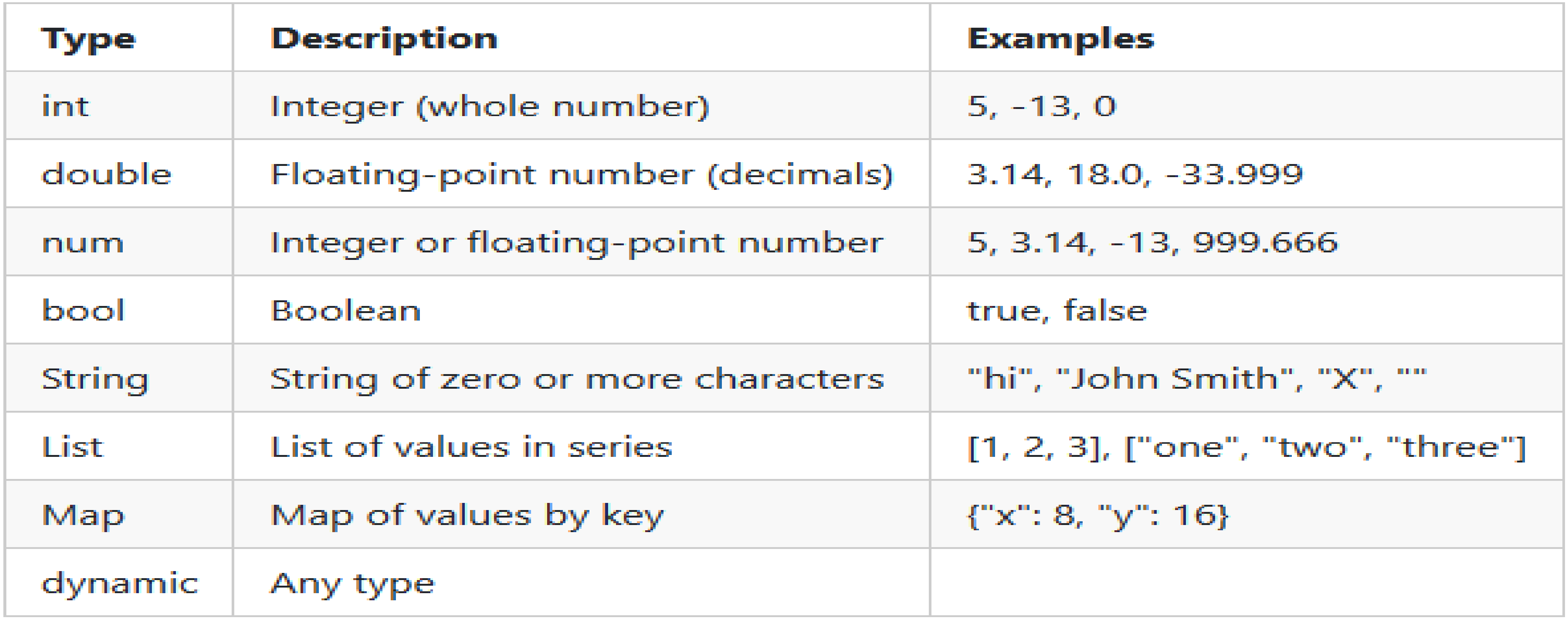
The Dynamic Keyword

Variables declared without a static type are implicitly declared as dynamic. Variables can be also declared using the dynamic keyword in place of the var keyword.



```
void main() {
    dynamic x = "Summer";
    print(x);
}
```

Fundamental Types



Type	Description	Examples
int	Integer (whole number)	5, -13, 0
double	Floating-point number (decimals)	3.14, 18.0, -33.999
num	Integer or floating-point number	5, 3.14, -13, 999.666
bool	Boolean	true, false
String	String of zero or more characters	"hi", "John Smith", "X", ""
List	List of values in series	[1, 2, 3], ["one", "two", "three"]
Map	Map of values by key	{"x": 8, "y": 16}
dynamic	Any type	

var Recap

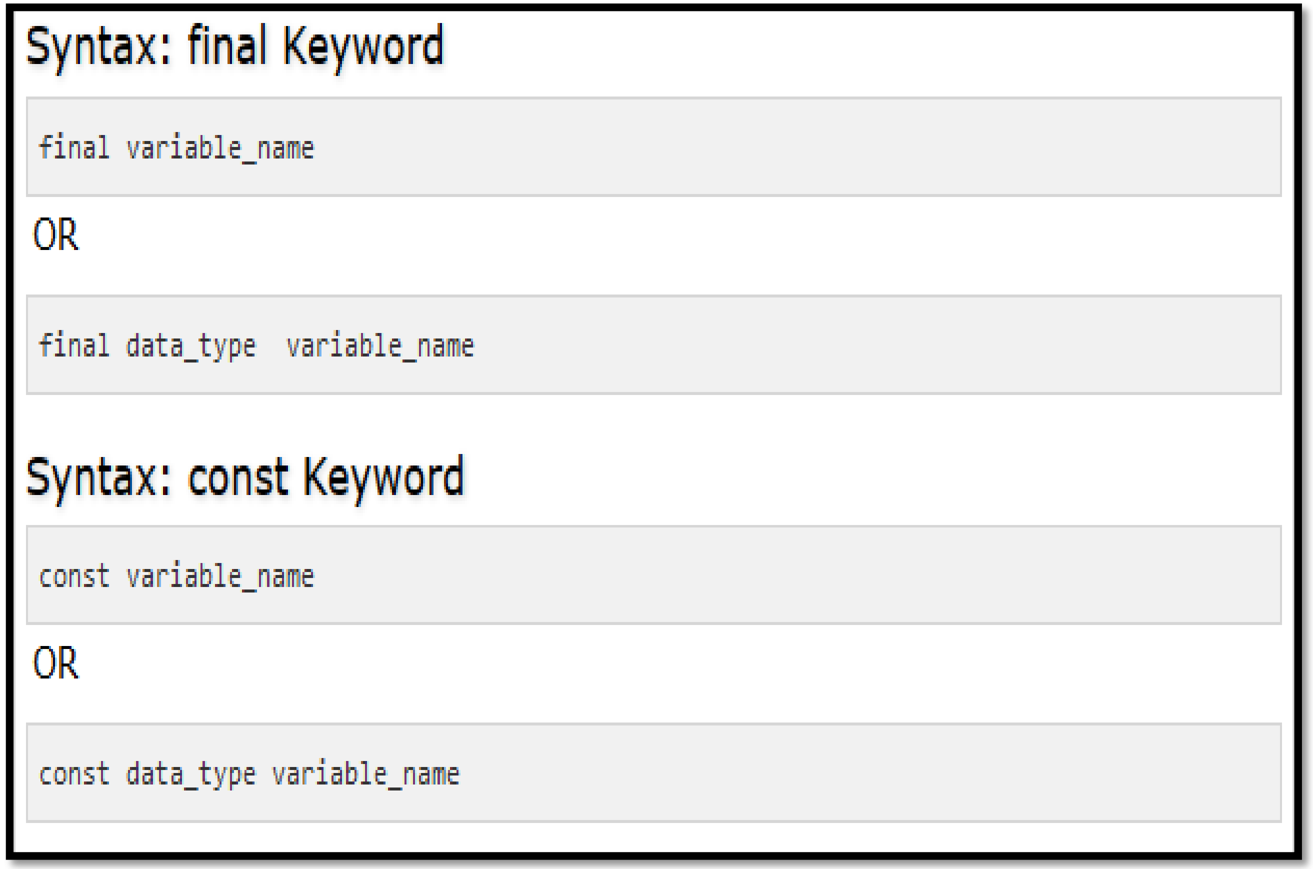
Remember, if you declare a variable with var, but don't assign a value in the same statement, Dart will assume it should be dynamic, but if you do initialize the value as you declare the variable, Dart will infer the type based on that value's type. So the following two lines are equivalent:

```
var z;              var y = 99;

dynamic z;          int y = 99;
```

Final and Const

The **final** and **const** keyword are used to declare constants. Dart prevents modifying the values of a variable declared using the final or const keyword. These keywords can be used in conjunction with the variable's data type or instead of the **var** keyword.

The **const** keyword is used to represent a compile-time constant. Variables declared using the **const** keyword are implicitly final.



Syntax: final Keyword

```
final variable_name
```

OR

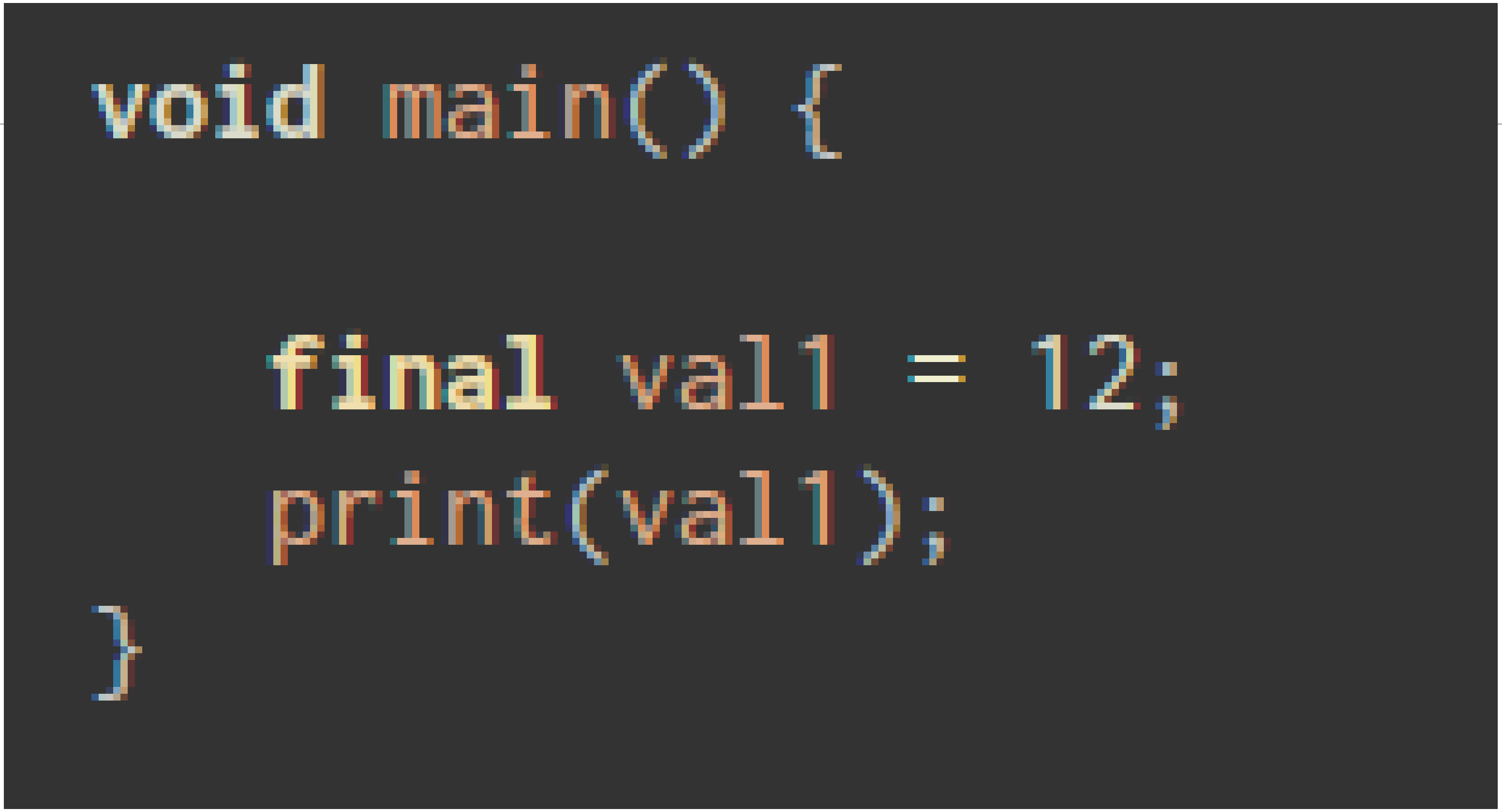
```
final data_type  variable_name
```

Syntax: const Keyword

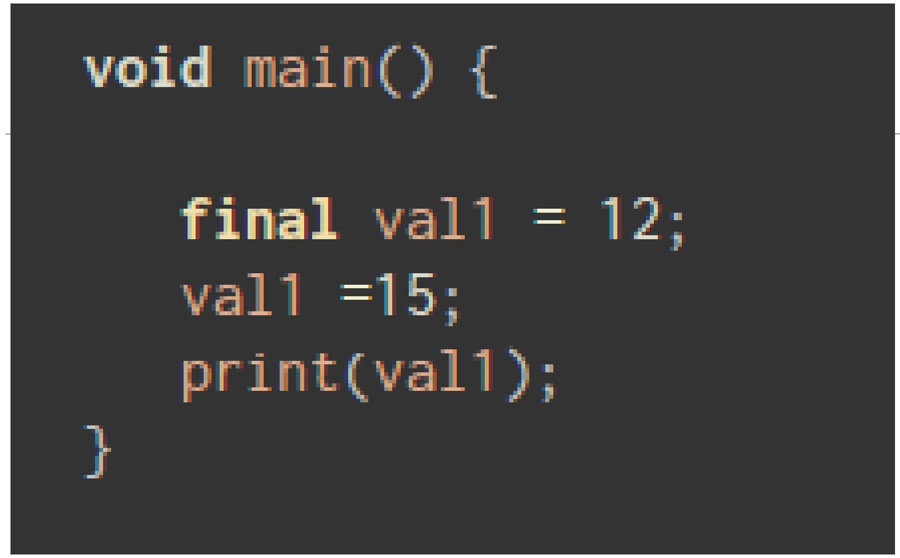
```
const variable_name
```

OR

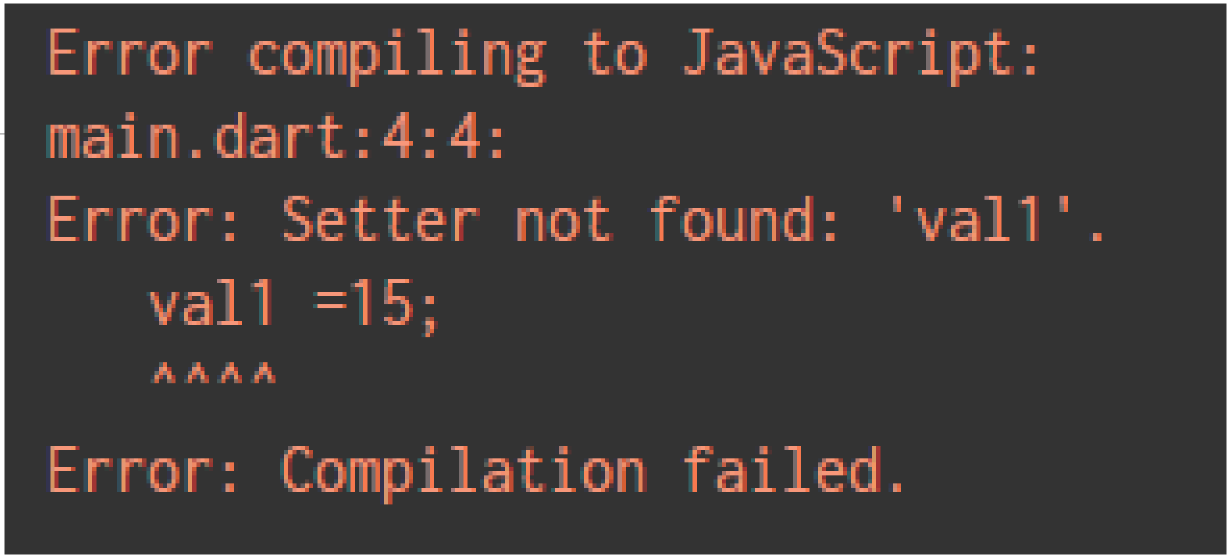
```
const data_type variable_name
```



```
void main() {

    final val1 = 12;
    print(val1);
}
```



```
void main() {

    final val1 = 12;
    val1 =15;
    print(val1);
}
```



```
Error compiling to JavaScript:
main.dart:4:4:
Error: Setter not found: 'val1'.
    val1 =15;
    ^^^^
Error: Compilation failed.
```

What caused the error?

Operators

An expression is a special kind of statement that evaluates to a value. Every expression is composed of:

Operands – Represents the data

Operator – Defines how the operands will be processed to produce a value.

Types of Operators

1. Arithmetic Operators
2. Equality and Relational Operators
3. Type test Operators
4. Bitwise Operators
5. Assignment Operators
6. Logical Operators

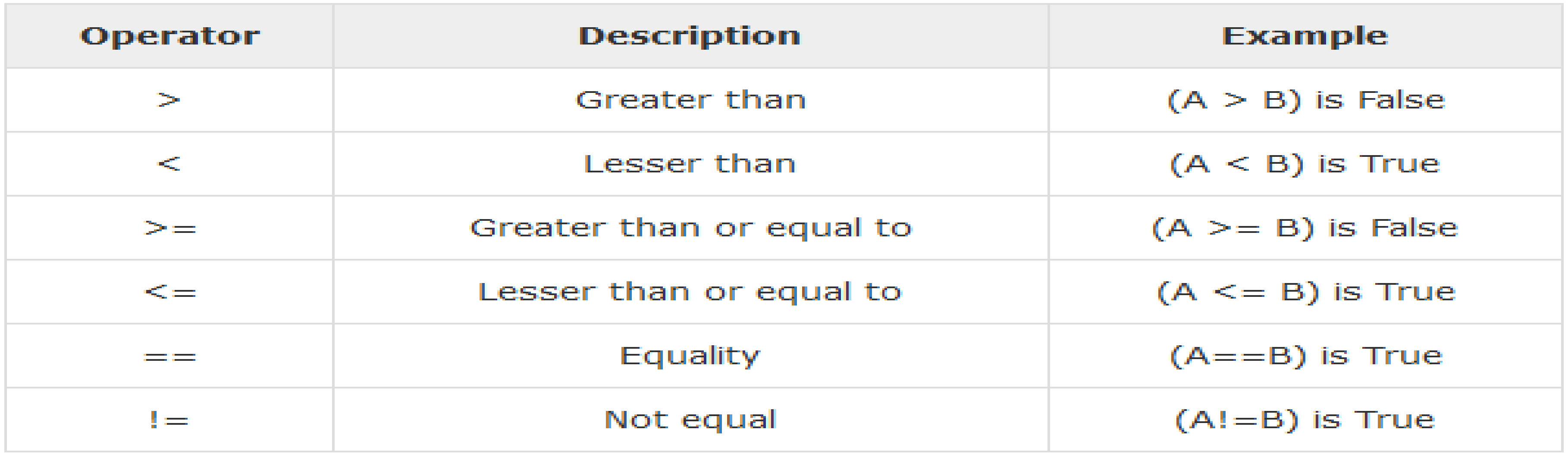
Arithmetic Operators

Operator	Meaning
+	Add
-	Subtract
-expr	Unary minus, also known as negation (reverse the sign of the expression)
*****	Multiply
/	Divide
~/	Divide, returning an integer result
%	Get the remainder of an integer division (modulo)
++	Increment
--	-- Decrement

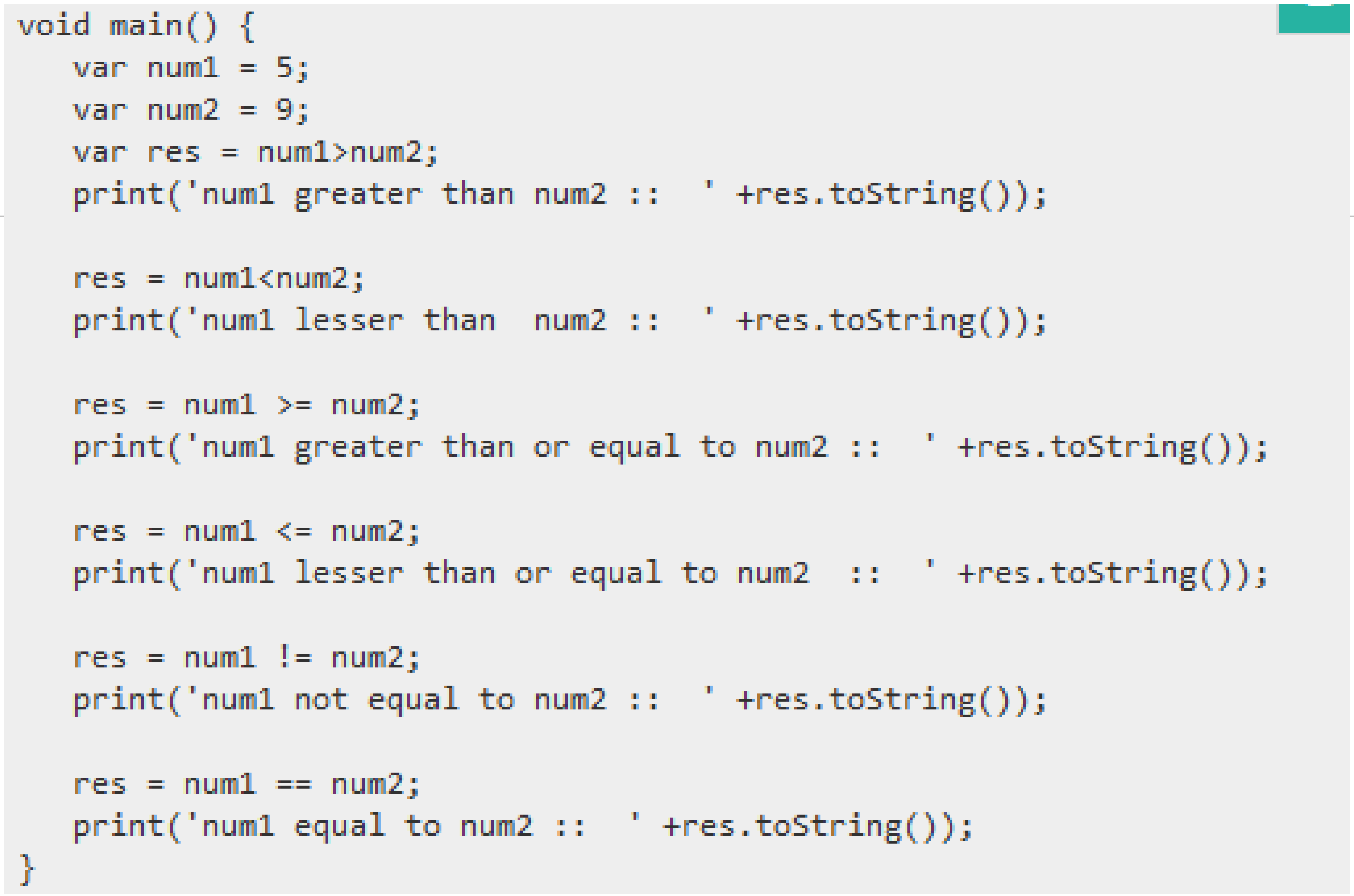
Equality and Relational Operators

Relational Operators tests or defines the kind of relationship between two entities. Relational operators return a Boolean value i.e. true/ false.

Assume the value of A is 10 and B is 20.



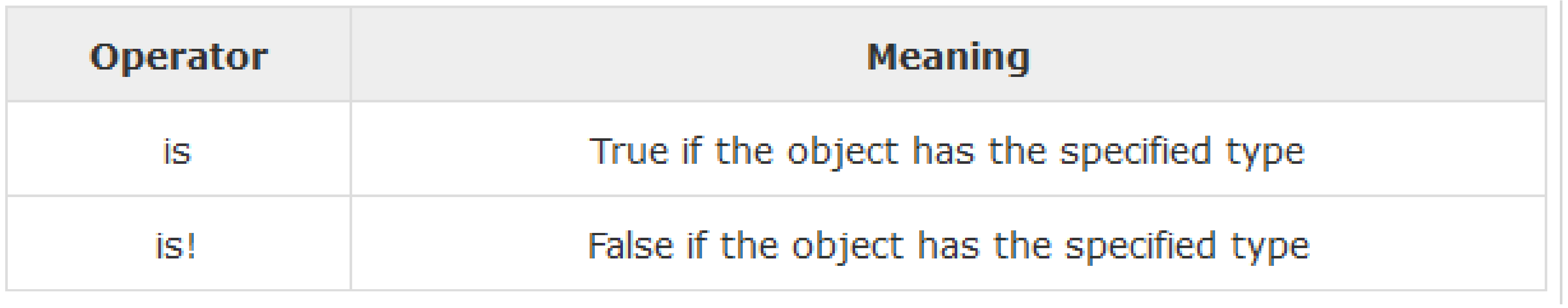
Operator	Description	Example
>	Greater than	(A > B) is False
<	Lesser than	(A < B) is True
>=	Greater than or equal to	(A >= B) is False
<=	Lesser than or equal to	(A <= B) is True
==	Equality	(A==B) is True
!=	Not equal	(A!=B) is True



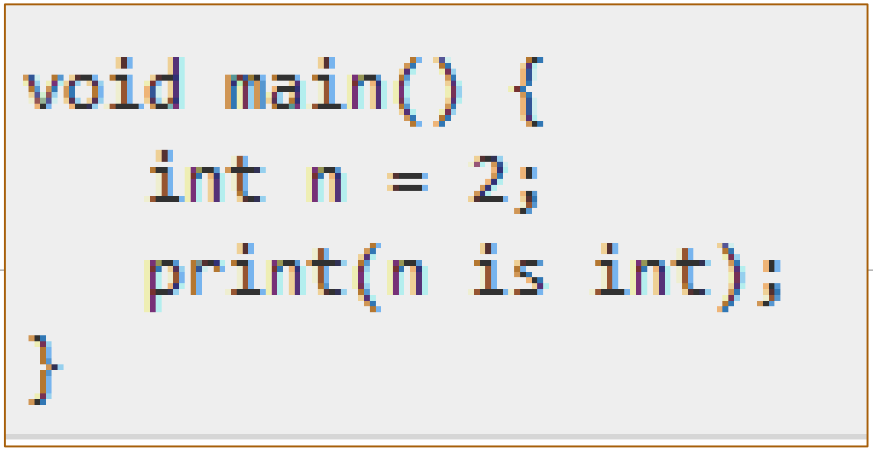
```
void main() {
   var num1 = 5;
   var num2 = 9;
   var res = num1>num2;
   print('num1 greater than num2 ::   ' +res.toString());

   res = num1<num2;
   print('num1 lesser than  num2 ::   ' +res.toString());

   res = num1 >= num2;
   print('num1 greater than or equal to num2 ::   ' +res.toString());

   res = num1 <= num2;
   print('num1 lesser than or equal to num2  ::   ' +res.toString());

   res = num1 != num2;
   print('num1 not equal to num2 ::   ' +res.toString());

   res = num1 == num2;
   print('num1 equal to num2 ::   ' +res.toString());
}
```

Type test Operators

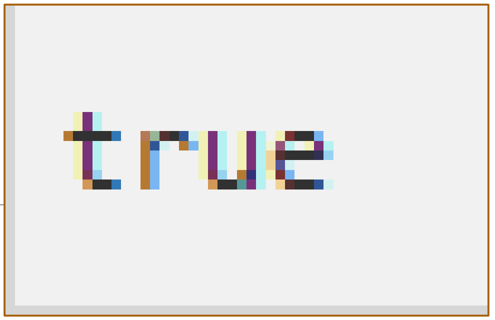
- These operators are handy for checking types at runtime.



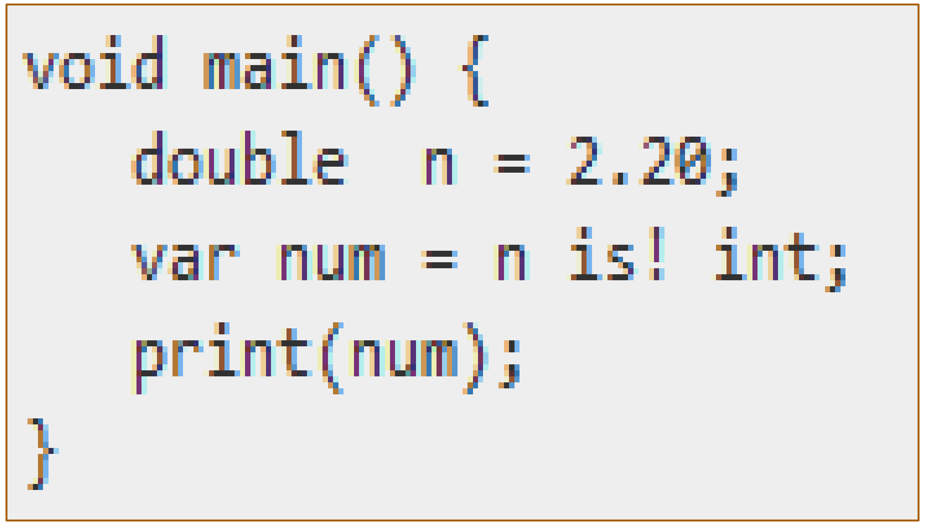
Operator	Meaning
is	True if the object has the specified type
is!	False if the object has the specified type



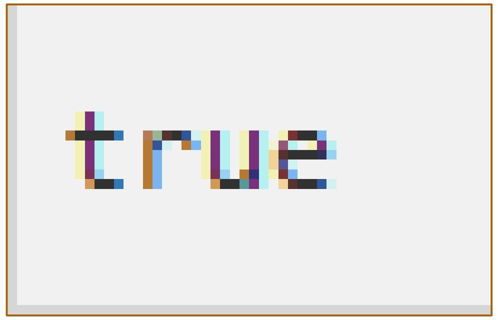
```
void main() {
    int n = 2;
    print(n is int);
}
```



```
true
```



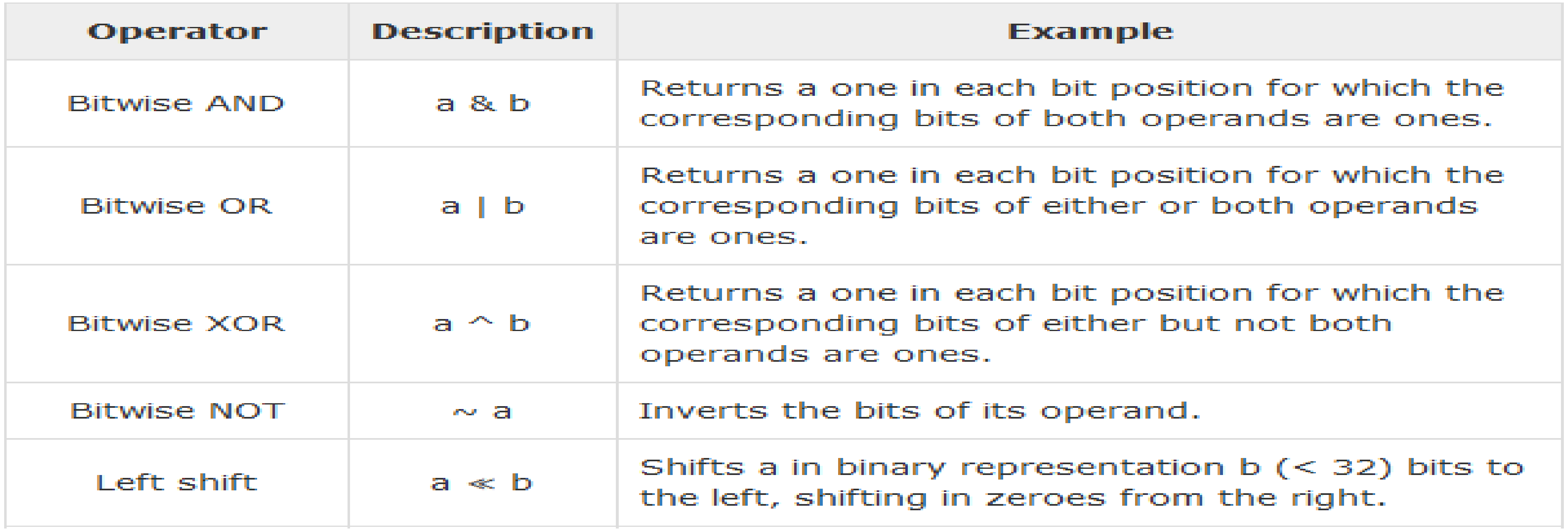
```
void main() {
   double  n = 2.20;
   var num = n is! int;
   print(num);
}
```



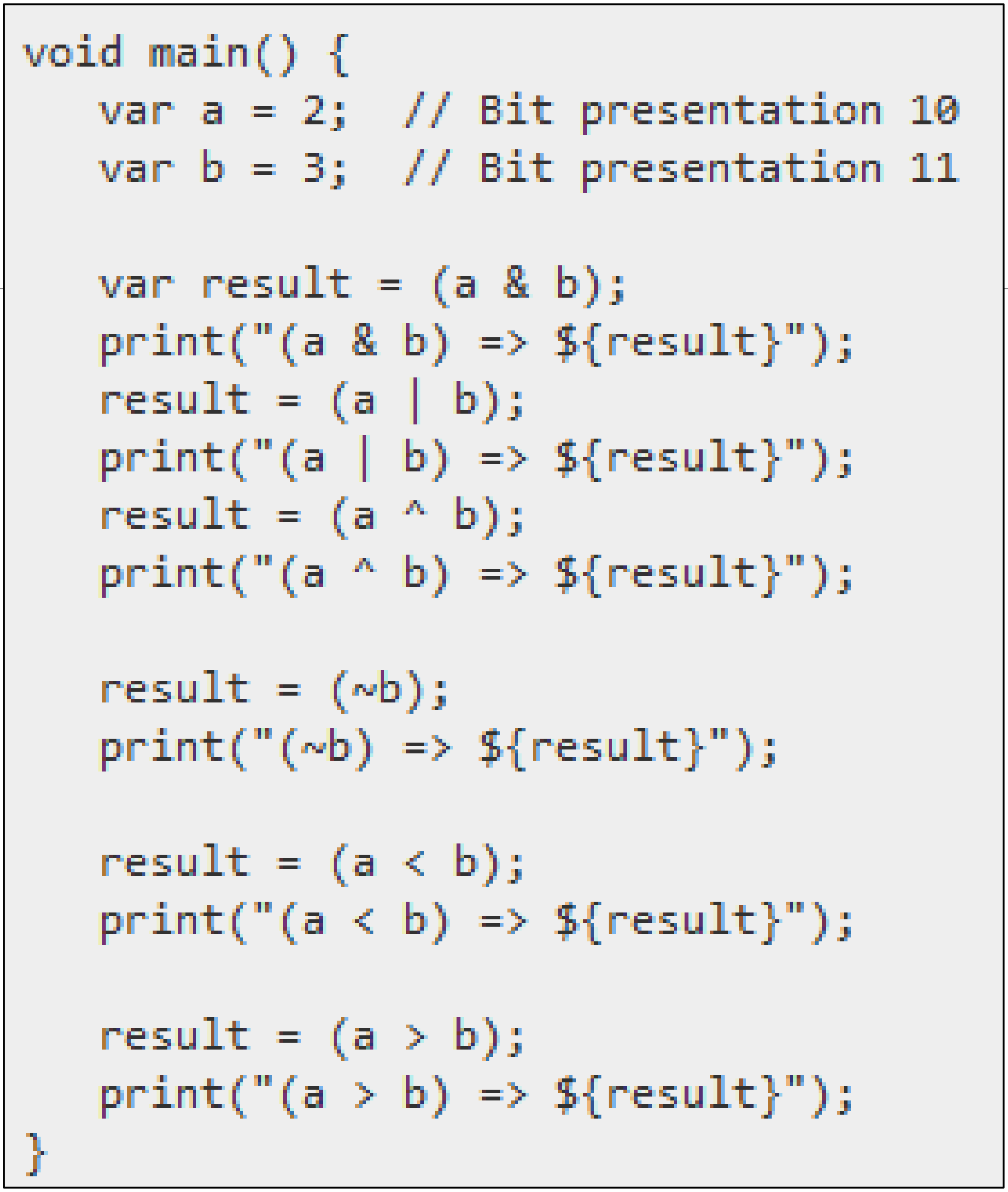
```
true
```

Bitwise Operators

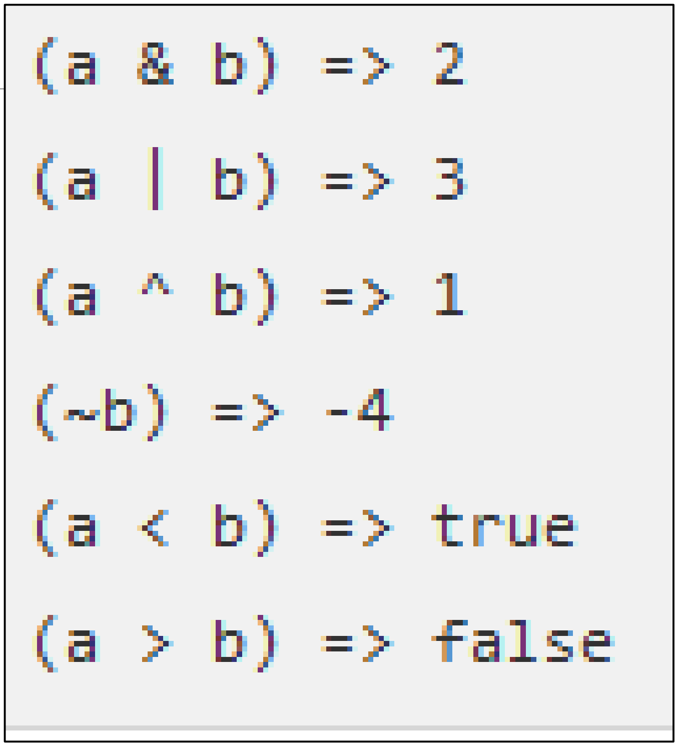
Bitwise operators are operators (just like +, *, &&, etc.) that operate on **ints** and **uints** at the binary level.



Operator	Description	Example
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.



```
void main() {
   var a = 2;  // Bit presentation 10
   var b = 3;  // Bit presentation 11

   var result = (a & b);
   print("(a & b) => ${result}");
   result = (a | b);
   print("(a | b) => ${result}");
   result = (a ^ b);
   print("(a ^ b) => ${result}");

   result = (~b);
   print("(~b) => ${result}");

   result = (a < b);
   print("(a < b) => ${result}");

   result = (a > b);
   print("(a > b) => ${result}");
}
```



```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a < b) => true
(a > b) => false
```

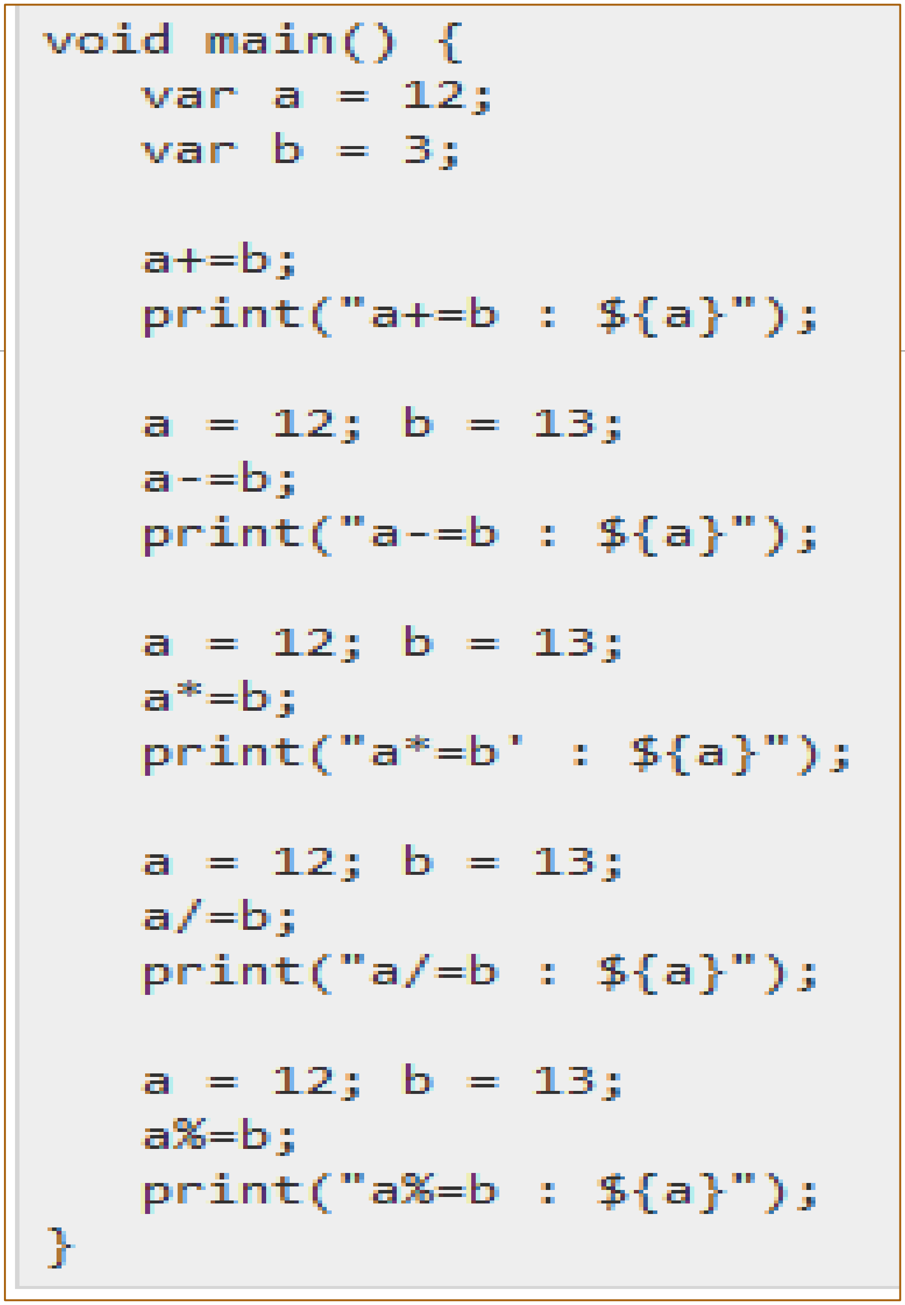
Assignment Operators

➢Operations that allow the assignment of value to a variable

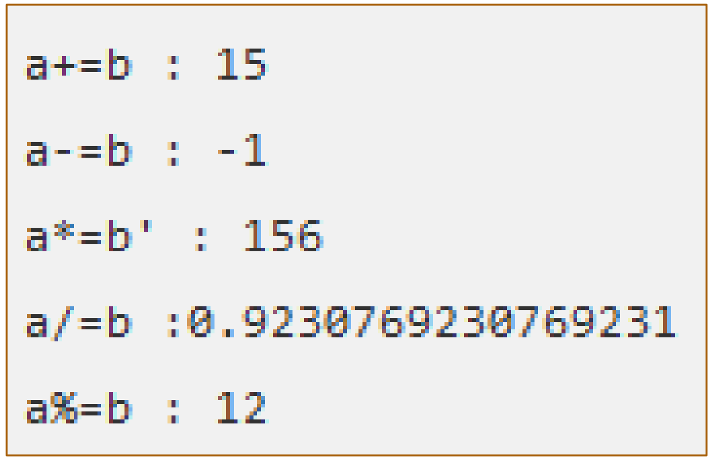
Sr.No	Operator & Description
1	**=(Simple Assignment)** Assigns values from the right side operand to the left side operand **Ex**: C = A + B will assign the value of A + B into C
2	**??=** Assign the value only if the variable is null
3	**+=(Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand. **Ex**: C += A is equivalent to C = C + A

Assignment Operators

Sr.No	Operator & Description
4	**–= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex**: C -= A is equivalent to C = C – A
5	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex**: C *= A is equivalent to C = C * A
6	**/= (Divide and Assignment)** It divides the left operand with the right operand and assigns the result to the left operand.



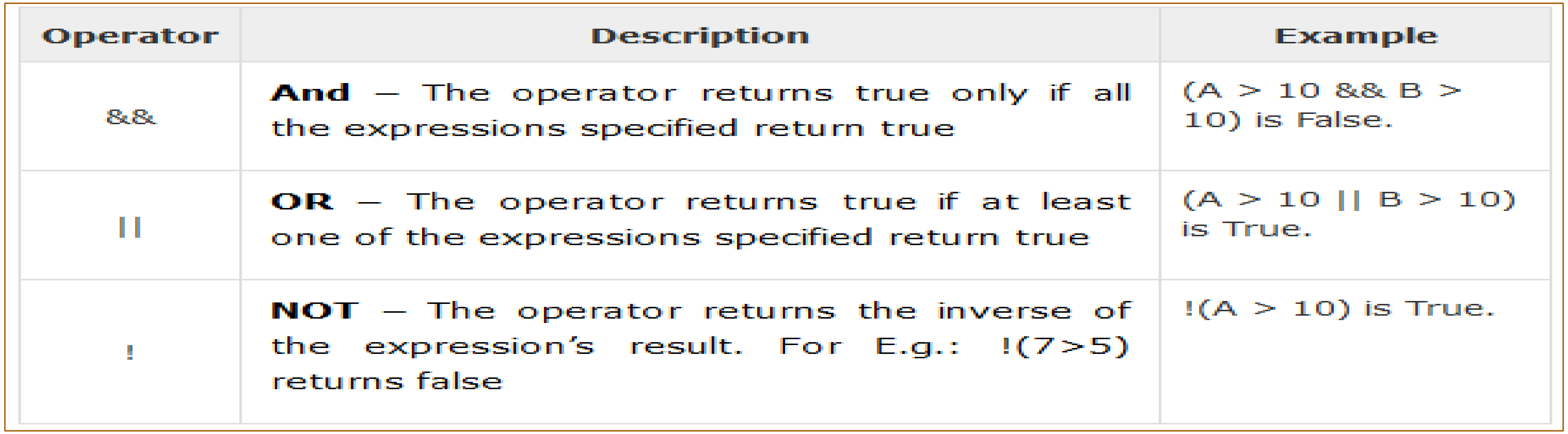
```
void main() {
    var a = 12;
    var b = 3;

    a+=b;
    print("a+=b : ${a}");

    a = 12; b = 13;
    a-=b;
    print("a-=b : ${a}");

    a = 12; b = 13;
    a*=b;
    print("a*=b' : ${a}");

    a = 12; b = 13;
    a/=b;
    print("a/=b : ${a}");

    a = 12; b = 13;
    a%=b;
    print("a%=b : ${a}");
}
```



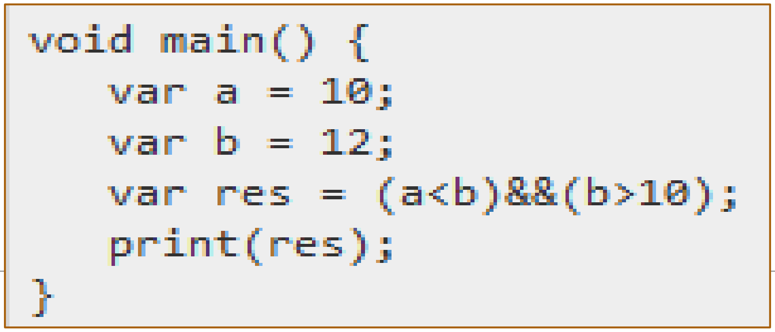
```
a+=b : 15

a-=b : -1

a*=b' : 156

a/=b :0.9230769230769231

a%=b : 12
```

Logical Operators

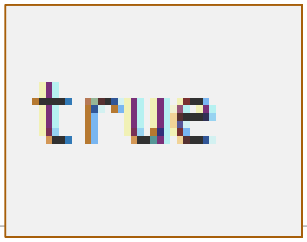
- Logical operators are used to combine two or more conditions. Logical operators return a Boolean value. Assume the value of variable A is 10 and B is 20.



Operator	Description	Example
&&	**And** – The operator returns true only if all the expressions specified return true	(A > 10 && B > 10) is False.
||	**OR** – The operator returns true if at least one of the expressions specified return true	(A > 10 || B > 10) is True.
!	**NOT** – The operator returns the inverse of the expression's result. For E.g.: !(7>5) returns false	!(A > 10) is True.

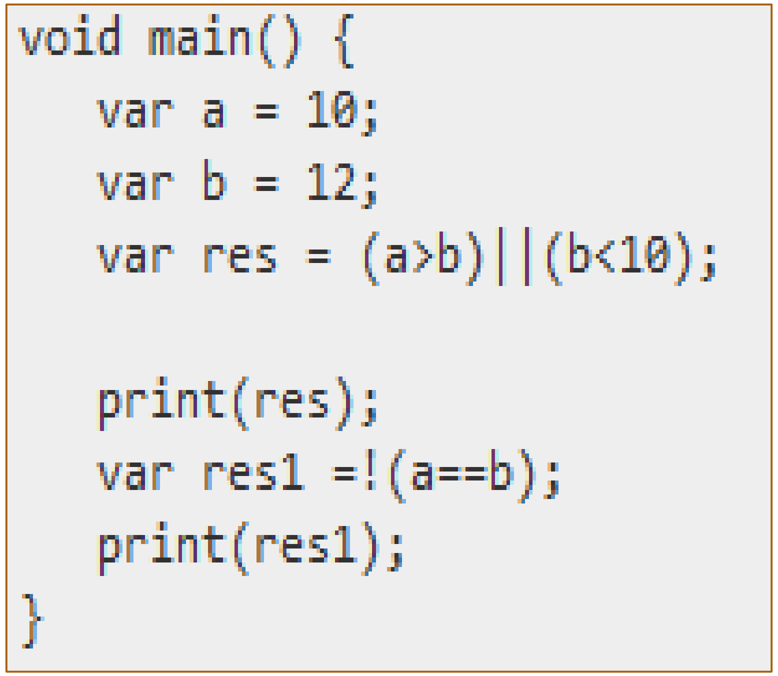


```
void main() {
    var a = 10;
    var b = 12;
    var res = (a<b)&&(b>10);
    print(res);
}
```

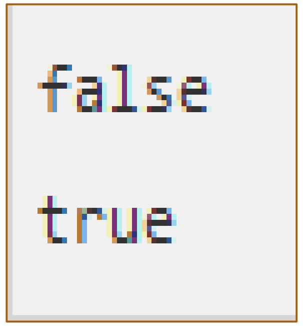


```
true
```

&& and ||



```
void main() {
   var a = 10;
   var b = 12;
   var res = (a>b)||(b<10);

   print(res);
   var res1 =!(a==b);
   print(res1);
}
```



```
false
true
```

Conditional Expressions

```
if (temperature > 75) {
  print("It is hot today.");  //print is used to output text to the console
}
```

Conditional Expressions

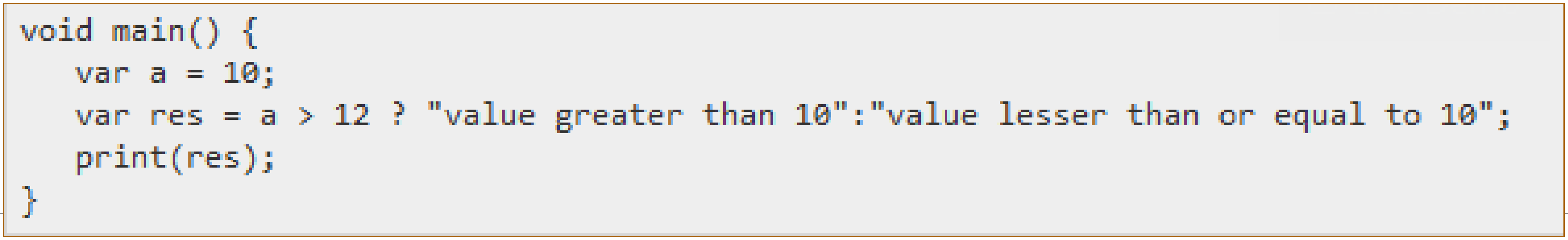
Dart has two operators that let you evaluate expressions that might otherwise require ifelse statements :

condition ? expr1 : expr2

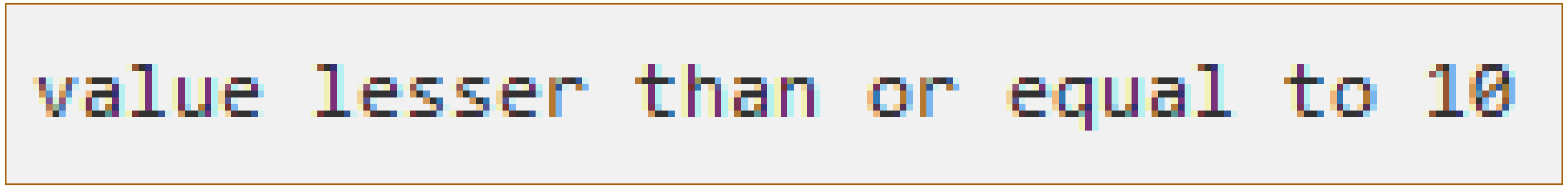
If condition is true, then the expression evaluates **expr1** (and returns its value); otherwise, it evaluates and returns the value of **expr2**.

expr1 ?? expr2

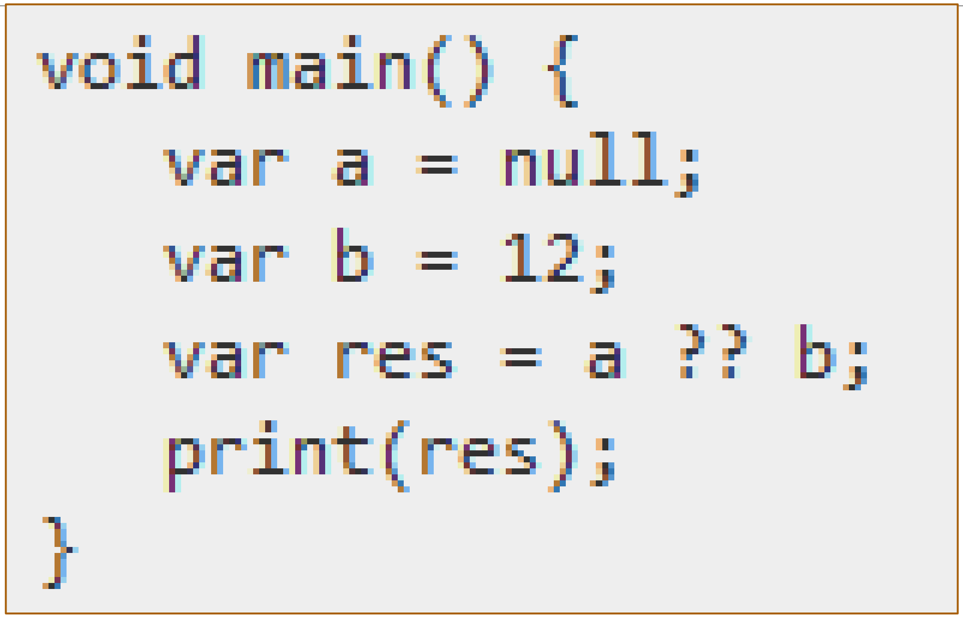
If **expr1** is non-null, returns its value; otherwise, evaluates and returns the value of **expr2**



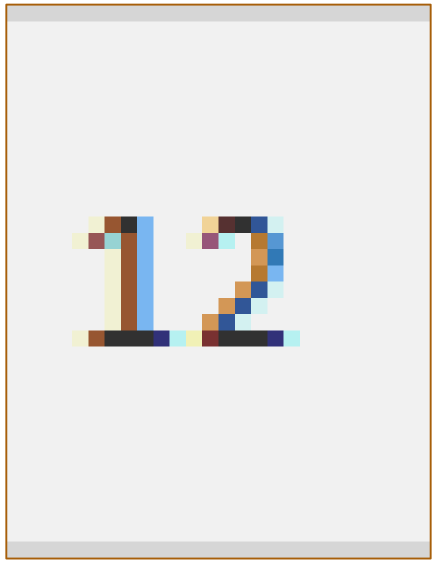
```
void main() {
   var a = 10;
   var res = a > 12 ? "value greater than 10":"value lesser than or equal to 10";
   print(res);
}
```



```
value lesser than or equal to 10
```



```
void main() {
    var a = null;
    var b = 12;
    var res = a ?? b;
    print(res);
}
```

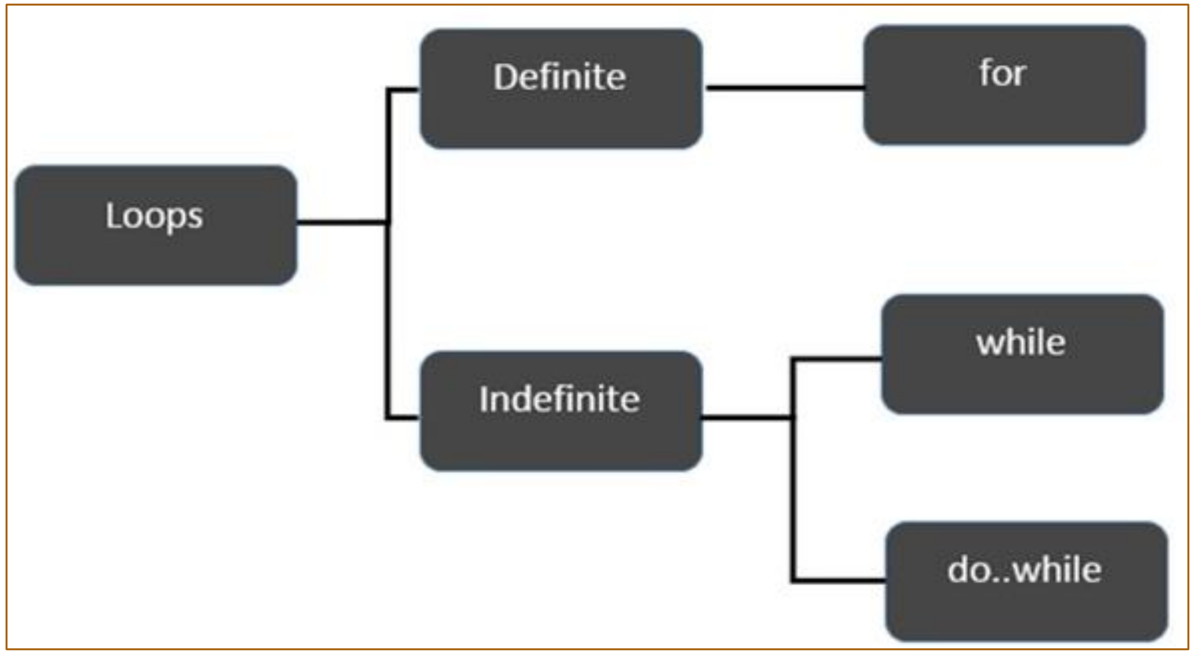


```
12
```

Dart Programming - Loops

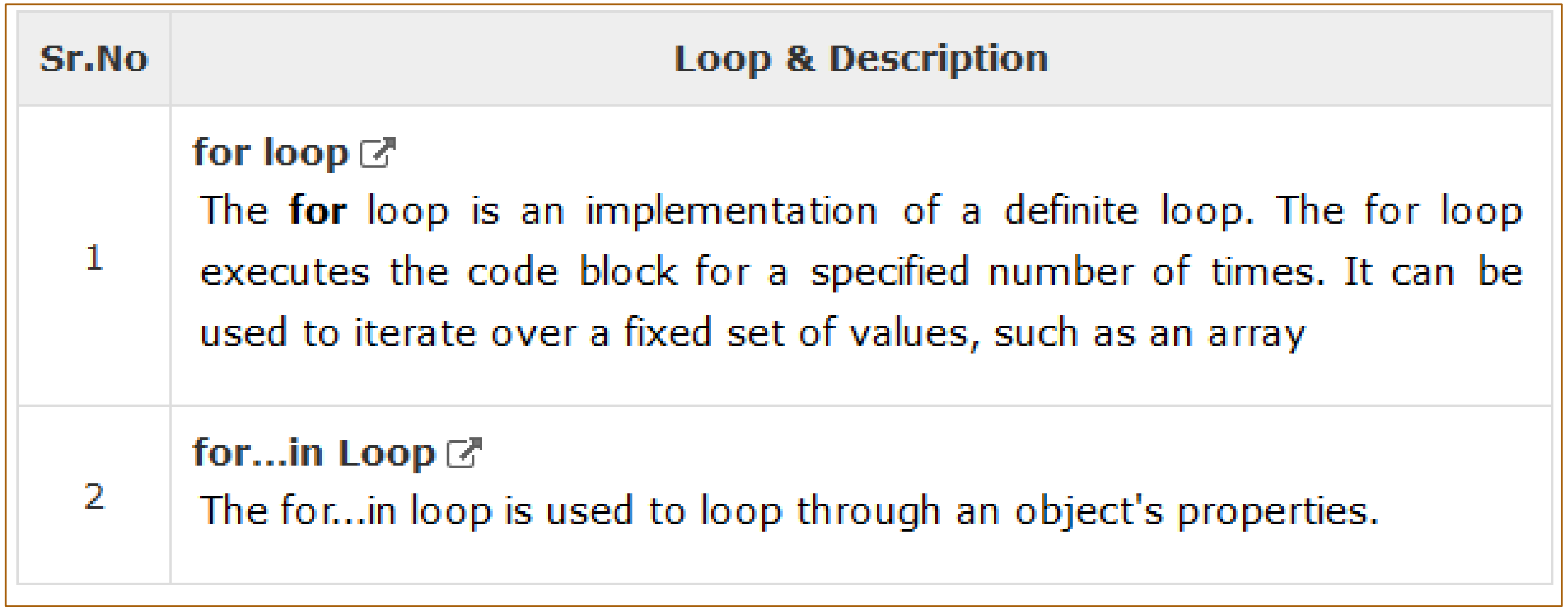
Loops

- Loop statements are used to execute the block of code repeatedly for a specified number of times or until it meets a specified condition.



Definite Loop

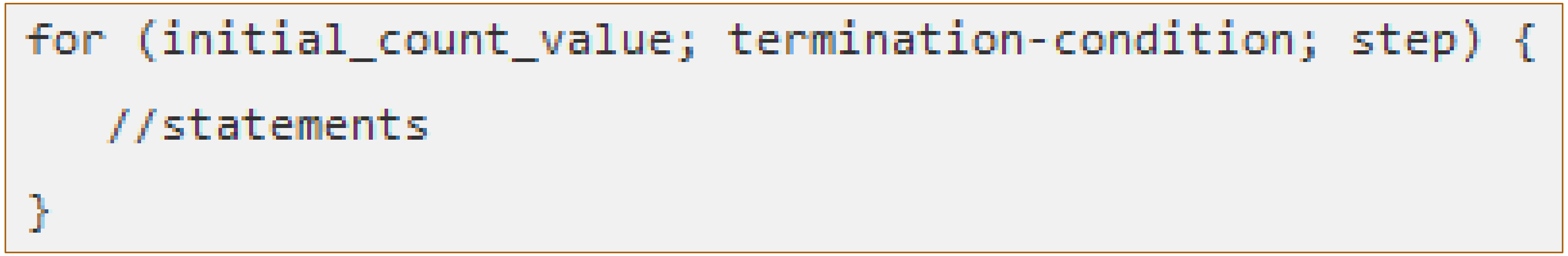
➢A loop whose number of iterations are definite/fixed is termed as a **definite loop**.



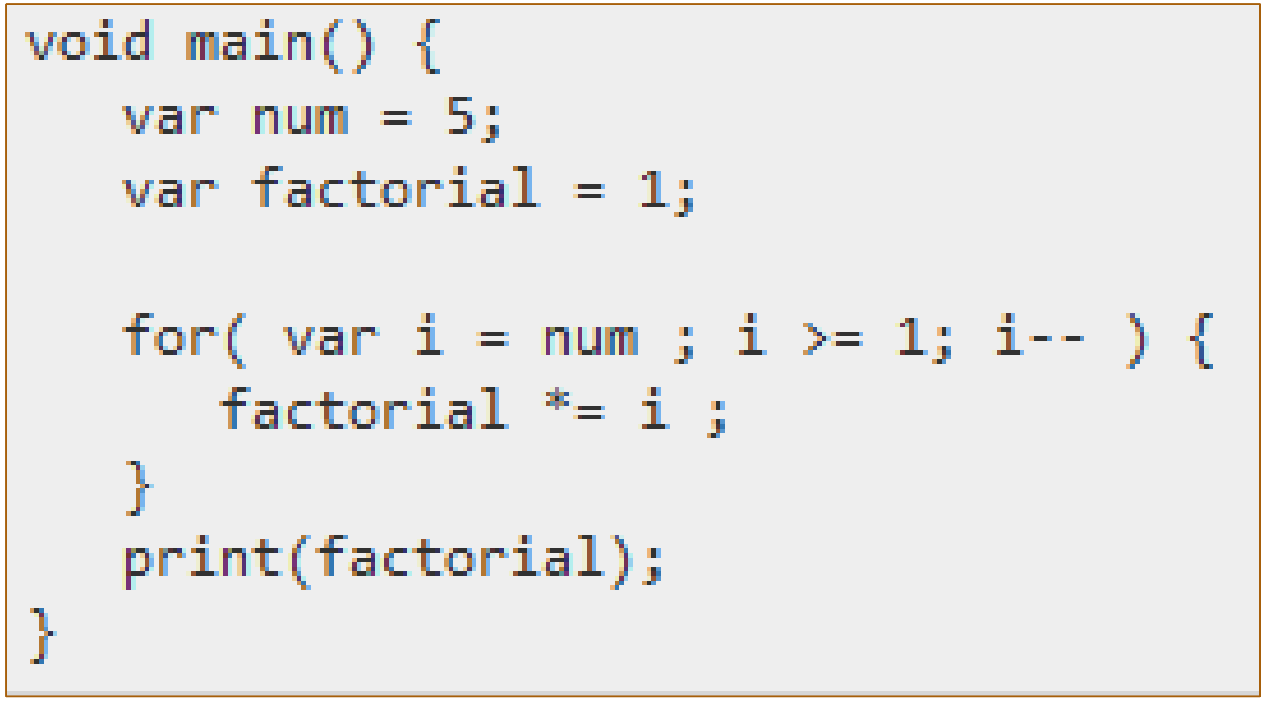
Sr.No	Loop & Description
1	**for loop** The **for** loop is an implementation of a definite loop. The for loop executes the code block for a specified number of times. It can be used to iterate over a fixed set of values, such as an array
2	**for...in Loop** The for...in loop is used to loop through an object's properties.

for loop

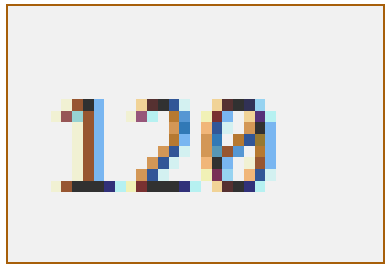
- The **for** loop executes the code block for a specified number of times. It can be used to iterate over a fixed set of values, such as an array.



```
for (initial_count_value; termination-condition; step) {

   //statements

}
```



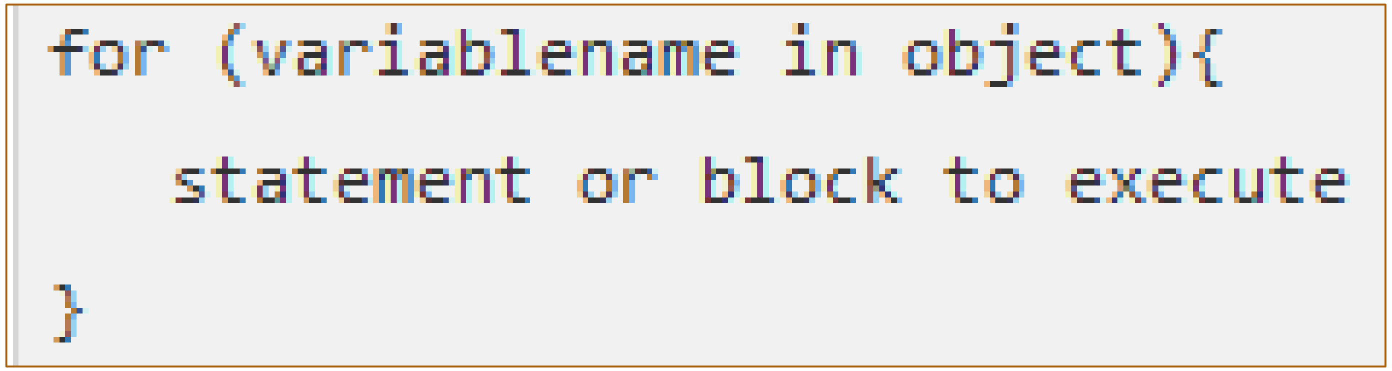
```
void main() {
   var num = 5;
   var factorial = 1;

   for( var i = num ; i >= 1; i-- ) {
      factorial *= i ;
   }
   print(factorial);
}
```



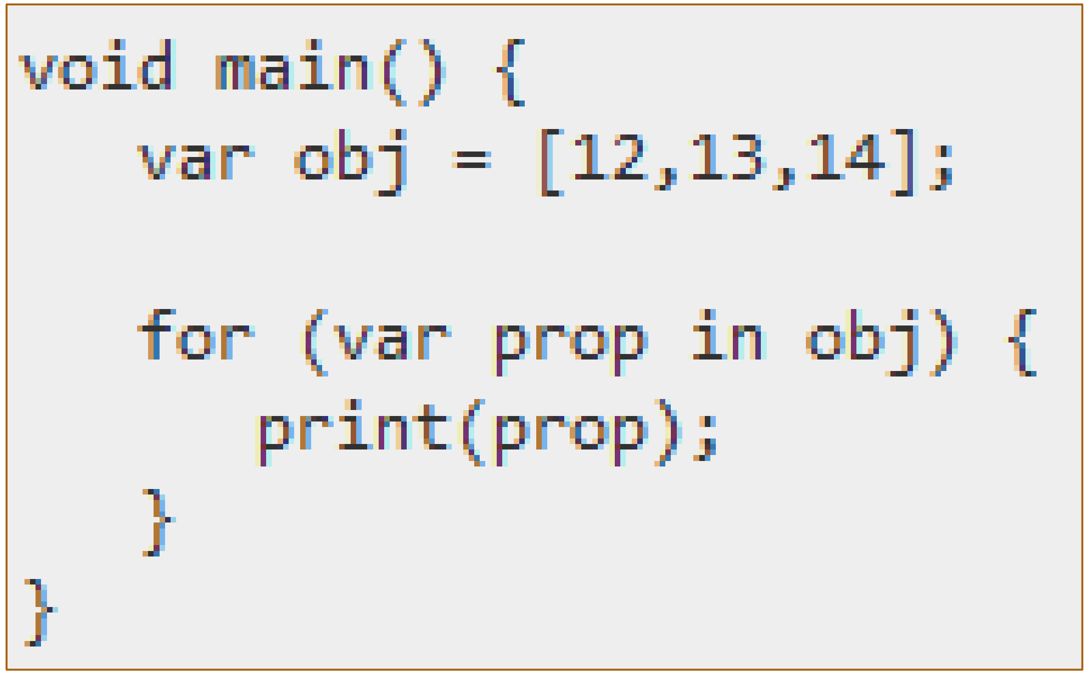
```
120
```

for...in loop

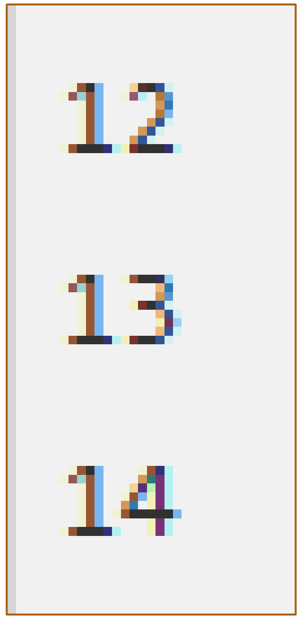
- The for...in loop is used to loop through an object's properties.



```
for (variablename in object){
    statement or block to execute
}
```



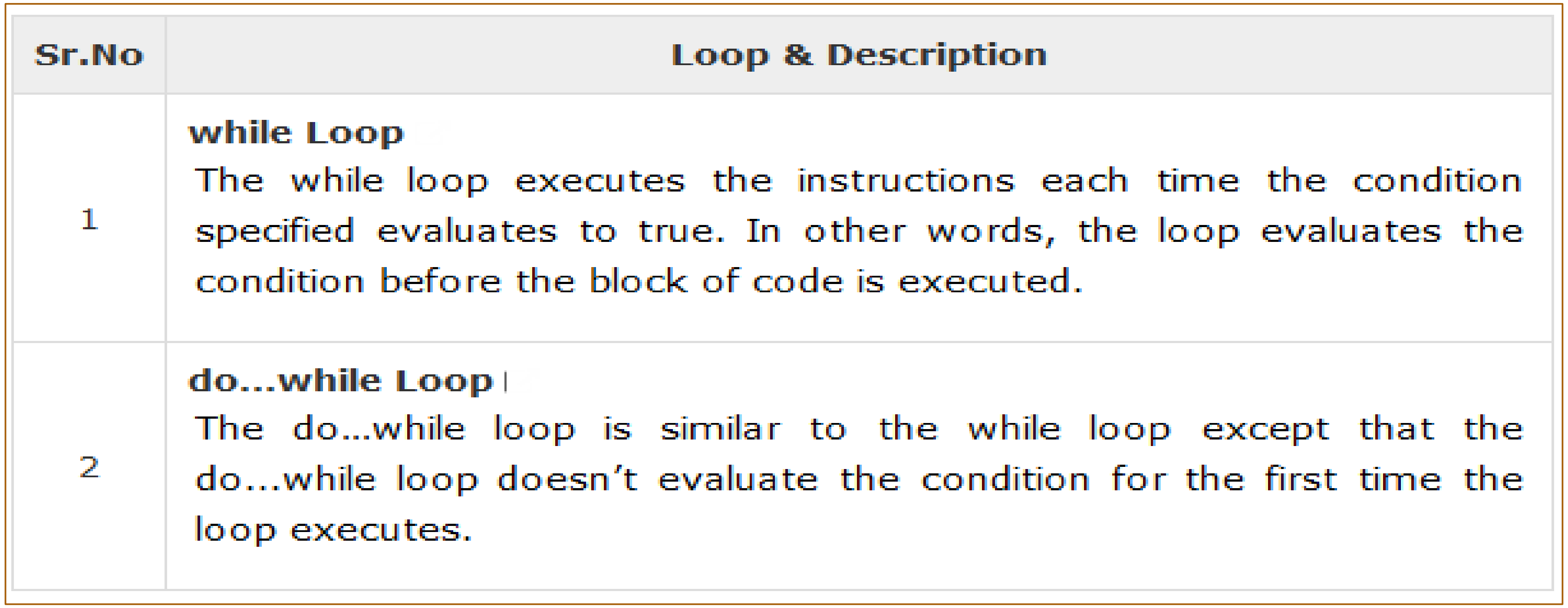
```
void main() {
    var obj = [12,13,14];

    for (var prop in obj) {
        print(prop);
    }
}
```



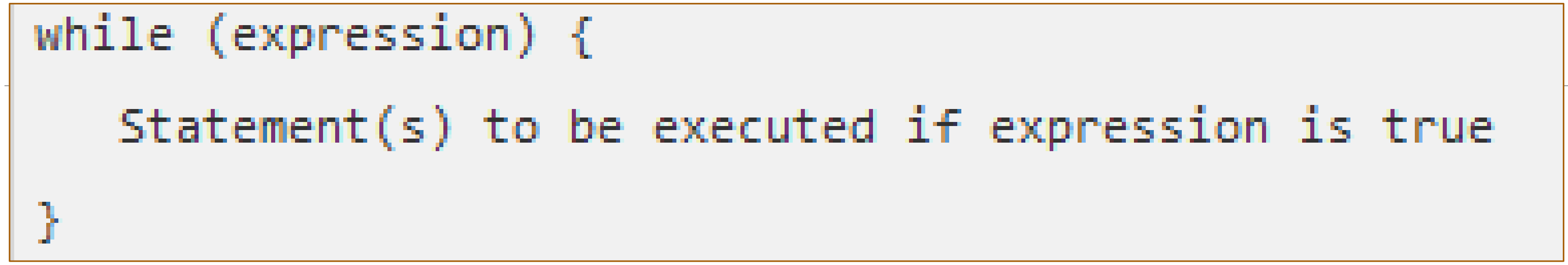
```
12
13
14
```

Indefinite Loop

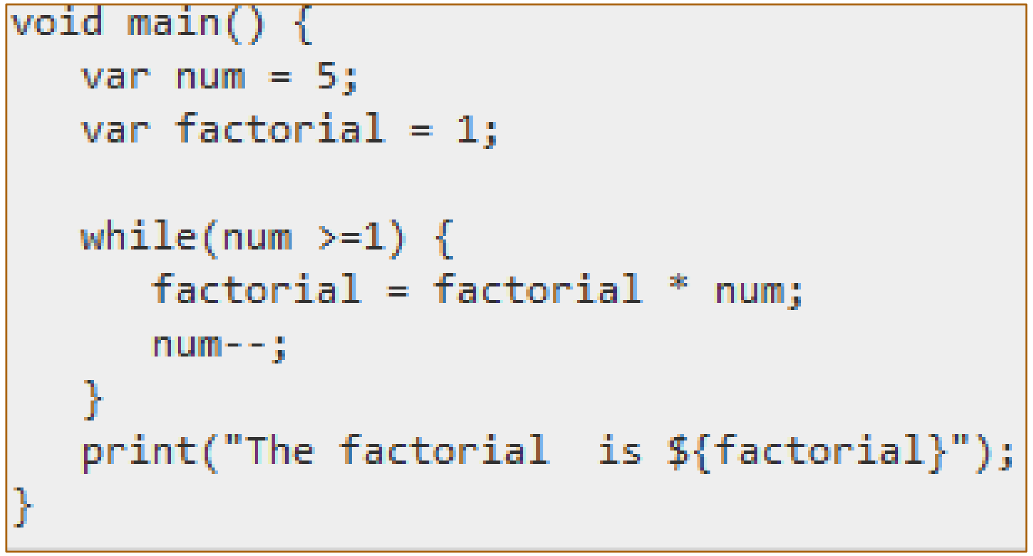
- Used when the number of iterations in a loop is indeterminate or unknown. Indefinite loops can be implemented using –



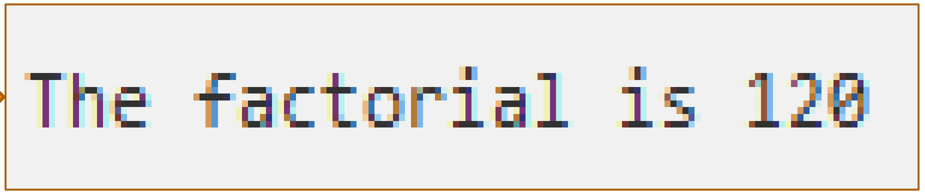
Sr.No	Loop & Description
1	**while Loop** The while loop executes the instructions each time the condition specified evaluates to true. In other words, the loop evaluates the condition before the block of code is executed.
2	**do...while Loop** The do…while loop is similar to the while loop except that the do...while loop doesn’t evaluate the condition for the first time the loop executes.



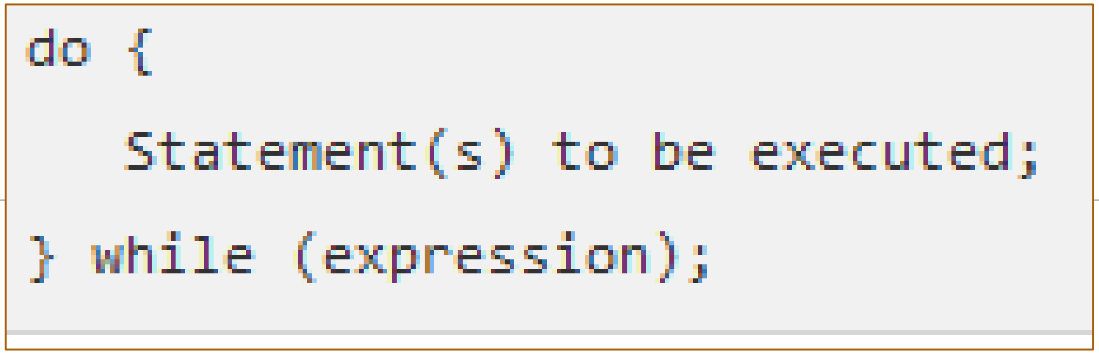
```
while (expression) {

   Statement(s) to be executed if expression is true

}
```



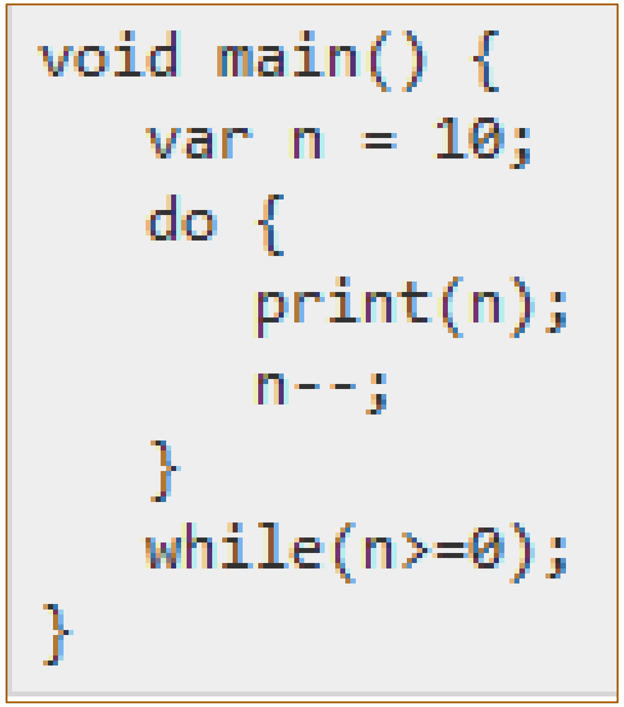
```
void main() {
   var num = 5;
   var factorial = 1;

   while(num >=1) {
      factorial = factorial * num;
      num--;
   }
   print("The factorial  is ${factorial}");
}
```



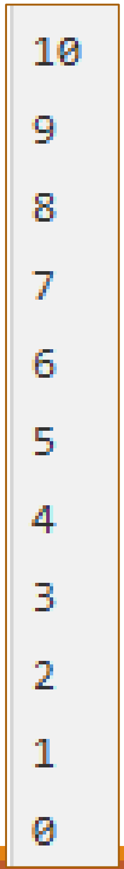
```
The factorial is 120
```



```
do {
   Statement(s) to be executed;
} while (expression);
```



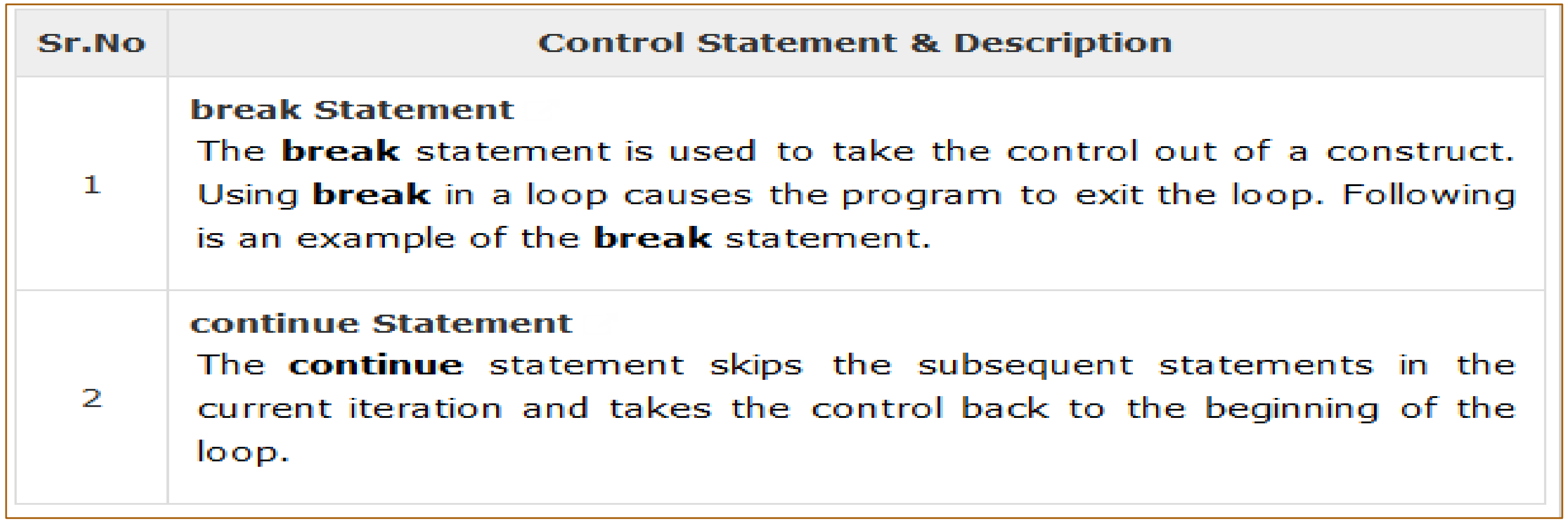
```
void main() {
   var n = 10;
   do {
      print(n);
      n--;
   }
   while(n>=0);
}
```



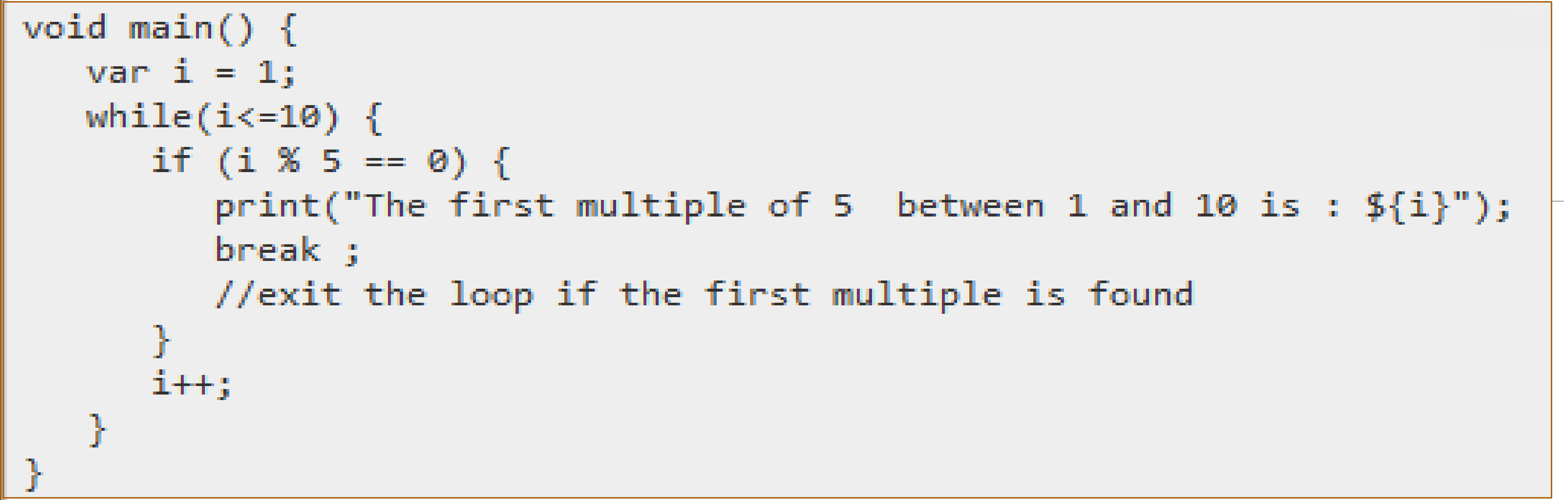
```
10
9
8
7
6
5
4
3
2
1
0
```

Loop Control Statements

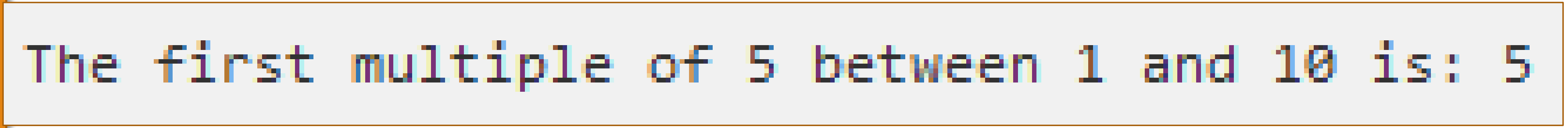
- **Control** flow **statements** allow us to break up the flow of code by employing decision-making, **looping**, branching and enabling our program to conditionally execute particular blocks of code



Sr.No	Control Statement & Description
1	**break Statement** The **break** statement is used to take the control out of a construct. Using **break** in a loop causes the program to exit the loop. Following is an example of the **break** statement.
2	**continue Statement** The **continue** statement skips the subsequent statements in the current iteration and takes the control back to the beginning of the loop.



```dart
void main() {
   var i = 1;
   while(i<=10) {
      if (i % 5 == 0) {
         print("The first multiple of 5  between 1 and 10 is : ${i}");
         break ;
         //exit the loop if the first multiple is found
      }
      i++;
   }
}
```



```
The first multiple of 5 between 1 and 10 is: 5
```

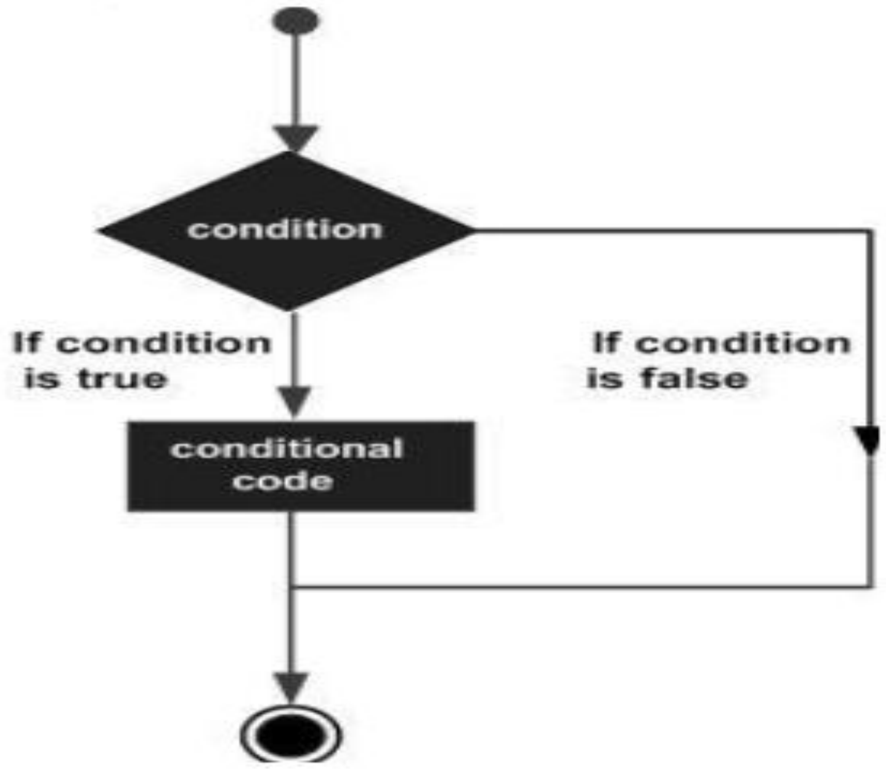
```
void main() {
   var num = 0;
   var count = 0;

   for(num = 0;num<=20;num++) {
      if (num % 2==0) {
         continue;
      }
      count++;
   }
   print(" The count of odd values between 0 and 20 is: ${count}");
}
```

```
The count of odd values between 0 and 20 is: 10
```

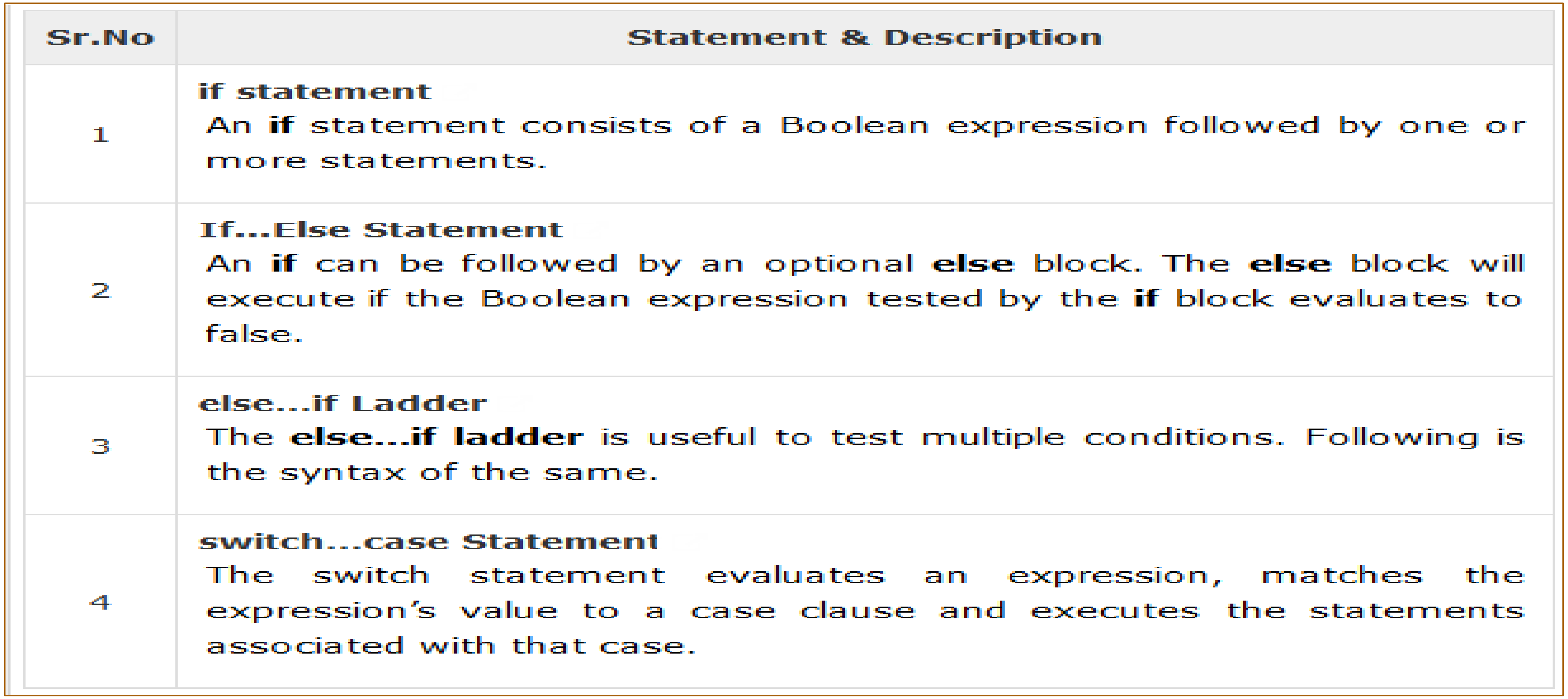
Dart Programming - Decision Making

Conditional/Decision-Making

- A conditional/decision-making construct evaluates a condition before the instructions are executed.



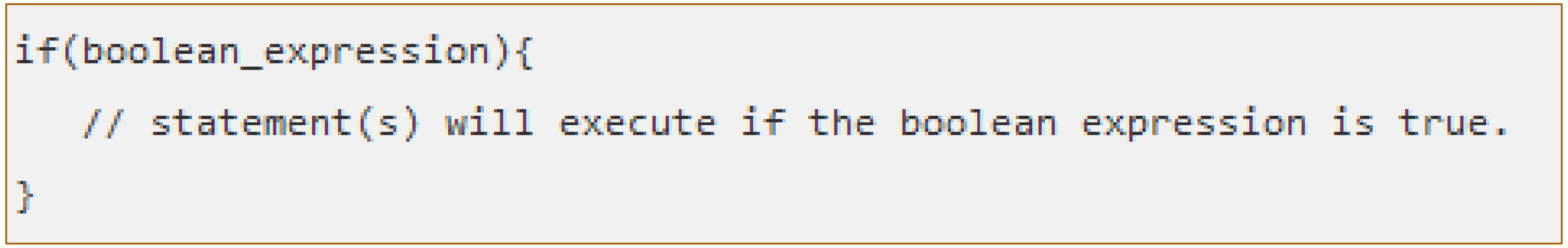
Conditional constructs in Dart are classified in the following



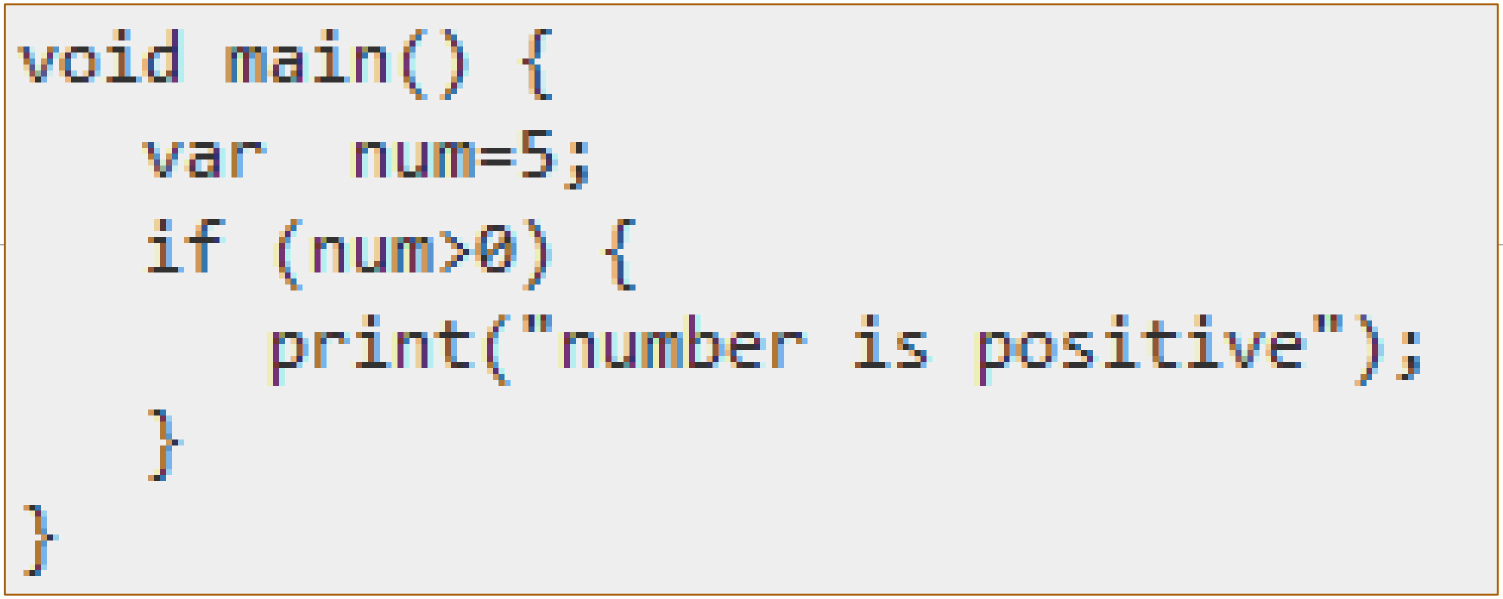
Sr.No	Statement & Description
1	**if statement** An **if** statement consists of a Boolean expression followed by one or more statements.
2	**If...Else Statement** An **if** can be followed by an optional **else** block. The **else** block will execute if the Boolean expression tested by the **if** block evaluates to false.
3	**else...if Ladder** The **else...if ladder** is useful to test multiple conditions. Following is the syntax of the same.
4	**switch...case Statement** The switch statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case.

If statement

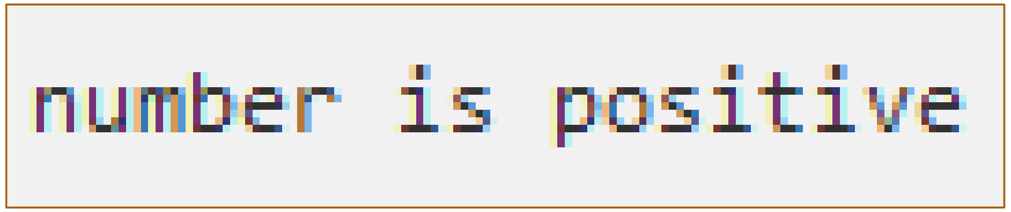
- The **if** construct evaluates a condition before a block of code is executed.



```
if(boolean_expression){
   // statement(s) will execute if the boolean expression is true.
}
```



```
void main() {
    var  num=5;
    if (num>0) {
        print("number is positive");
    }
}
```

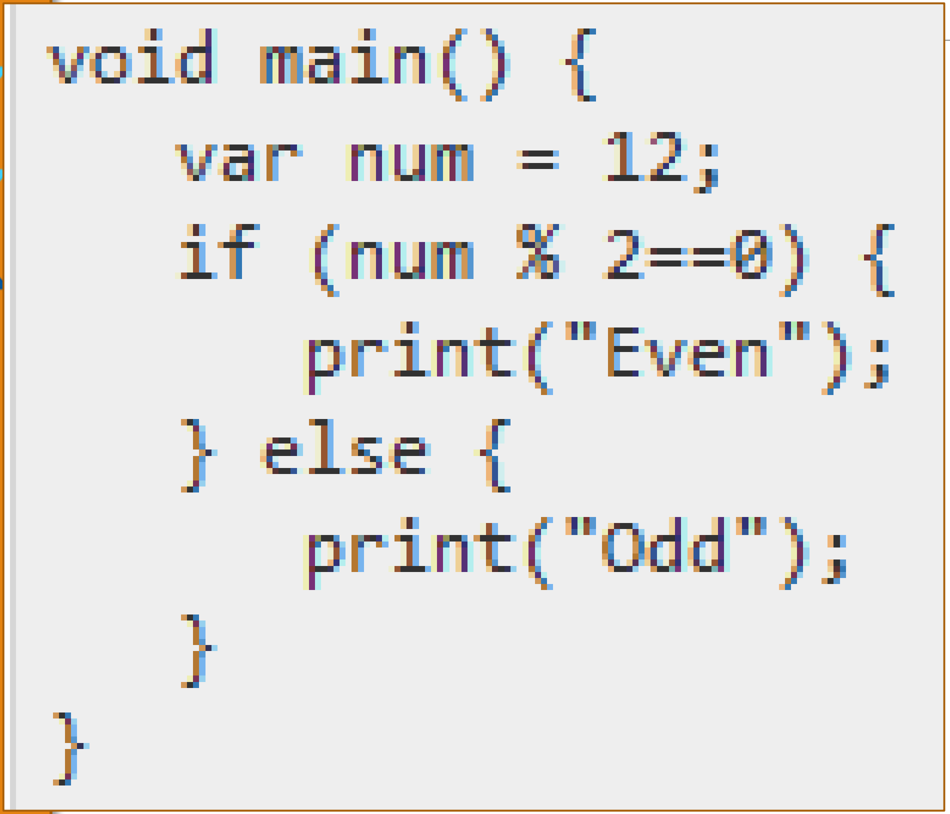


```
number is positive
```

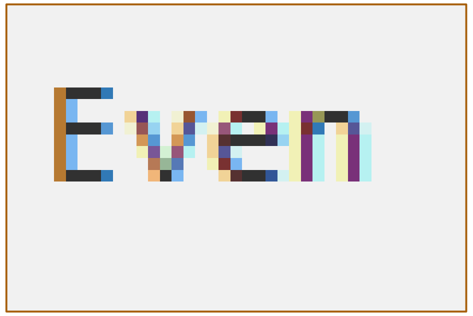
If … Else Statement

- An **if** can be followed by an optional **else** block. The **else** block will execute if the Boolean expression tested by the **if** block evaluates to false.

```
if(boolean_expression){
   // statement(s) will execute if the Boolean expression is true.
} else {
   // statement(s) will execute if the Boolean expression is false.
}
```



```
void main() {
    var num = 12;
    if (num % 2==0) {
        print("Even");
    } else {
        print("Odd");
    }
}
```

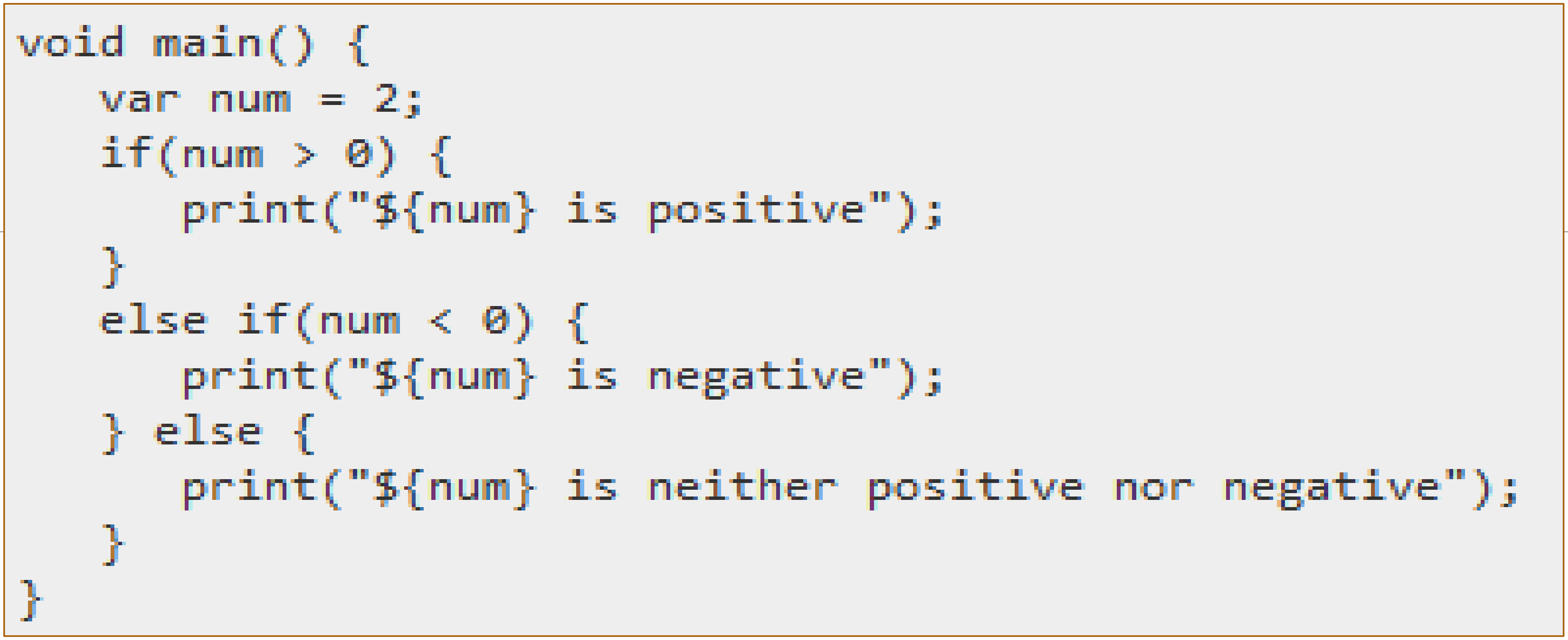


```
Even
```

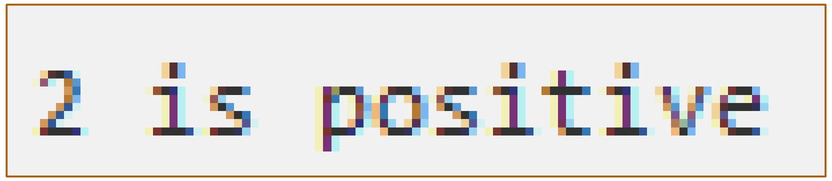
else…if ladder statement

➢The **else…if ladder** is useful to test multiple conditions. Following is the syntax of the same.

```
if (boolean_expression1) {
   //statements if the expression1 evaluates to true
}
else if (boolean_expression2) {
   //statements if the expression2 evaluates to true
}
else {
   //statements if both expression1 and expression2 result to false
}
```



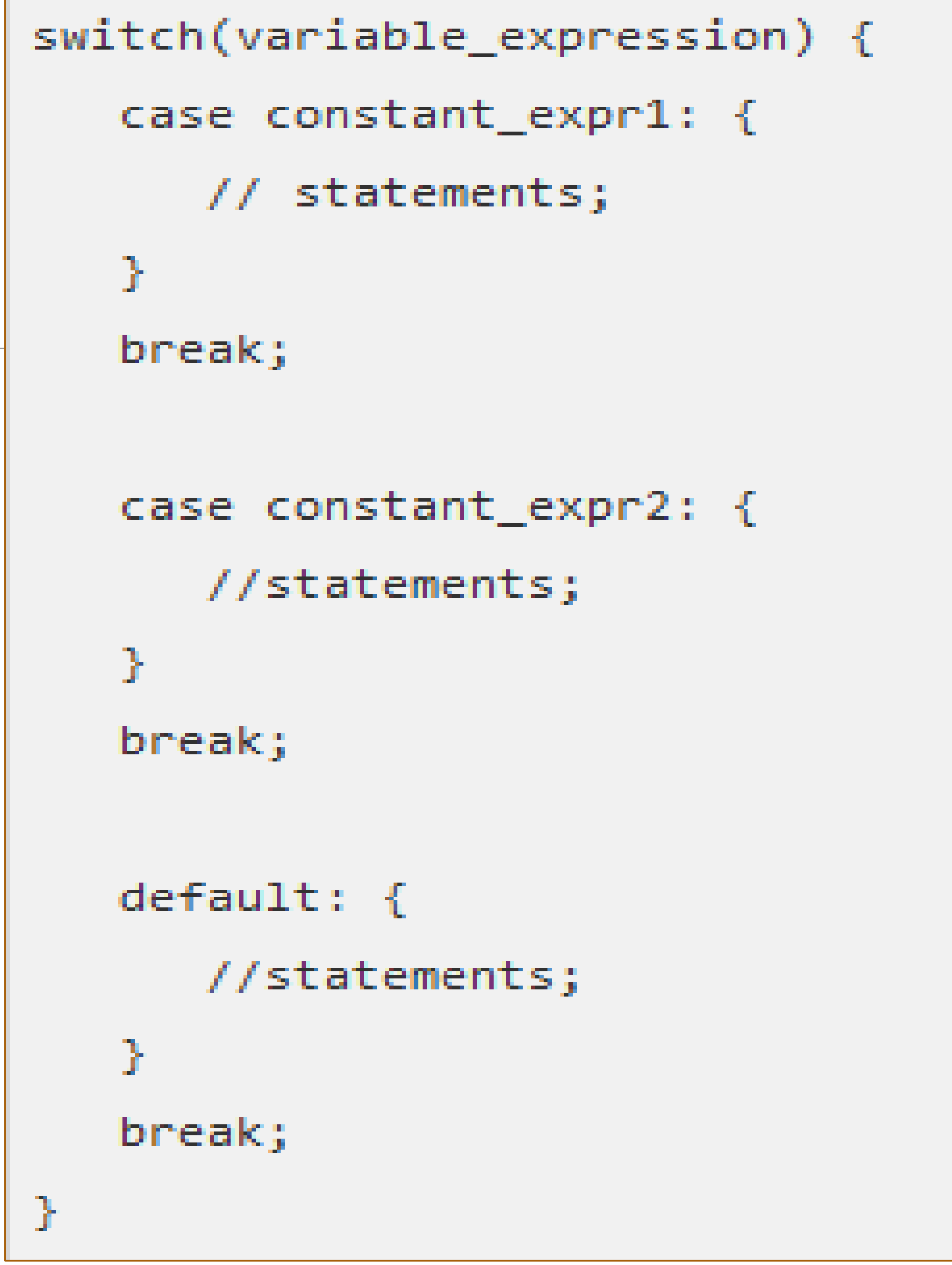
```
void main() {
   var num = 2;
   if(num > 0) {
      print("${num} is positive");
   }
   else if(num < 0) {
      print("${num} is negative");
   } else {
      print("${num} is neither positive nor negative");
   }
}
```



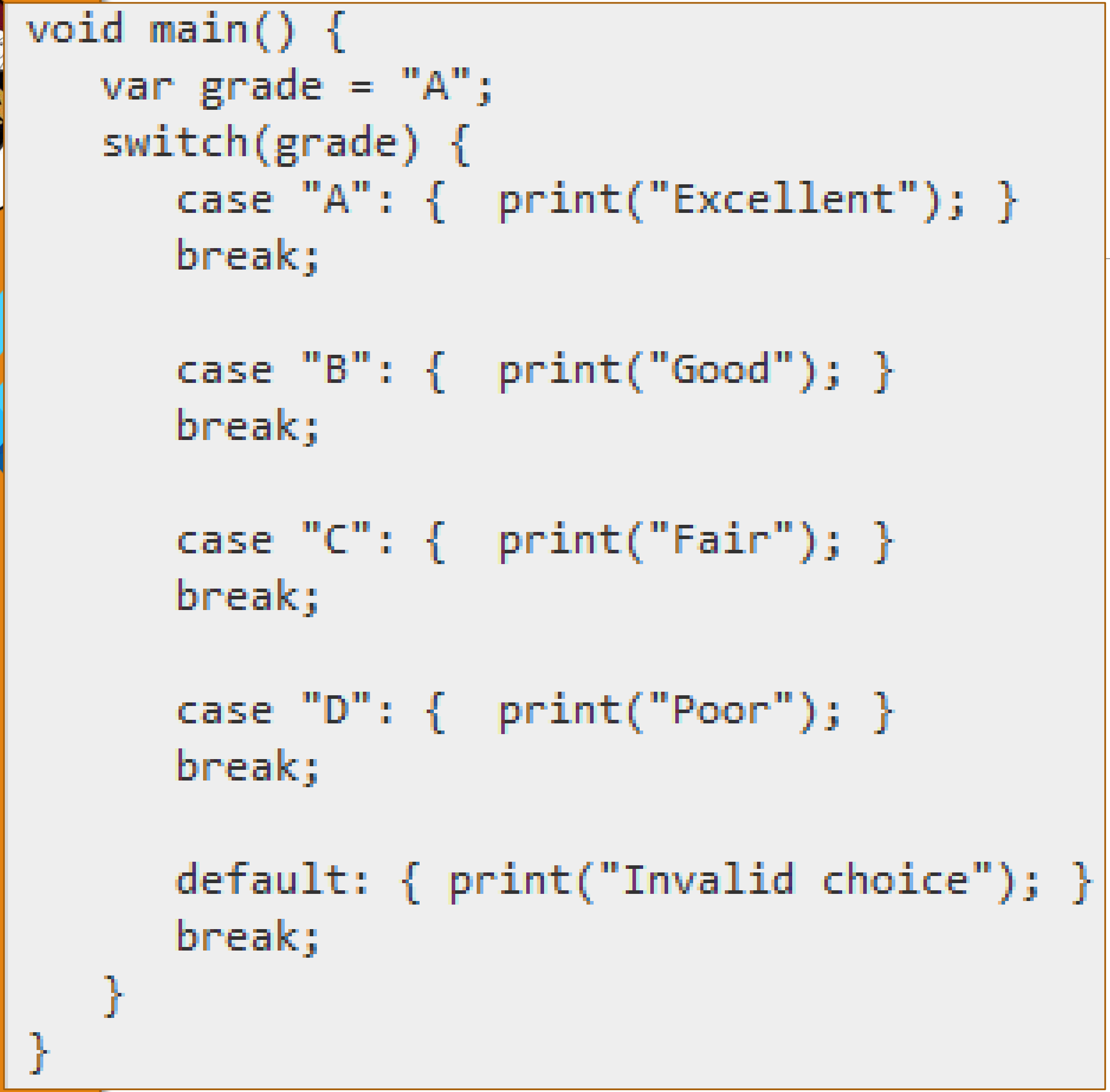
```
2 is positive
```

Switch Case Statement

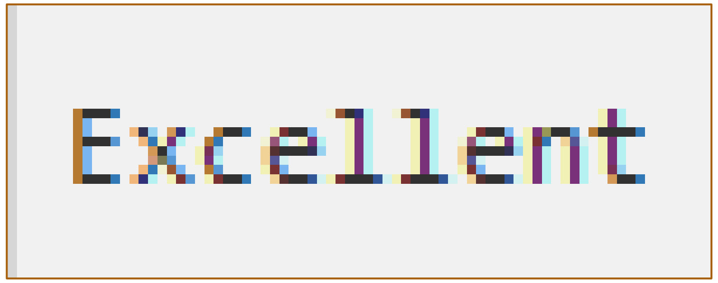
➢The switch statement evaluates an expression, matches the expression’s value to a case clause and executes the statements associated with that case.



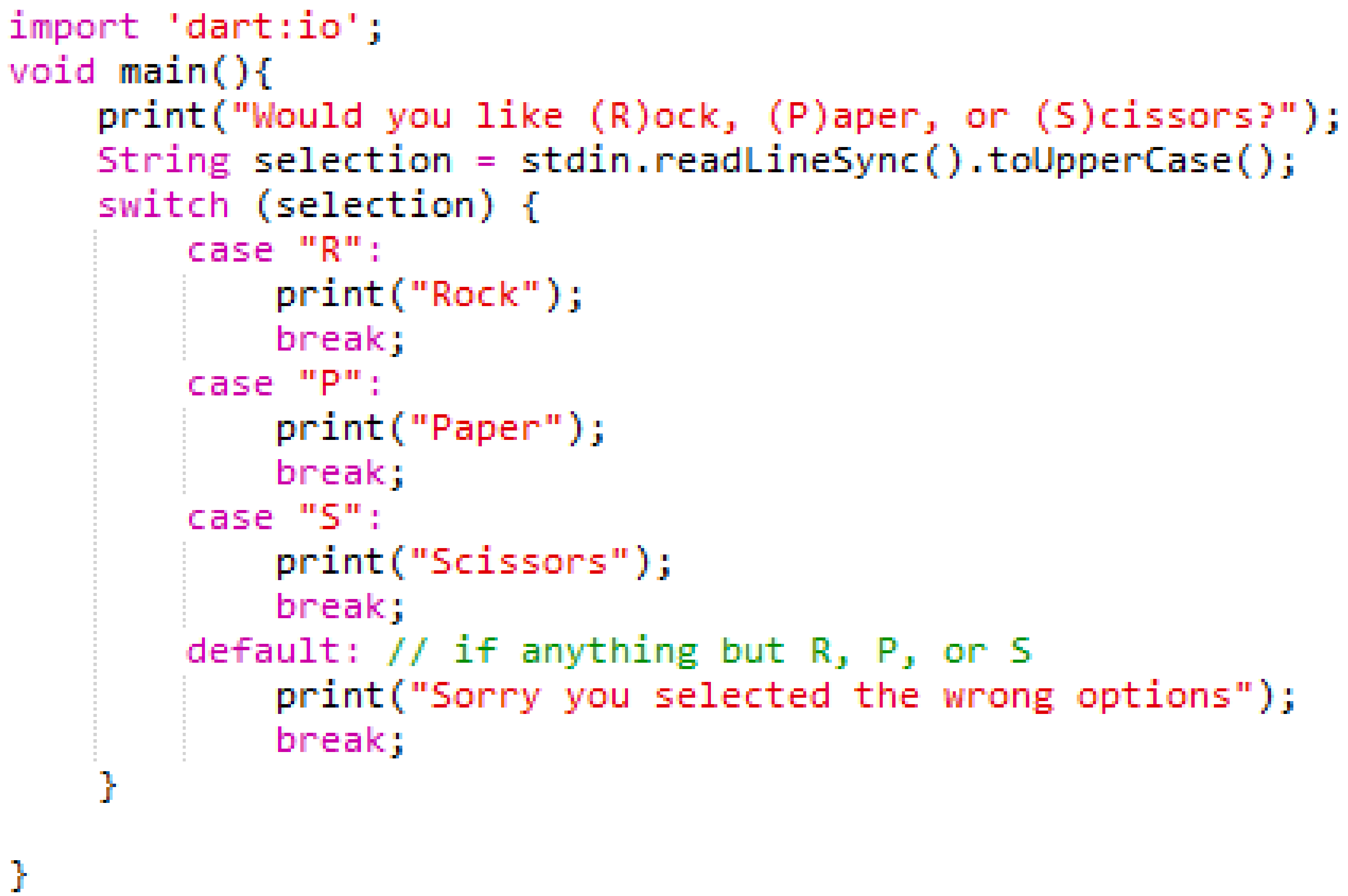
```
switch(variable_expression) {
   case constant_expr1: {
      // statements;
   }
   break;

   case constant_expr2: {
      //statements;
   }
   break;

   default: {
      //statements;
   }
   break;
}
```



```
void main() {
   var grade = "A";
   switch(grade) {
      case "A": {  print("Excellent"); }
      break;

      case "B": {  print("Good"); }
      break;

      case "C": {  print("Fair"); }
      break;

      case "D": {  print("Poor"); }
      break;

      default: { print("Invalid choice"); }
      break;
   }
}
```



```
Excellent
```



```dart
import 'dart:io';
void main(){
    print("Would you like (R)ock, (P)aper, or (S)cissors?");
    String selection = stdin.readLineSync().toUpperCase();
    switch (selection) {
        case "R":
            print("Rock");
            break;
        case "P":
            print("Paper");
            break;
        case "S":
            print("Scissors");
            break;
        default: // if anything but R, P, or S
            print("Sorry you selected the wrong options");
            break;
    }

}
```

Assignment

1. Read about Dart OOP
2. Prepare for hands-on exam

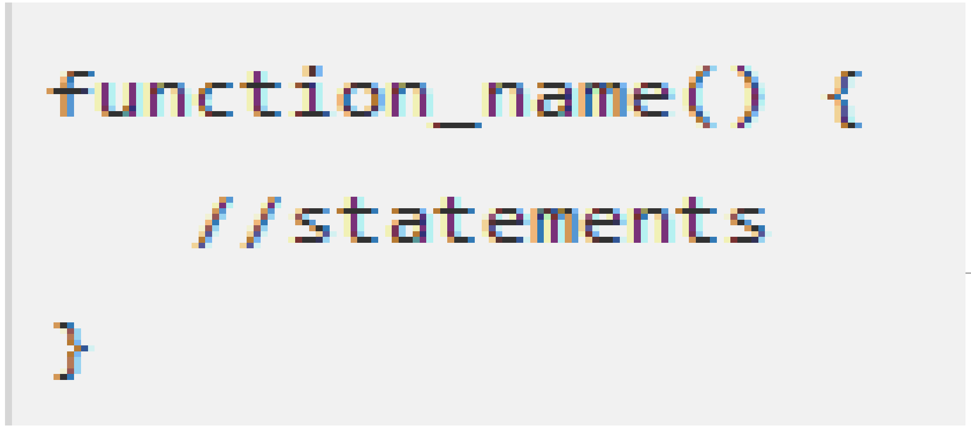
Dart Functions and OOP

Functions

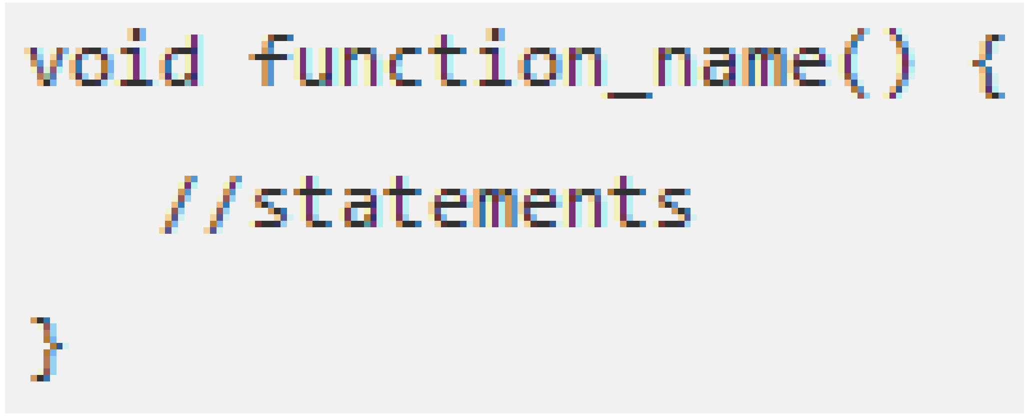
Functions are the building blocks of readable, maintainable, and reusable code. A function is a set of statements to perform a specific task. Functions organize the program into logical blocks of code. Once defined, functions may be called to access code.

```
main() {
  // main function's code goes here
}
```

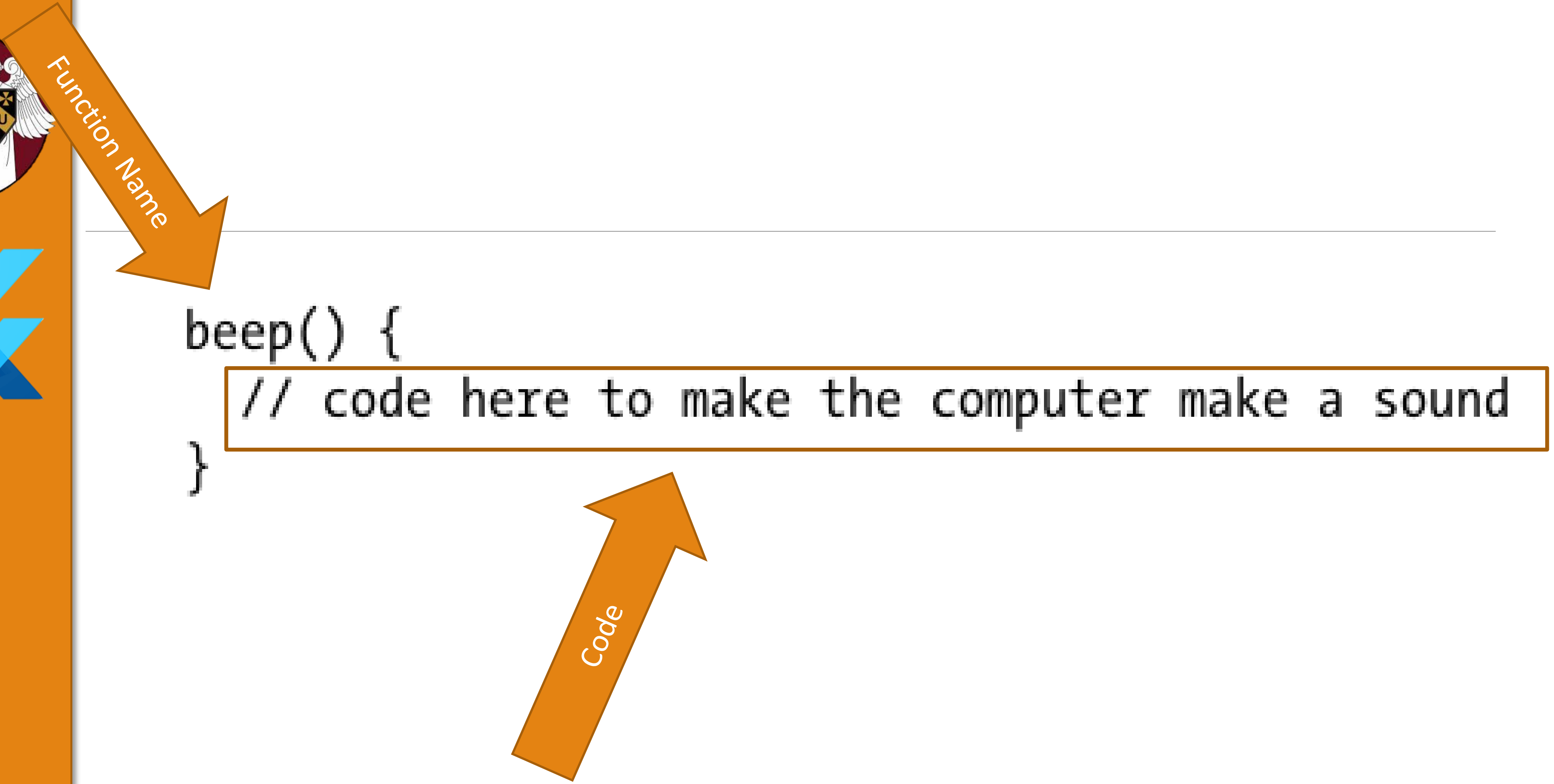
```
beep() {
  // code here to make the computer make a sound
}
```



```
function_name() {
   //statements
}
```



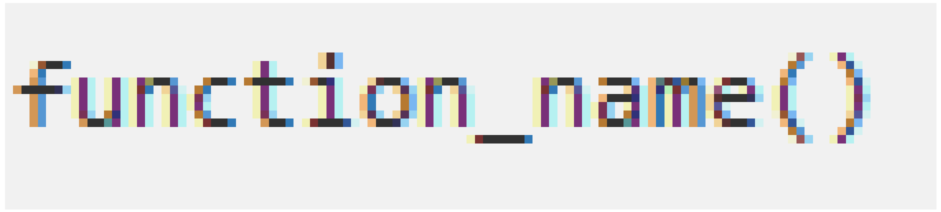
```
void function_name() {
   //statements
}
```



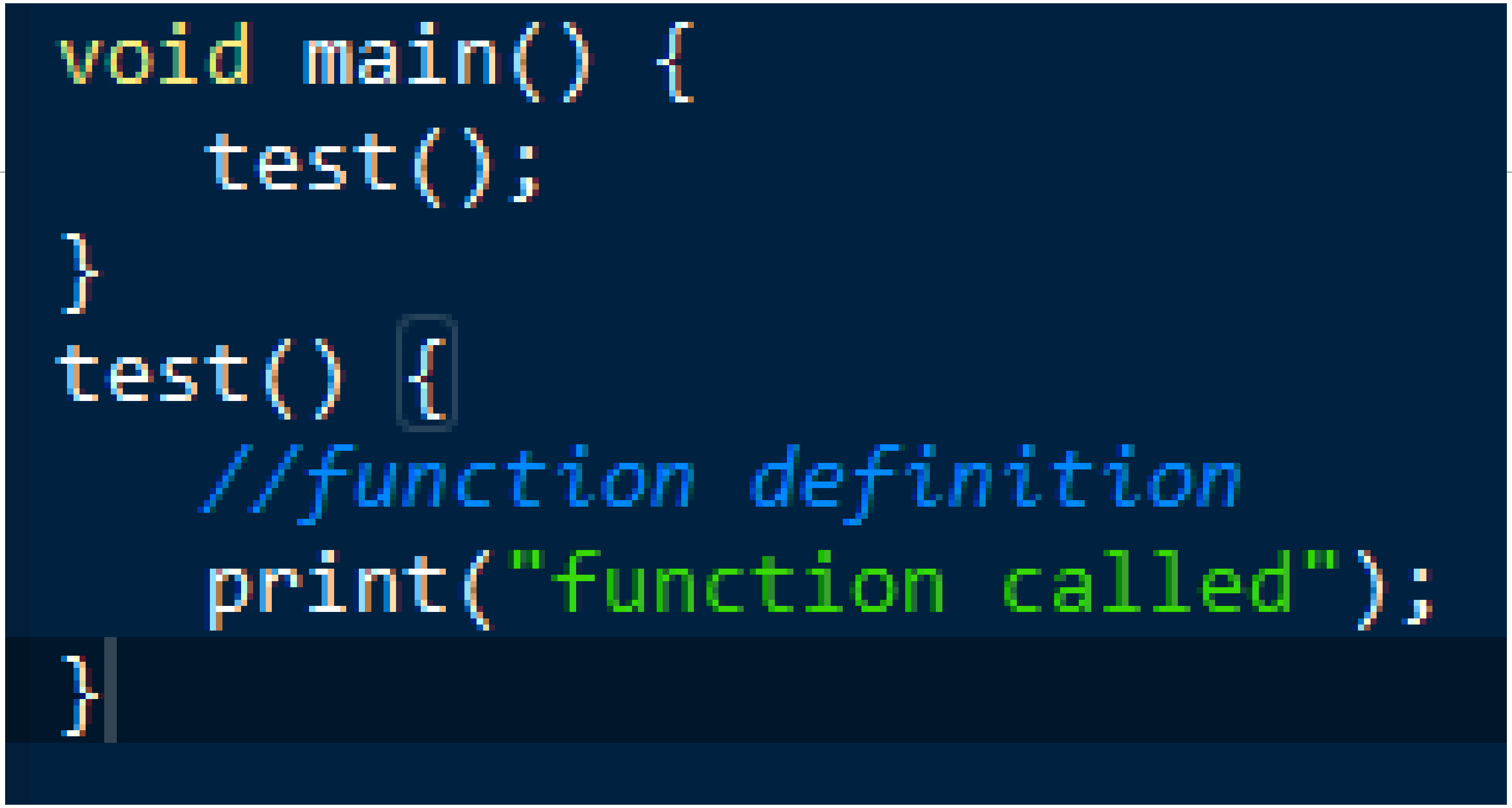
```
beep() {
  // code here to make the computer make a sound
}
```

Calling a Function

A function must be called to execute it. This process is termed as **function invocation**.



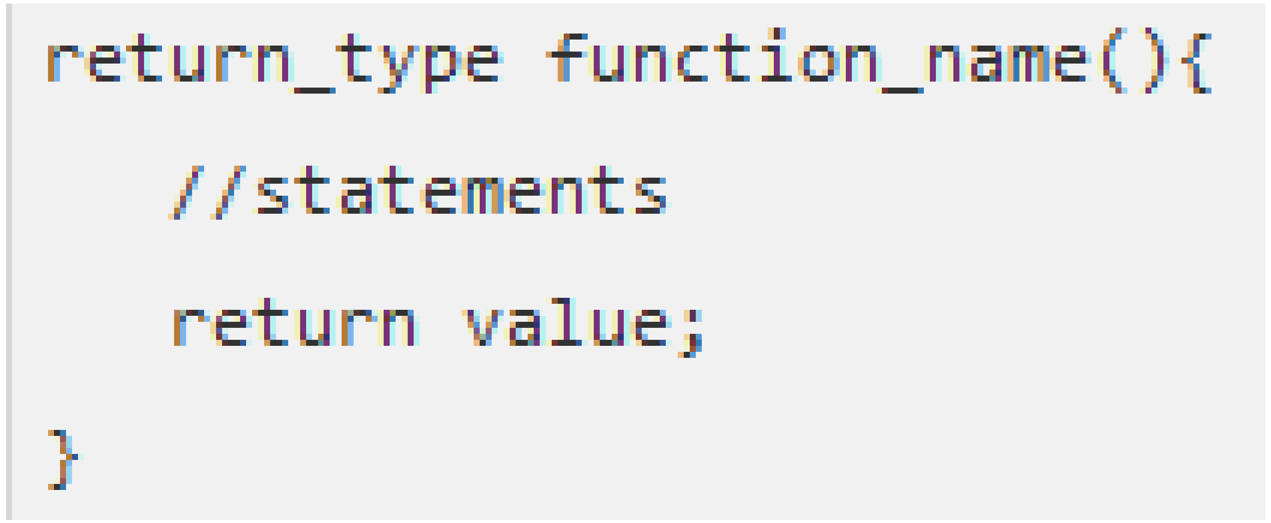
```
function_name()
```



```
void main() {
    test();
}
test() {
    //function definition
    print("function called");
}
```

Returning Function Values

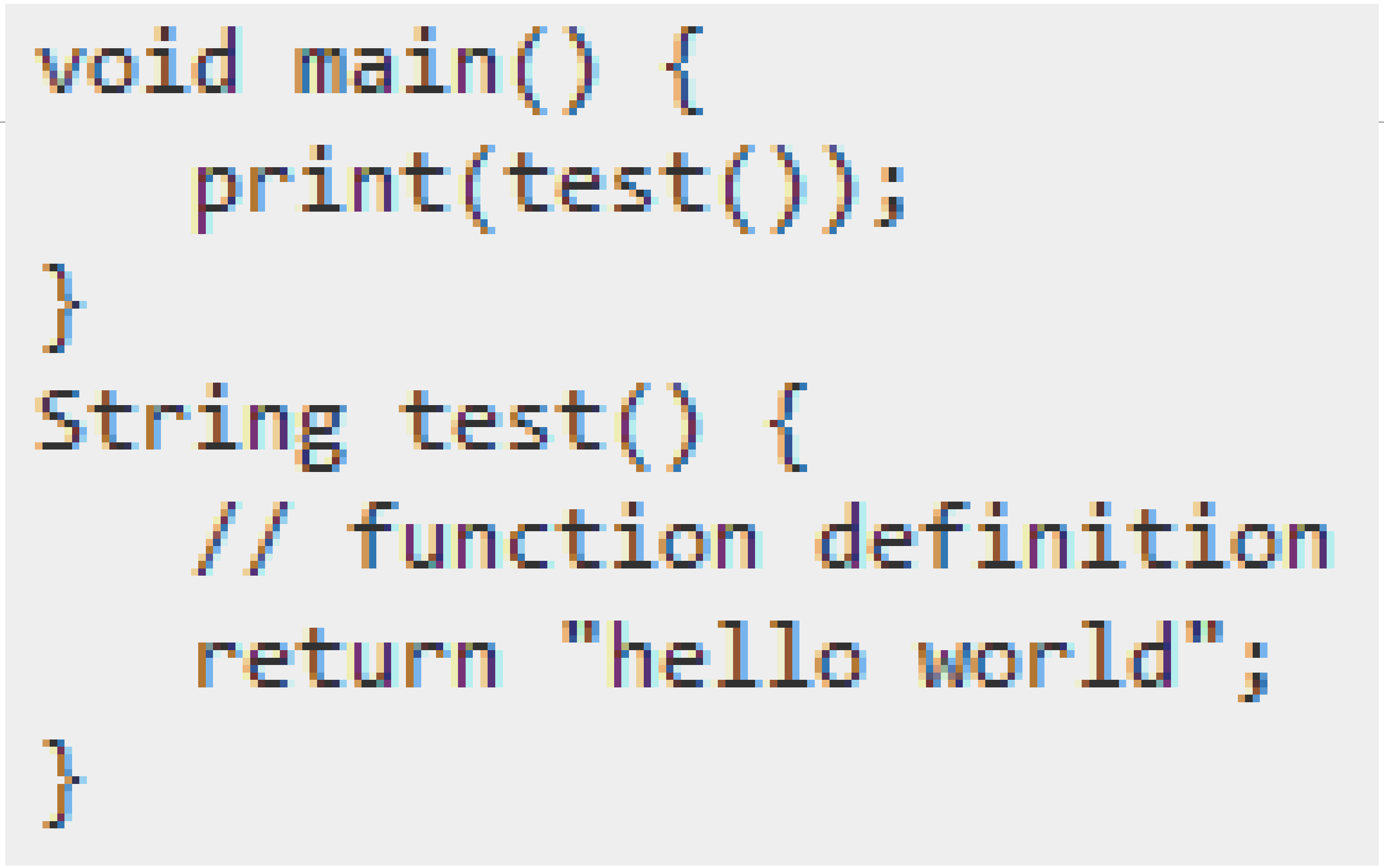
Functions may also return value along with the control, back to the caller. Such functions are called as **returning functions**.



```
return_type function_name(){

   //statements

   return value;

}
```

Returning Function Values

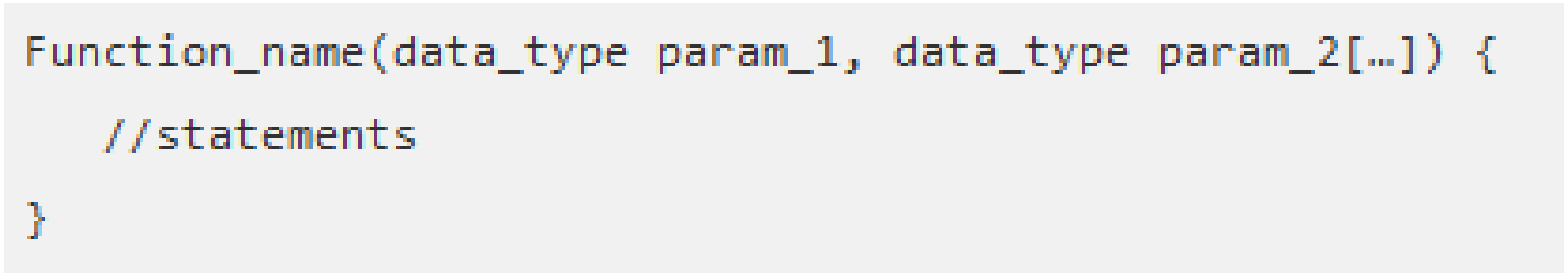
- The **return_type** can be any valid data type.
- The **return** statement is optional. I not specified the function returns null;
- The data type of the value returned must match the return type of the function.
- A function can return at the most one value. In other words, there can be only one return statement per function.



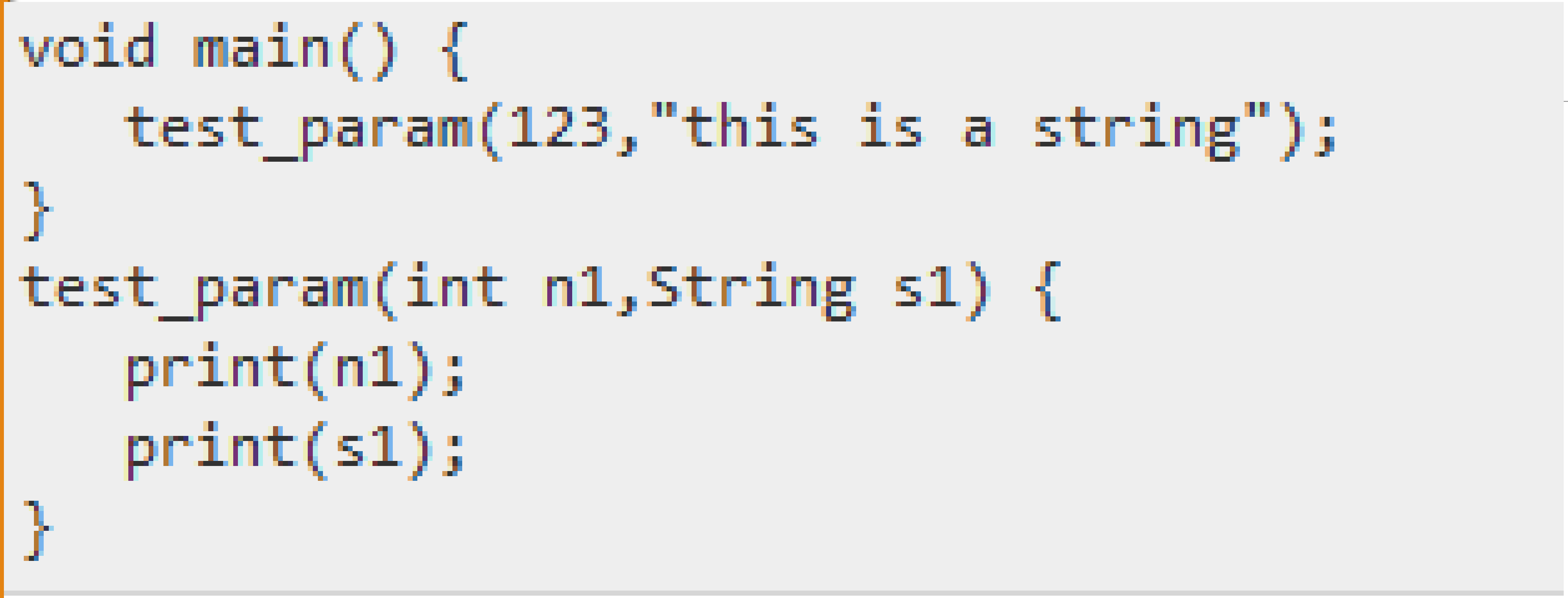
```dart
void main() {
   print(test());
}
String test() {
   // function definition
   return "hello world";
}
```

Parameterized Functions

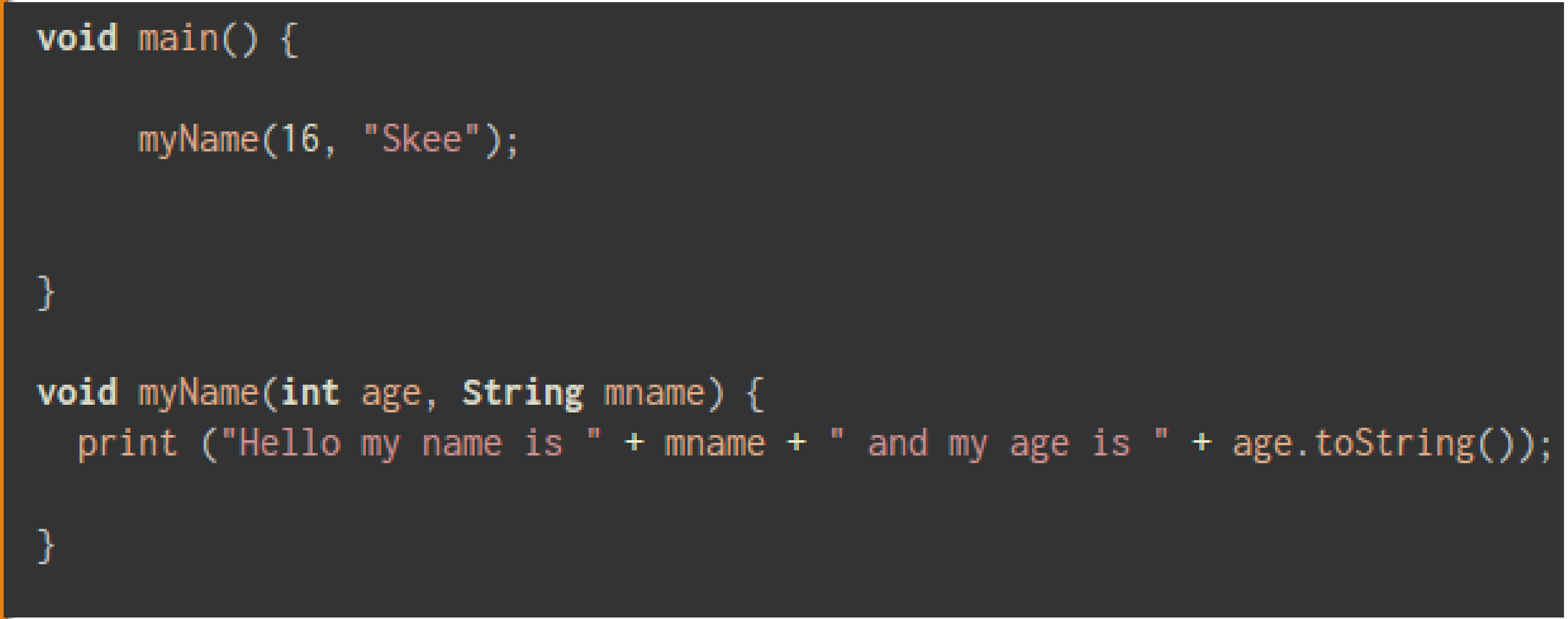
Functions can also have parameters. Parameters are a mechanism to pass values to functions. Parameters form a part of the function's signature. The parameter values are passed to the function during its invocation.



```
Function_name(data_type param_1, data_type param_2[…]) {
   //statements
}
```



```
void main() {
   test_param(123,"this is a string");
}
test_param(int n1,String s1) {
   print(n1);
   print(s1);
}
```



```
void main() {

    myName(16, "Skee");


}

void myName(int age, String mname) {
  print ("Hello my name is " + mname + " and my age is " + age.toString());

}
```

Exercise C3-1

Create a function name **findVolume** that computes for the a tank using the following formula

Volume = length * breadth * height

https://www.jdoodle.com/execute-dart-online

Object Oriented Programming

Object-Oriented Programming

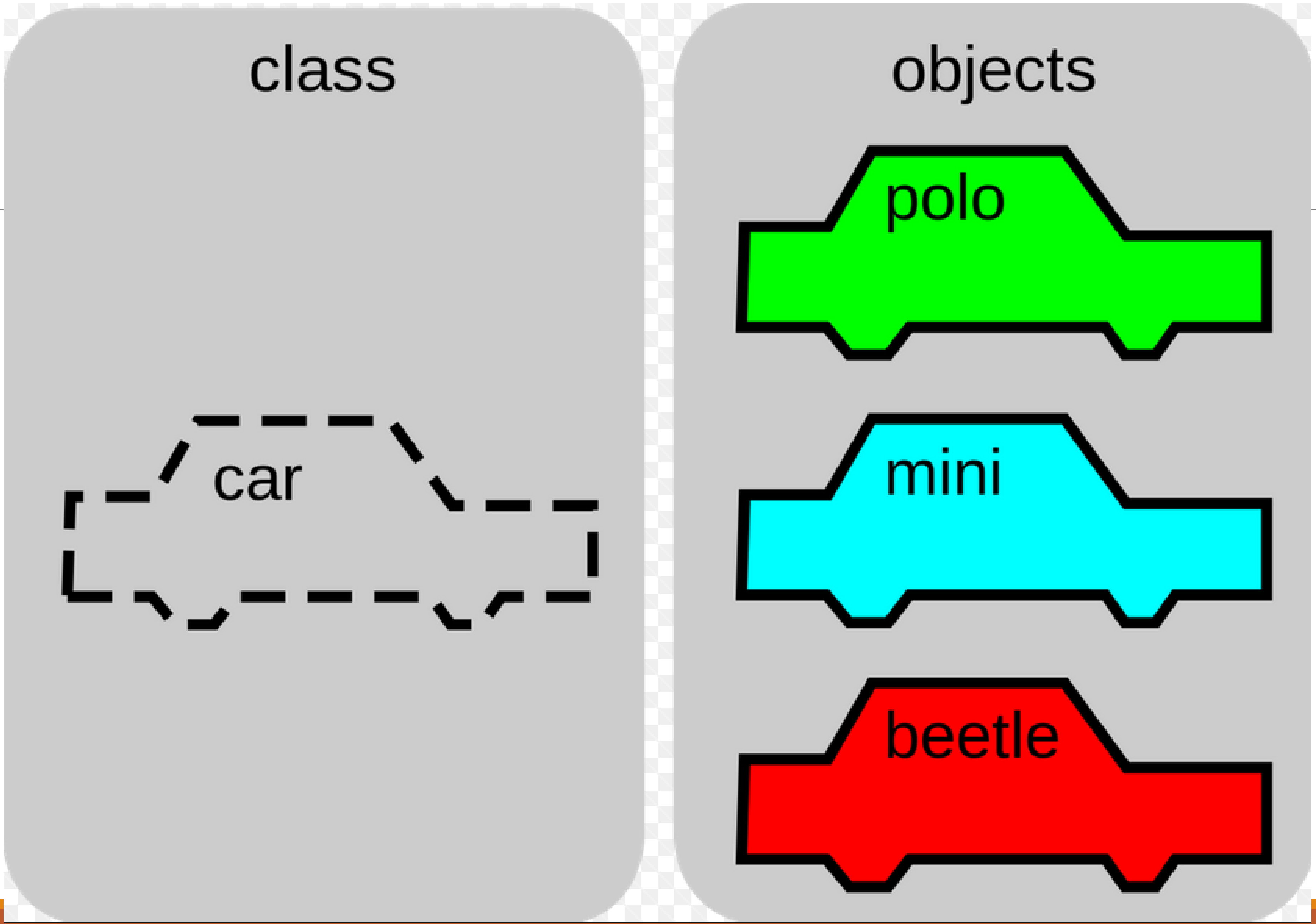
Object-oriented programming (OOP) is a programming language model in which programs are organized around data, or objects, rather than functions and logic.

What is an Object?

- An object is an abstraction meant to represent a component of a program. Objects can be created, destroyed, given attributes, and made to perform actions.
- Every object belongs to a **class**. The class of an object is its type. In other words, objects of the same class are the same type. Objects of the same class have the same instance variables and methods available to them (although the value of those instance variables may differ).

Class

a **class** is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).


class
car
objects
polo
mini
beetle

Declaring a Class

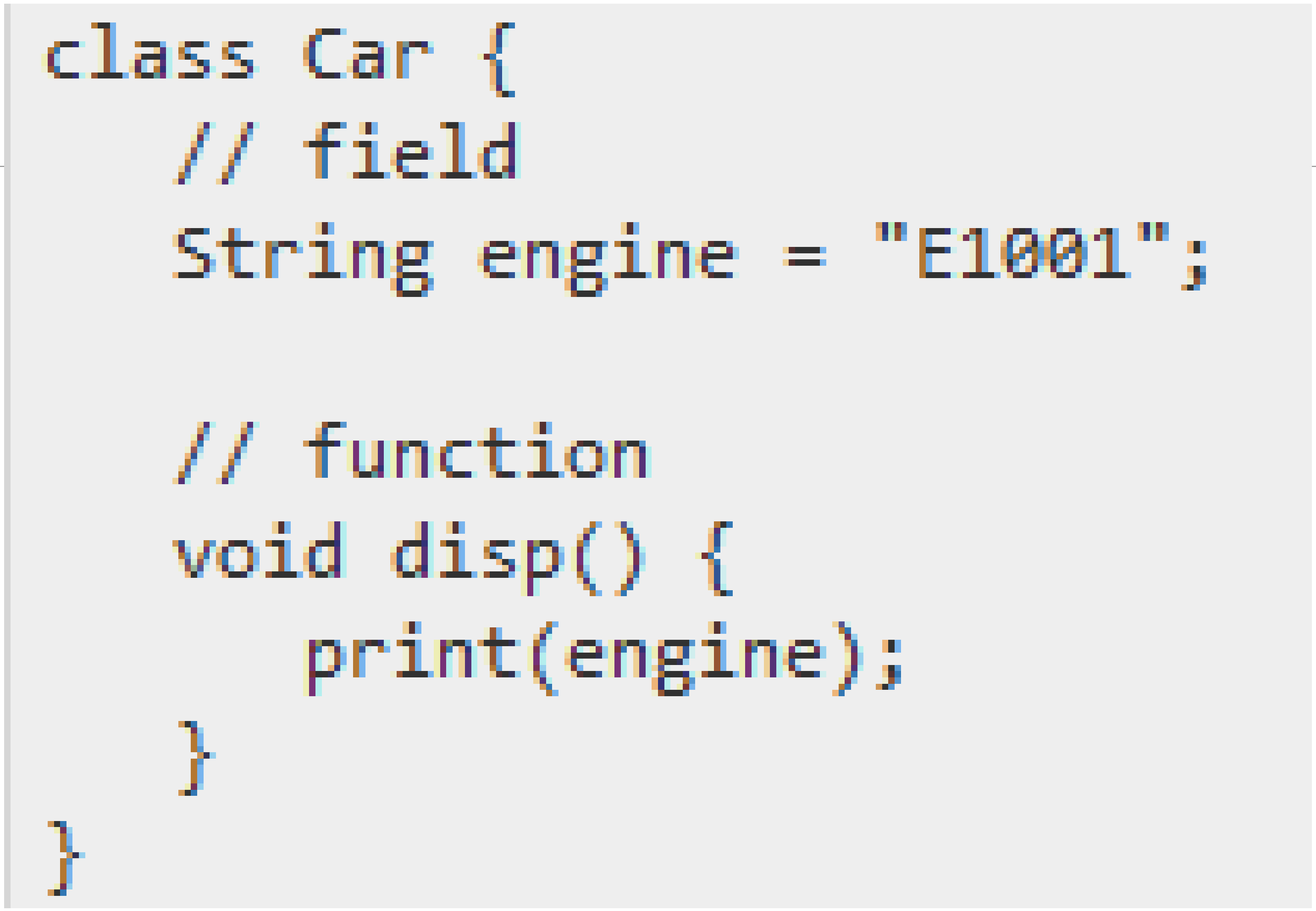
se the **class** keyword to declare a **class** in Dart. A class definition starts with the keyword class followed by the **class name**; and the class body enclosed by a pair of curly braces. The syntax for the same is given below

```
class class_name {
   <fields>
   <getters/setters>
   <constructors>
   <functions>
}
```

Declaring a Class

A class definition can include the following –

- **Fields** – A field is any variable declared in a class. Fields represent data pertaining to objects.
- **Setters and Getters** – Allows the program to initialize and retrieve the values of the fields of a class. A default getter/ setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.
- **Constructors** – responsible for allocating memory for the objects of the class.
- **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.



```
class Car {
   // field
   String engine = "E1001";

   // function
   void disp() {
      print(engine);
   }
}
```

Creating Instance of the class

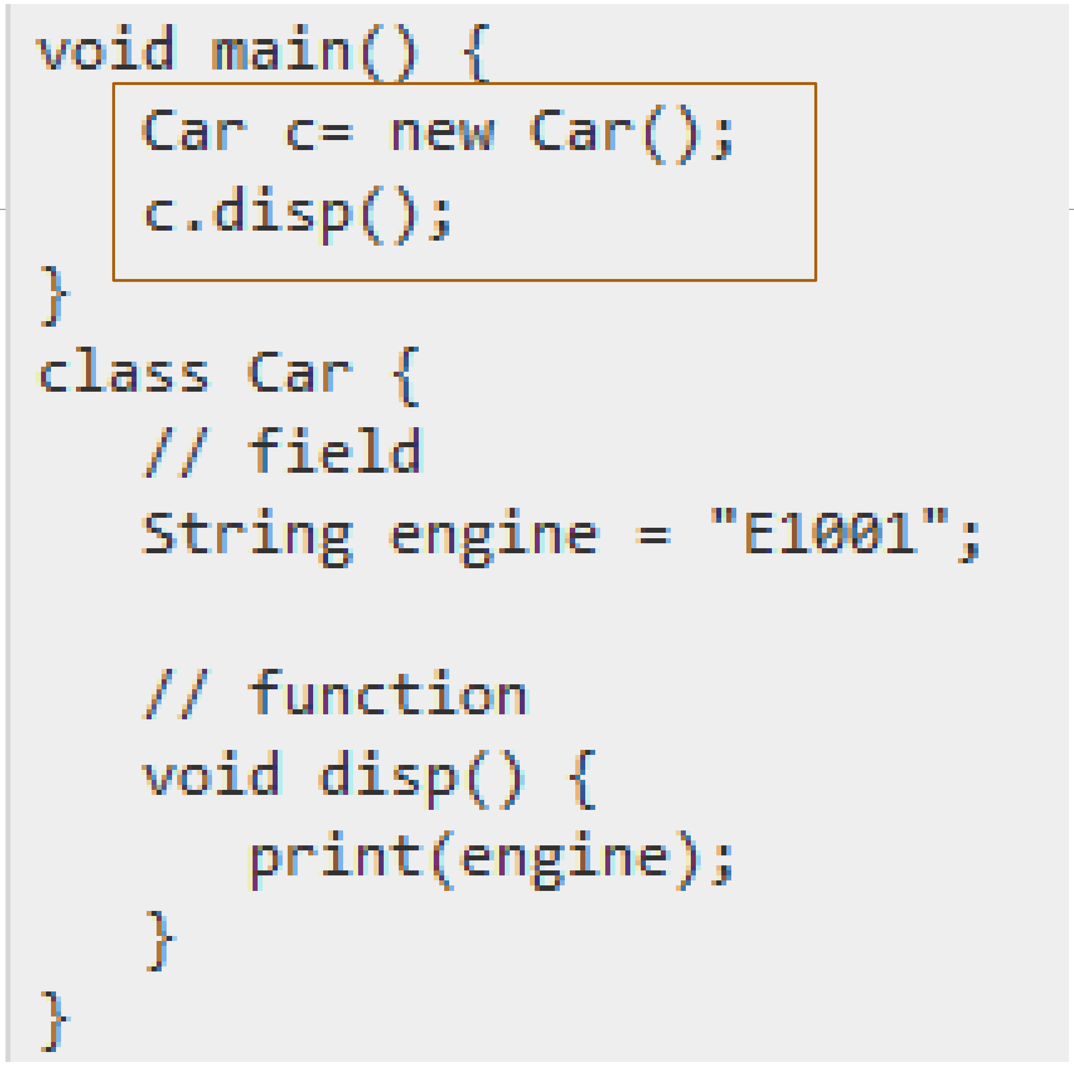
To create an instance of the class, use the **new** keyword followed by the class name. The syntax for the same is given below –

```
var object_name = new class_name([ arguments ])
```

The **new** keyword is responsible for instantiation.

The right-hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

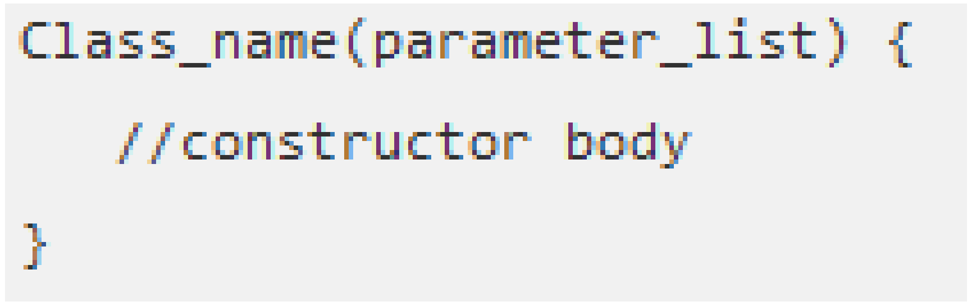
```
var obj = new Car("Engine 1")
```



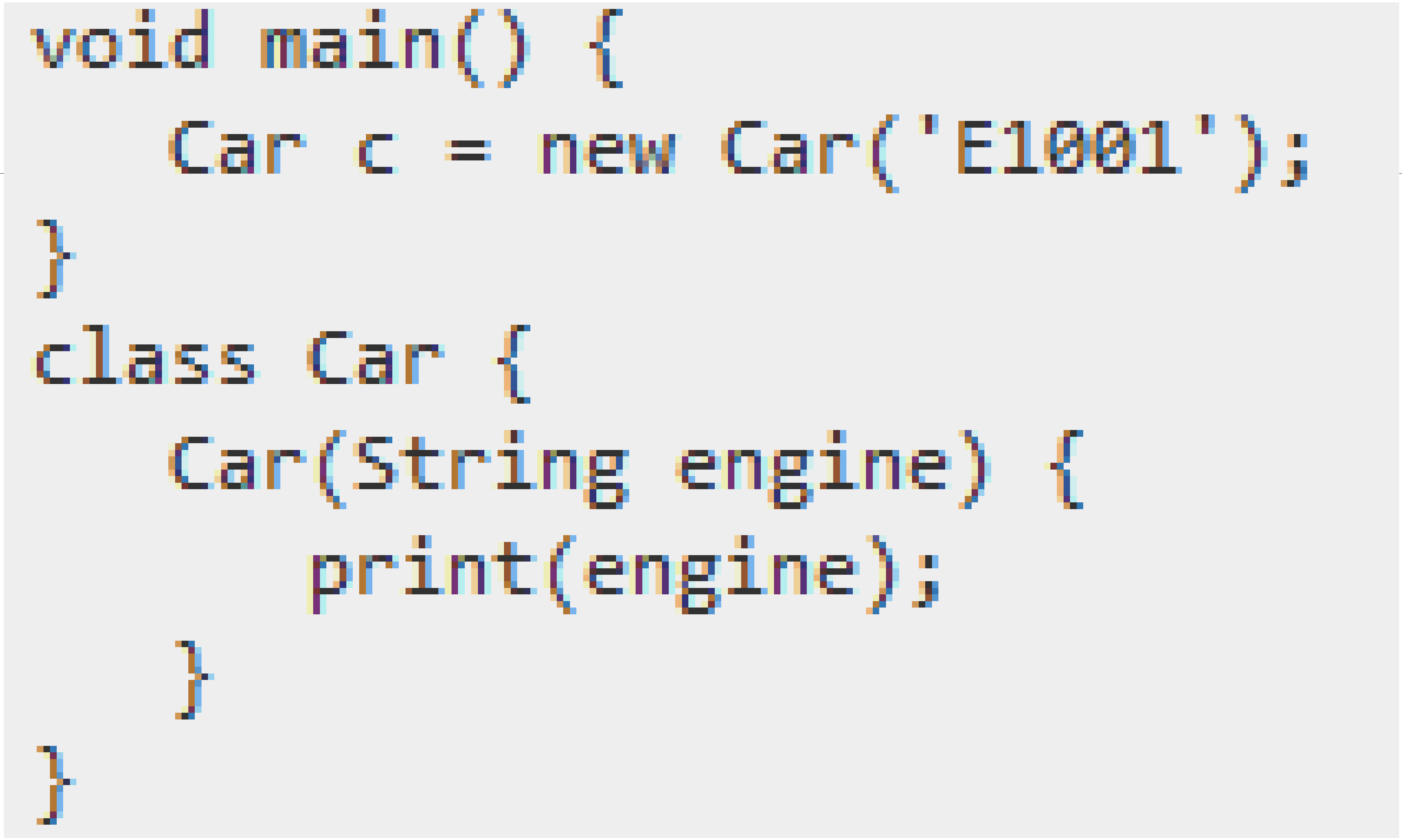
```
void main() {
    Car c= new Car();
    c.disp();
}
class Car {
    // field
    String engine = "E1001";

    // function
    void disp() {
        print(engine);
    }
}
```

Dart Constructors

A constructor is a special function of the class that is responsible for initializing the variables of the class. Dart defines a constructor with the same name as that of the class. A constructor is a function and hence can be parameterized. However, unlike a function, constructors cannot have a return type. If you don’t declare a constructor, a default **no-argument constructor** is provided for you.



```
Class_name(parameter_list) {

   //constructor body

}
```

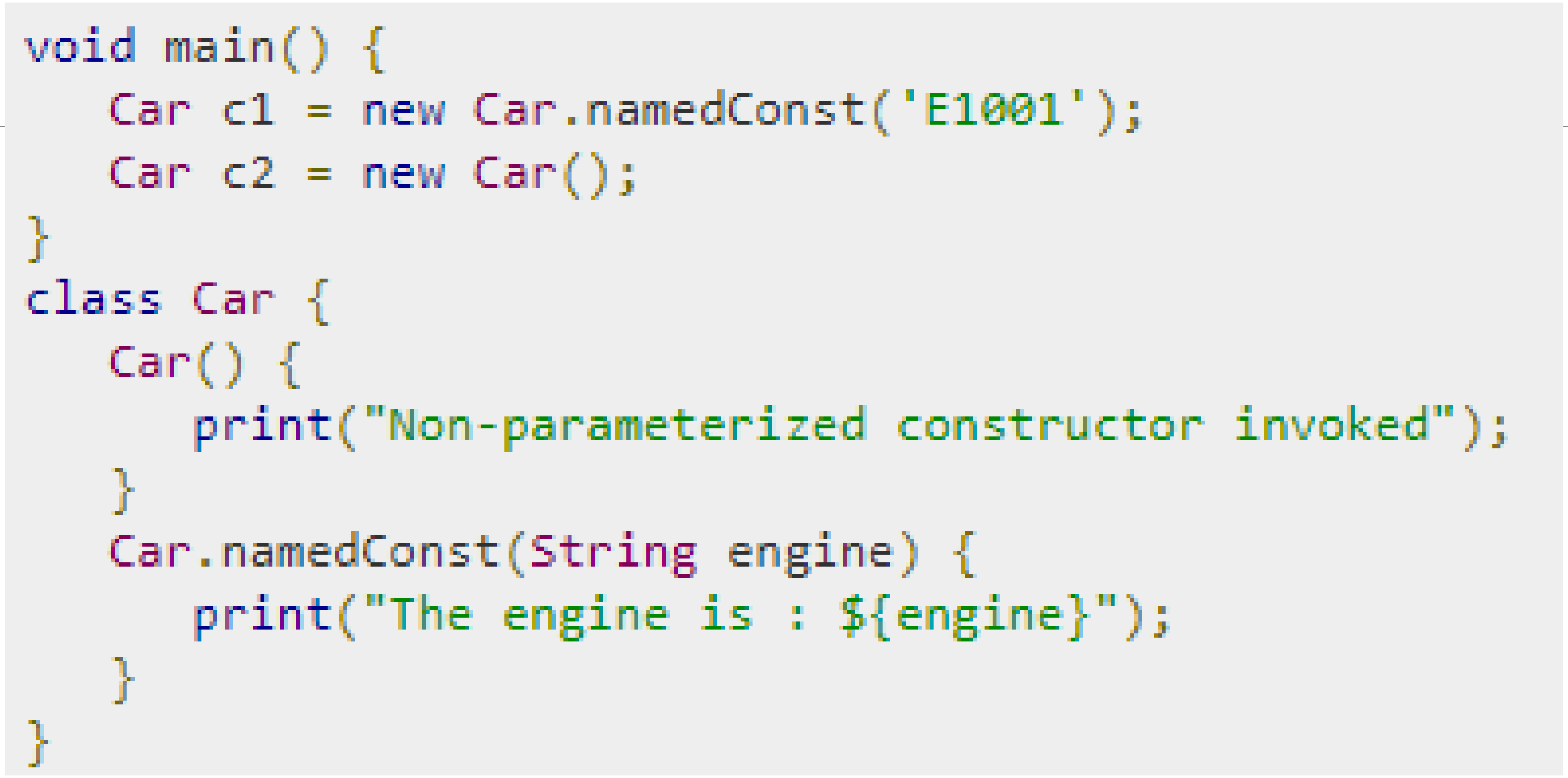


```
void main() {
   Car c = new Car('E1001');
}
class Car {
   Car(String engine) {
      print(engine);
   }
}
```

Named Constructors

Dart provides **named constructors** to enable a class define **multiple constructors**. The syntax of named constructors is as given below

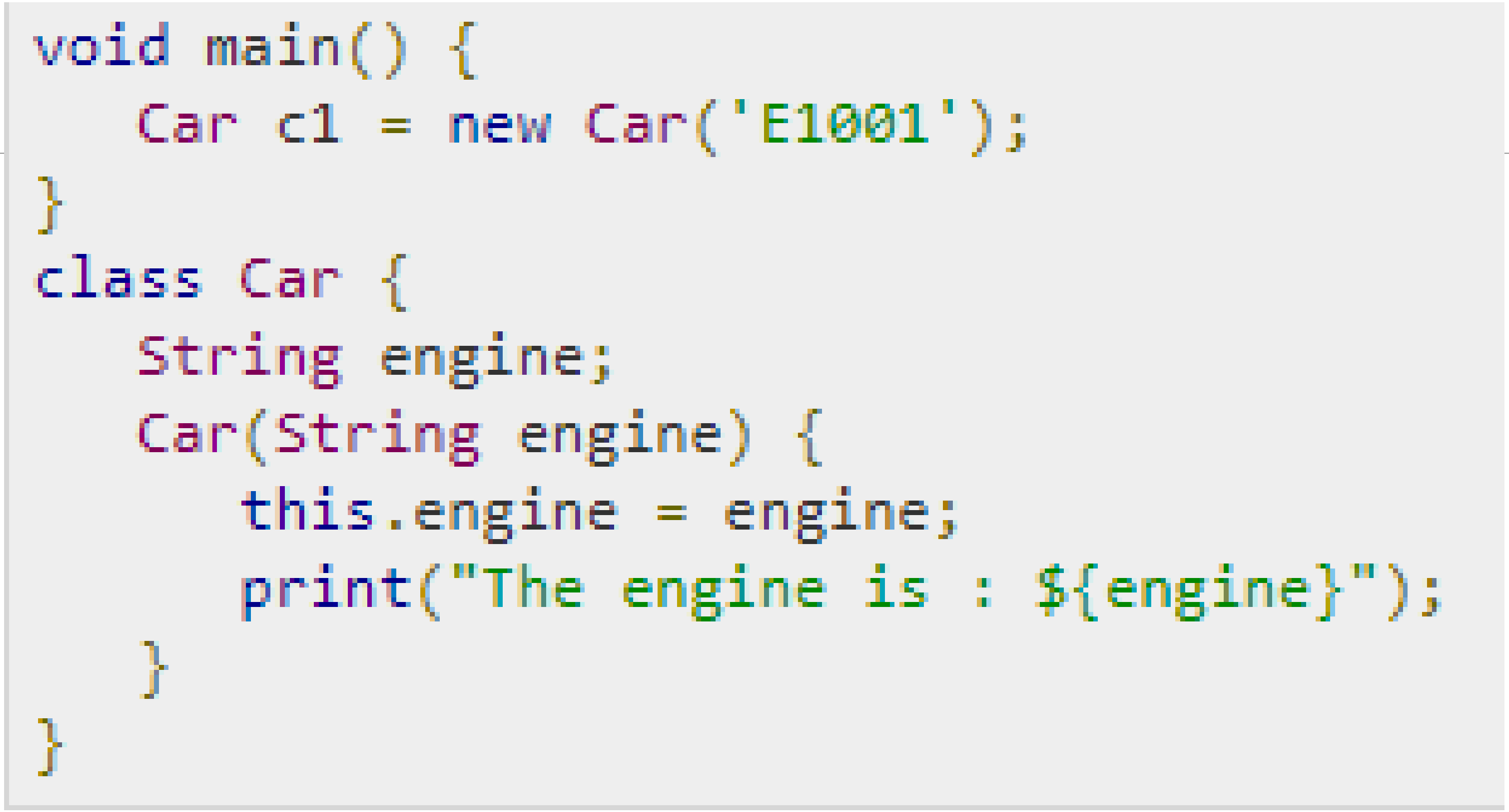
```
Class_name.constructor_name(param_list)
```



```
void main() {
   Car c1 = new Car.namedConst('E1001');
   Car c2 = new Car();
}
class Car {
   Car() {
      print("Non-parameterized constructor invoked");
   }
   Car.namedConst(String engine) {
      print("The engine is : ${engine}");
   }
}
```

The **this** Keyword

The **this** keyword refers to the current instance of the class. Here, the parameter name and the name of the class's field are the same. Hence to avoid ambiguity, the class's field is prefixed with the **this** keyword.



```dart
void main() {
   Car c1 = new Car('E1001');
}
class Car {
   String engine;
   Car(String engine) {
      this.engine = engine;
      print("The engine is : ${engine}");
   }
}
```

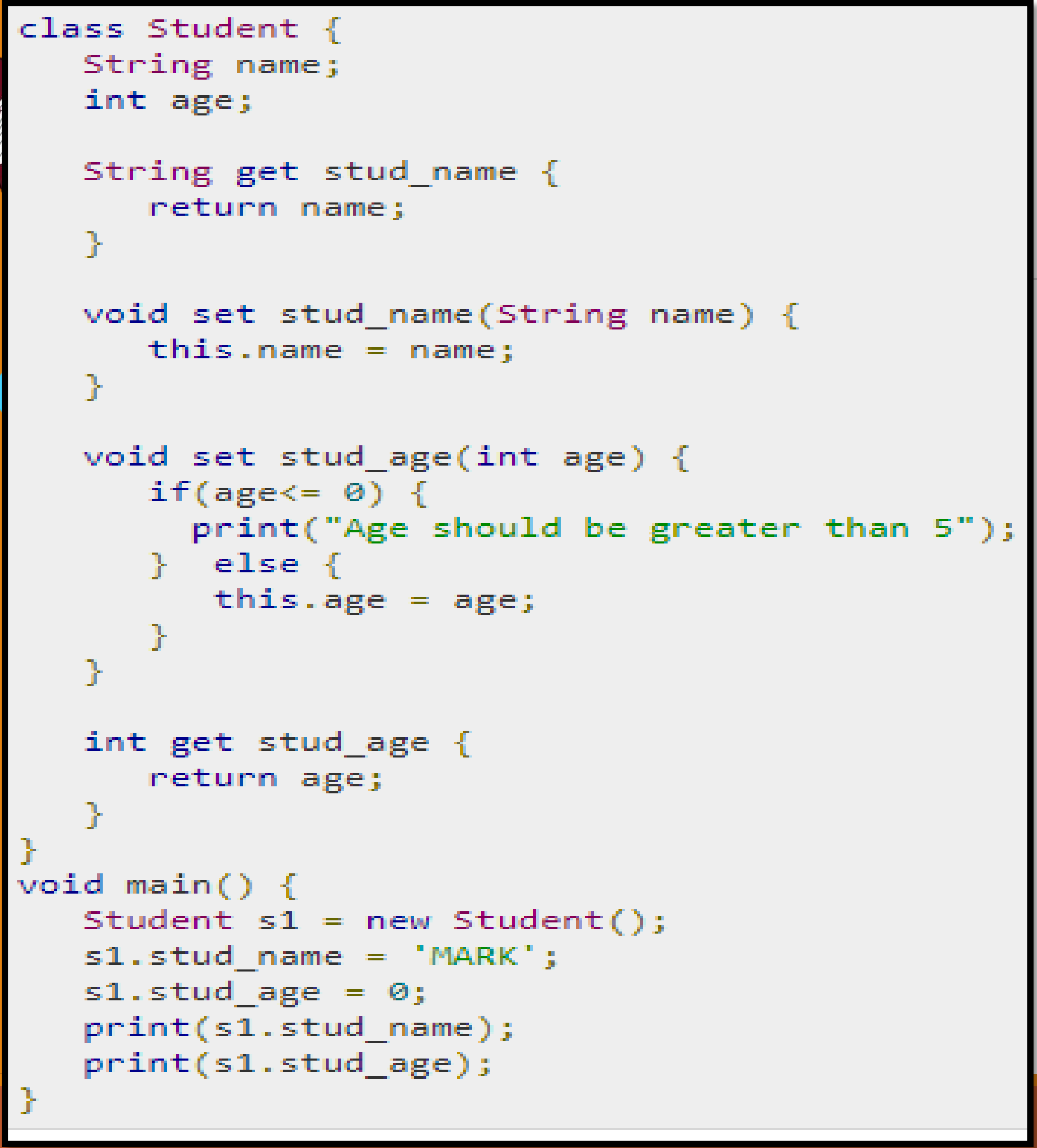
Dart Class – **Getters** and **Setters**

Getters and **Setters**, also called as **accessors** and **mutators**, allow the program to initialize and retrieve the values of class fields respectively. Getters or accessors are defined using the **get** keyword. Setters or mutators are defined using the **set** keyword.

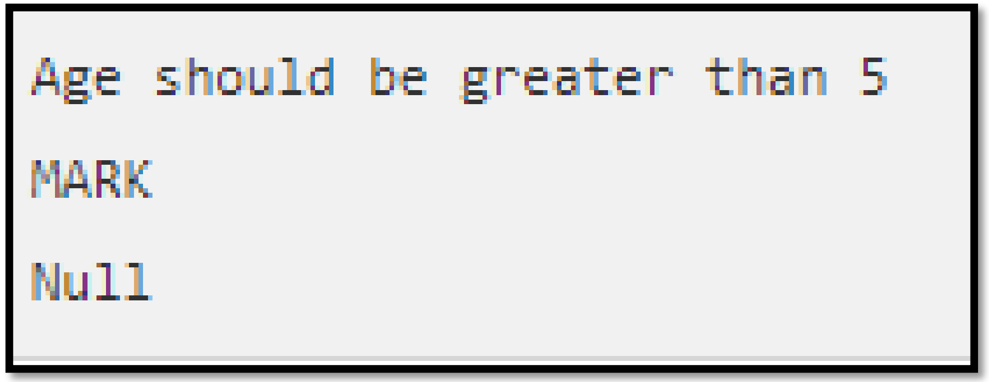
Exercise C3-2

Create an class name Person with the following properties firstName, lastName, Age, Address.

Call each value of the properties using the following methods or functions: myAge, myName and myAddress,myCompleteInformation

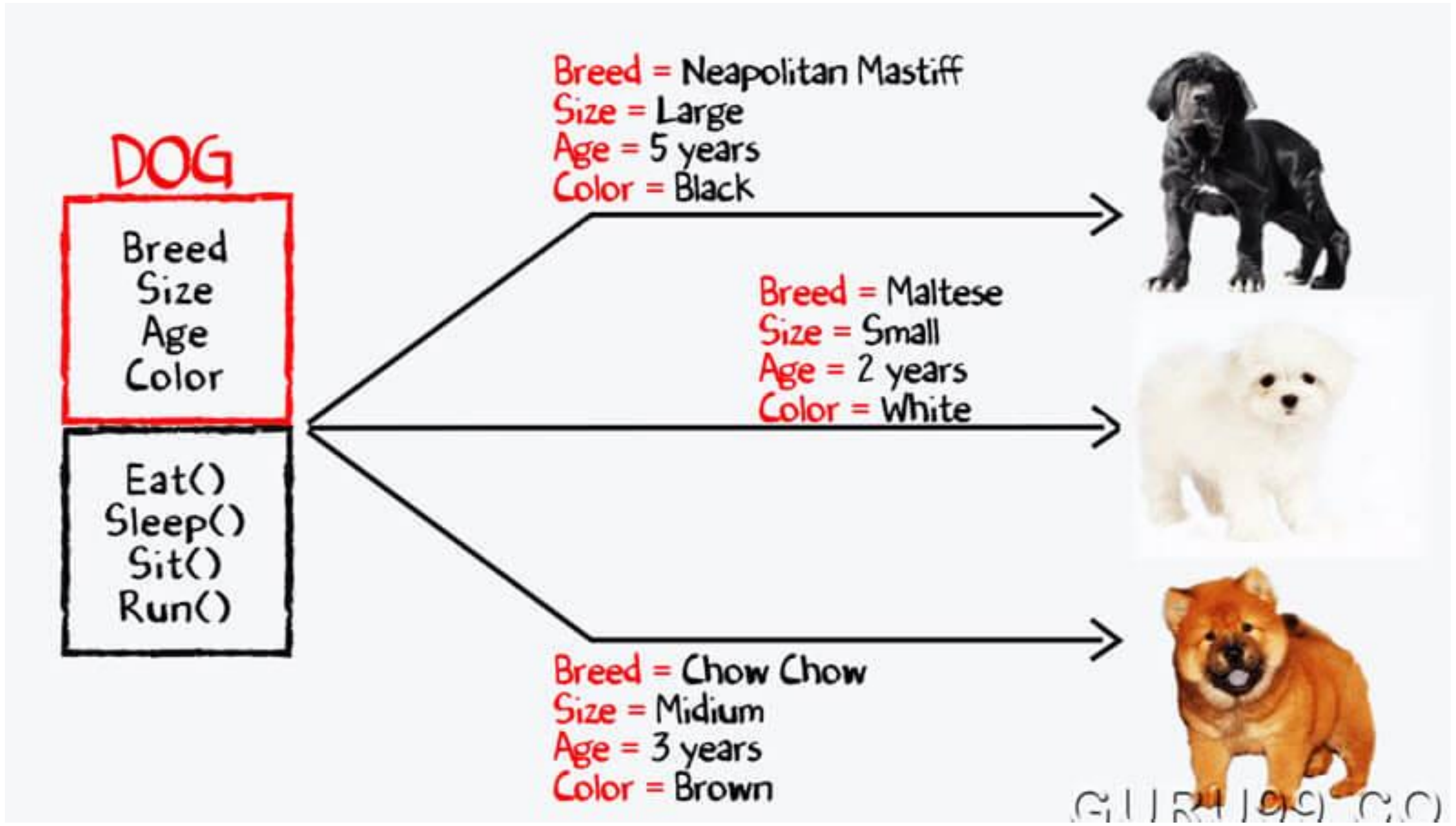


```
class Student {
   String name;
   int age;

   String get stud_name {
      return name;
   }

   void set stud_name(String name) {
      this.name = name;
   }

   void set stud_age(int age) {
      if(age<= 0) {
        print("Age should be greater than 5");
      }  else {
         this.age = age;
      }
   }

   int get stud_age {
      return age;
   }
}
void main() {
   Student s1 = new Student();
   s1.stud_name = 'MARK';
   s1.stud_age = 0;
   print(s1.stud_name);
   print(s1.stud_age);
}
```



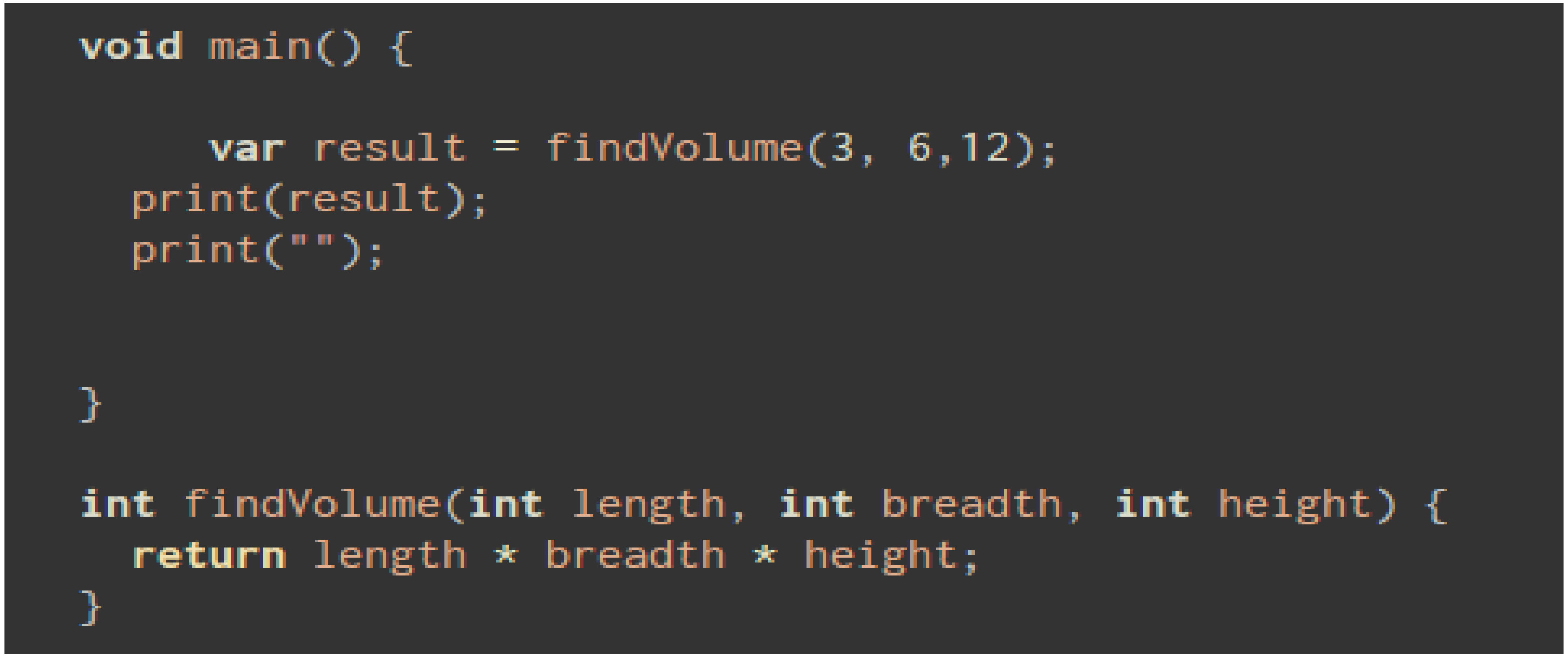
```
Age should be greater than 5

MARK

Null
```

Hands-On Exam



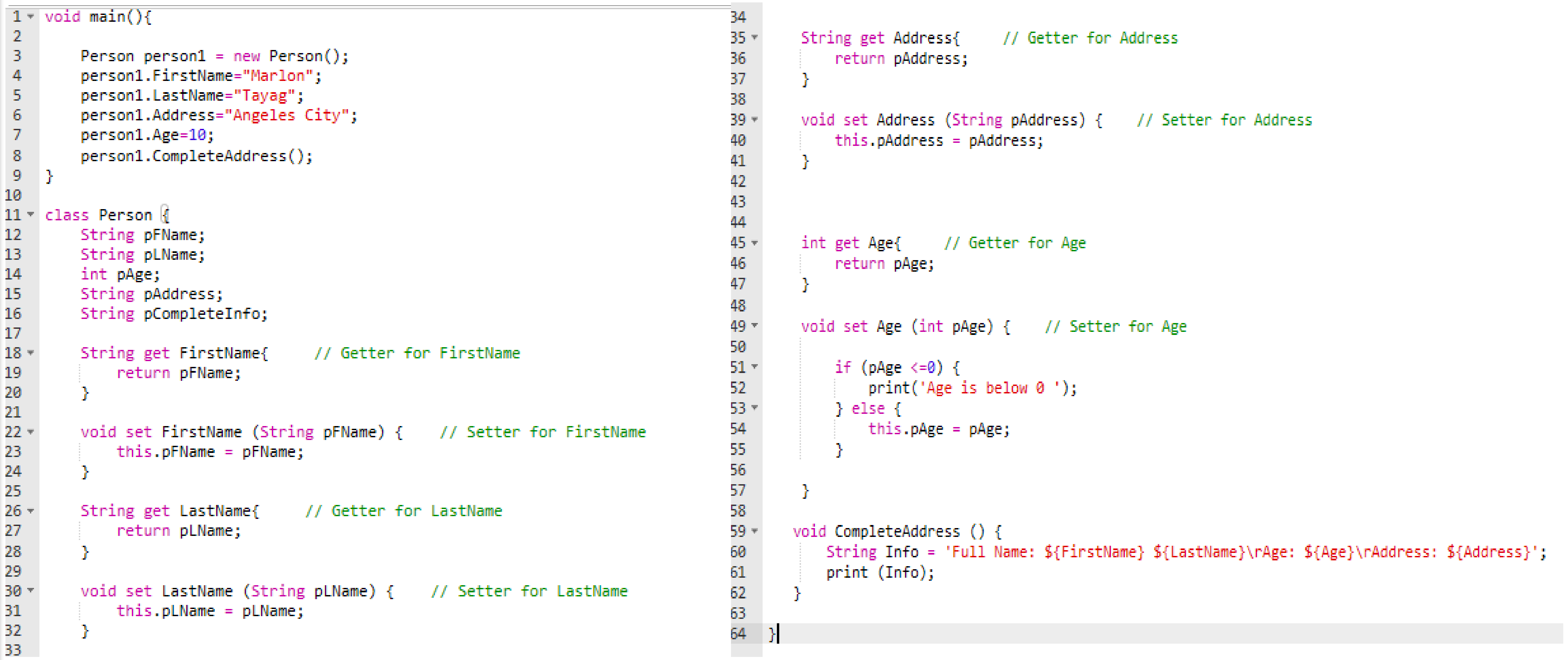
Solutions For Activity

Solution C3-1



```
void main() {

     var result = findVolume(3, 6,12);
  print(result);
  print("");


}

int findVolume(int length, int breadth, int height) {
  return length * breadth * height;
}
```

Solution C3-2



```
void main(){

    Person person1 = new Person();
    person1.FirstName="Marlon";
    person1.LastName="Tayag";
    person1.Address="Angeles City";
    person1.Age=10;
    person1.CompleteAddress();
}

class Person {
    String pFName;
    String pLName;
    int pAge;
    String pAddress;
    String pCompleteInfo;

    String get FirstName{     // Getter for FirstName
        return pFName;
    }

    void set FirstName (String pFName) {    // Setter for FirstName
        this.pFName = pFName;
    }

    String get LastName{     // Getter for LastName
        return pLName;
    }

    void set LastName (String pLName) {    // Setter for LastName
        this.pLName = pLName;
    }


    String get Address{     // Getter for Address
        return pAddress;
    }

    void set Address (String pAddress) {    // Setter for Address
        this.pAddress = pAddress;
    }



    int get Age{     // Getter for Age
        return pAge;
    }

    void set Age (int pAge) {    // Setter for Age

        if (pAge <=0) {
            print('Age is below 0 ');
        } else {
            this.pAge = pAge;
        }

    }

   void CompleteAddress () {
       String Info = 'Full Name: ${FirstName} ${LastName}\rAge: ${Age}\rAddress: ${Address}';
       print (Info);
   }

}
```