Mobile Application Development

Week 2. Introduction to Dart language

Joon-Woo Lee

School of Computer Science and Engineering
College of Software
Chung-Ang University

Term project team building

- The total number of students is 81.
- The number of teams will be 21.
  - 18 teams will include 4 members.
  - 3 teams will include 3 members.
  - 18 * 4 + 3 * 3 = 81
- If there are students who want to make a 3-member team, please contact by e-mail.
- If you have made a complete team, one of the team members should send an e-mail for notifying the team members.
- If you have made a non-complete team, you can send an e-mail with this non-complete team, then I will complete these teams.

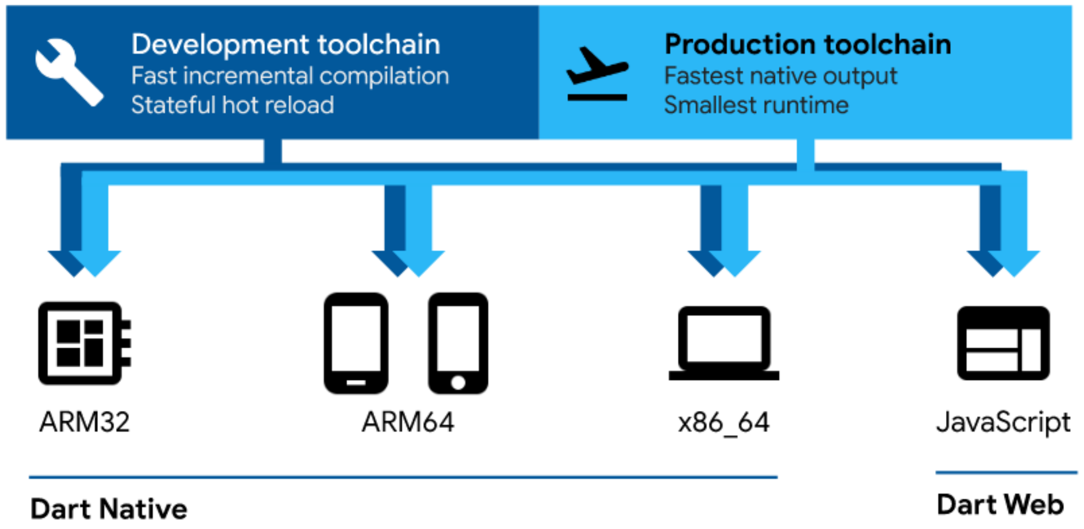
Dart Programming Language

Dart language

- The only programming language Flutter uses is **Dart** programming language.
- Dart is a programming language developed by **Google** for web and mobile programming**.**
- Dart is very similar to other OOP languages!
  - Java, C++, Python, ...

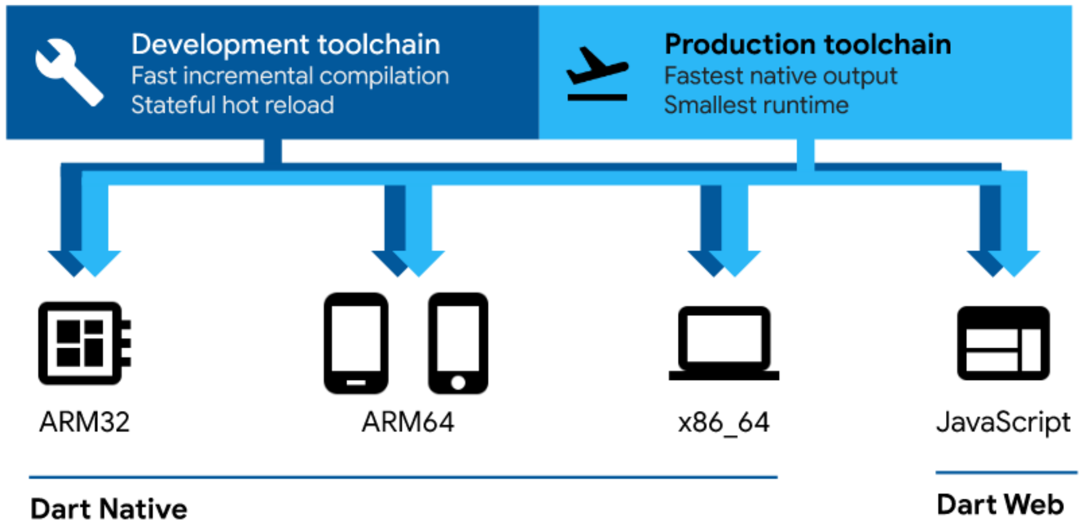
Why Flutter use Dart? (Compilation method)

- Dart supports both two compilation methods.
  - **Just-In-Time (JIT)** compiler: it converts source code into native machine code just before program execution.
    - ✓ Fast development cycle! (Hot reload)
  - **Ahead-Of-Time (AOT)** compiler: it compiles source code before it is "delivered" to whatever runtime environment runs the code.
    - ✓ Great performance of applications!



Why Flutter use Dart? (Compilation destination)

- Dart can be compiled into **native machine code**, which makes the applications fast.
  - JavaScript cannot be compiled into native machine code, so the resultant applications is not so fast.
- For web applications, Dart also supports compilation into **JavaScript**.



Why Flutter use Dart? (Google has it!)

- **Google** has been involved a big lawsuit by **Oracle** with the use of **Java API** in Android mobile development.
  - Google won this lawsuit from this 10-year patent dispute.
- Many expert analyzes that Google want to be free from this **patent dispute** for programming language.
- Google can also feel free to develop Dart programming language in the way that Flutter needs.

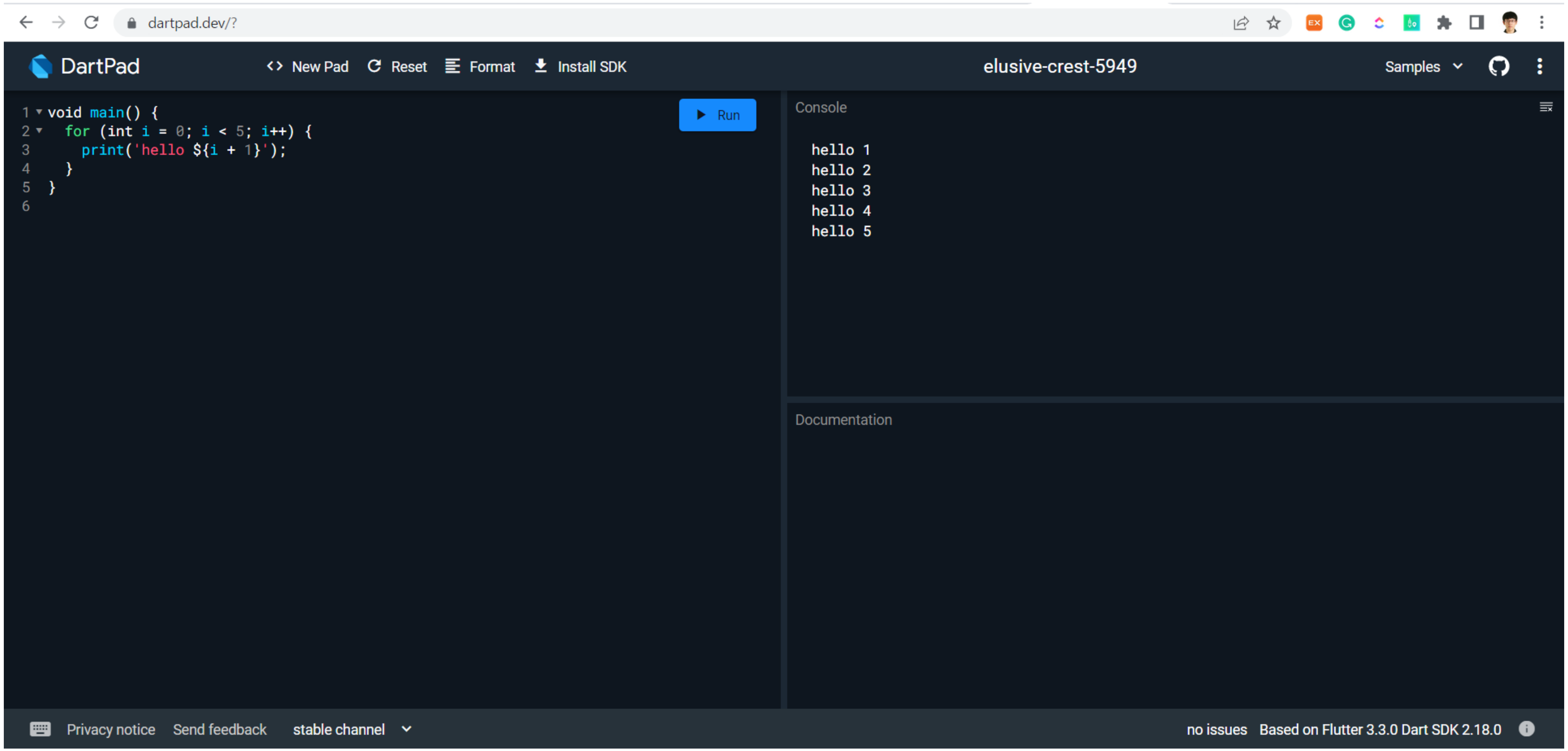
Advantage of Dart language

- Easy to learn for OOP programmers
- Great performance
- Fast development
- Support for various platforms and devices
- Active support for Flutter

DartPad

- DartPad is a free, open-source online editor for Dart language.
- It enables Dart and Flutter code to run in a Web browser environment.
- With DartPad, we do not need SDK or IDE installation.
- URL: dartpad.dev

DartPad



```dart
void main() {
  for (int i = 0; i < 5; i++) {
    print('hello ${i + 1}');
  }
}
```

```
hello 1
hello 2
hello 3
hello 4
hello 5
```

Hello World in Dart

- Like any C-based programming language, we should have one **main** function.
- The basic structure of functions is the same as C language.

```
void main() {
  print('Hello, Dart!’);
}
```

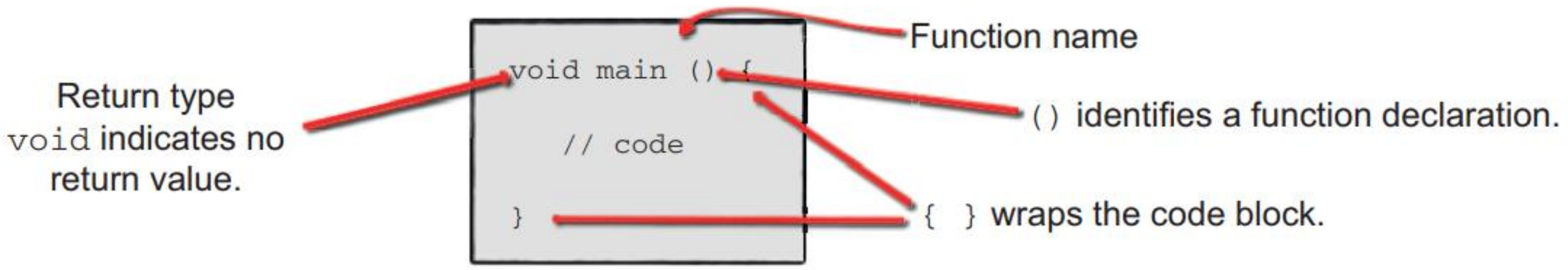

Function name
void main () {
// code
}
Return type void indicates no return value.
() identifies a function declaration.
{ } wraps the code block.

Figure 2.1 The `main` function in Dart

Function declaration and definition

- To use a user-defined function, the function does not need to be declared before the caller function.
- We simply need to implement the function after position where we want to use.

```
void main() {
  helloDart();
}

void helloDart() {
  print('Hello, Dart!’);
}
```

Function declaration and definition

- This function now expects a name argument.
- Trying to call this function with anything other than exactly one argument, of the type String, is an error.
- We can use **${…}** to print the content of a variable in print function.
  - The braces {} can be omitted when it is clear.
  - Dart throws warning when we use needless braces.

```
void main() {
  helloDart('Dart');
}

void helloDart(String name) {
  print('Hello, $name');
}
```

List data structure

- The List data structure is the basic **array-like** data structure.
- A List can manage its own size.
- You can create a list with the list-literal constructor using square brackets:

var myList = [a,b,c];

List<String> myList = [a,b,c];

```
void main() {
  List<String> greetings = [
    'World’, 'Mars', 'Oregon’,
    'Barry', 'David Bowie’,
  ];
  print(greetings[3]);

  for (var name in greetings) {
    helloDart(name);
  }
}
```

Map data structure

- The **Map** data structure is also a basic data structure in Dart.
- Instead of indexing each element as an ordered integer, Map indexes each element as a user-defined key.
- You can create a map with the list-literal constructor using braces:

var myMap = {p:a,q:b,r:c};

Map<String, int> myMap = {p:a,q:b,r:c};

```
void main() {
  Map<String, int> scores = {
    'World’: 30,
    'Mars’: 70,
    'Oregon’: 40,
  };

  print(scores[‘World’]);

  for (var score in scores.values) {
    print(‘Score is $score’);
  }
}
```

Common programming concepts in Dart

- Dart is an **object-oriented** language and supports **single inheritance**.
- In Dart, **everything is an object**, and every object is an instance of a class. Every object inherits from the *Object* class. Even numbers are objects, not primitives.
- Dart is **typed**. You cannot return a number from a function that declares it returns a string.
- Dart supports **top-level functions** and **variables**, often referred to as library members.
- Dart is **lexically scoped**.

Dynamic types

- When you set a variable as **dynamic**, you're telling the compiler to accept any type for that variable:

dynamic myNumber = 'Hello';

- Then, we can re-assign any types of variables or literals.

myNumber = 1;

- If we use **var** keyword, we can initialize the variable without writing the specific types, but the type of variable cannot be changed.

var myNumber = 'Hello';

my Number = 1; // Error!

When we use dynamic type

- The **dynamic** type is used in List or Map data structure in case we want to insert any types of elements in one list or map.

Map<String, dynamic> json = {‘a’: 1, ‘b’: ‘Hello’};

- We cannot write the following using **var** keyword.

Map<String, var> json; // Error!

null value

- All unassigned variables in Dart are **null**. null is a special value that means “nothing.”
- In Dart, null is an object, like everything else.
- If we do not determine any value in a variable declaration, the null value is assigned in this variable.

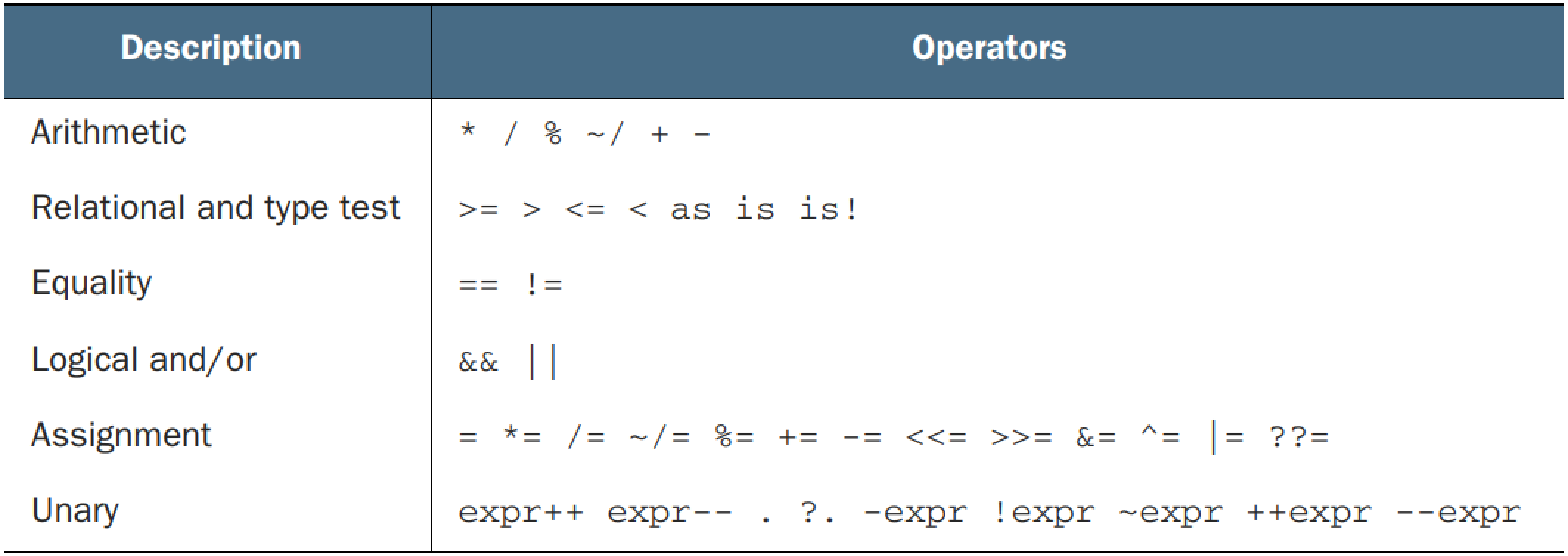
String name; // null will be assigned.

final, const keyword

- The first two, final and const, are similar. You should use these keywords if you want to make a variable immutable.
- **final** variables can only be assigned once. However, they can be declared before they're set at the class level.
  - a final variable is almost always a variable of a class that will be assigned in the constructor.
- **const** variables, on the other hand, won't be declared before they're assigned.
  - Constants are variables that are always the same, no matter what, starting at compile time.

Dart operators

Table 2.1 Dart operators



Description	Operators
Arithmetic	`* / % ~/ + -`
Relational and type test	`>= > <= < as is is!`
Equality	`== !=`
Logical and/or	`&& ||`
Assignment	`= *= /= ~/= %= += -= <<= >>= &= ^= |= ??=`
Unary	`expr++ expr-- . ?. -expr !expr ~expr ++expr --expr`

Dart operators

- **~/** is the symbol for integer division. This never returns a decimal point number, but rather rounds the result of your division to the nearest integer.

5 ~/ 2 == 2

- **as** is a keyword that typecasts. A variable in parent class can be typecast to child class.

a = Animal();

b = a as Dog;

- **is** and **is!** check that two objects are the same type. They are equivalent to == and !=.

if (a is int) { print('a is int type'); }

Null-aware operators

- In any language, having variables and values fly around that are null can be problematic.
- Programmers often have to write **if (response == null) return** at the top of a function.
- Null-aware operators, **?.**, **??**, and **??=**, can make the codes more concise.

?. operator

```
void getUserAge(String username) async {
  final request = new UserRequest(username);
  final response = await request.get();
  User user = new User.fromResponse(response);
  if (user != null) {
    this.userAge = user.age;
  }
}
```

```
void getUserAge(String username) async {
  final request = new UserRequest(username);
  final response = await request.get();
  User user = new User.fromResponse(response);
  this.userAge = user?.age;
}
```

- Assign userAge to user.age.
- If the user object is null, just assign userAge to null, rather than throwing an error.

?? operator

```
void getUserAge(String username) async {
  final request = new UserRequest(username);
  final response = await request.get();
  User user = new User.fromResponse(response);
  this.userAge = user.age ?? 18;
}
```

- Assign userAge to user.age.
- If the user object is null, then use this backup value.

??= operator

- If this object is null, then assign it to this value.
- If it's not, just return the object as is.

```
int x = 5;
x ??= 3;
```

- In the second line, x will not be assigned 3, because it already has a value.

Control flow: if, else if, else

- A condition must evaluate to a **Boolean**.
- There is only one way to say “true” (**true**) and one way to say “false” (**false**).
- In such languages, you can write **if (3) {,** and it works. That is not the case in Dart.

```
if (inPortland && isSummer) {
  print('The weather is amazing!’);
} else if(inPortland && isAnyOtherSeason) {
  print('Torrential downpour.’);
} else {
  print ('Check the weather!’);
}
```

Control flow: switch and case

```
int number = 1;
switch(number) {
  case 0:
    print('zero!’);
    break;
  case 1:
    print('one!’);
    break;
  case 2:
    print('two!’);
    break;
  default:
    print('choose a different number!’);
}
```

```
int number = 1;
switch(number) {
  case -1:
  case -2:
  case -3:
    print('negative!’);
    break;
  case 1:
  case 2:
  case 3:
    print('positive!’);
    break;
  case 0:
  default:
    print('zero!’);
    break;
}
```

Exiting switching statement

- Each case in a switch statement should end with a keyword that exits the switch.
- Most commonly, you'll use **break** or **return**.
    - **break** simply exits out of the switch
    - **return** immediately ends the function's execution
- You can use the **throw** keyword, which throws an error.
- You can use a **continue** statement and a **label** if you want to fall through but still have logic in every case.

```
Stringanimal = 'tiger';
switch(animal) {
 case 'tiger':
  print('it's a tiger');
  continue alsoCat;
 case 'lion':
  print('it's a lion');
  continue alsoCat;
 alsoCat:
 case 'cat':
  print('it's a cat');
  break;
 // ...
}
```

Ternary operator

- The ternary expression is used to conditionally assign a value.
- It’s called ternary because it has three portions—the condition, the value if the condition is true, and the value if the condition is false.

```
var nametag = user.title == 'Boss' ? user.name.toUpperCase() : user.name;
```

- This code says, “If this user’s title is ‘Boss,’ change her name to uppercase letters. Otherwise, keep it as it is.”

Loops: for, for-in, while, do while

```
for (var i = 0; i < 5; i++) {
  print(i);
}
```

```
List pets = ['Odyn', 'Buck', 'Yeti’];
for (var pet in pets) {
  print(pet);
}
```

```
while(someConditionIsTrue) {
  // do some things
}
```

```
do {
  // do somethings at least once
} while(someConditionIsTrue);
```

```
for (var i = 0; i < 55; i++) {
  if (i == 5) {
    continue;
  }
  if (i == 10) {
    break;
  } print(i);
}
```

Functions

- Functions look familiar in Dart if you’re coming from any C-like language.

```
String makeGreeting(String name) {
  return 'Hello, $name’;
}
```

- Dart also supports a nice shorthand syntax for any function that has only one expression, which we call arrow function.

```
String makeGreeting(String name) => 'Hello, $name';
```

Function parameters

- Dart functions allow **positional parameters**, **named parameters**, and **optional positional and named parameters**, or a **combination** of all of them.
- **Positional parameters** are simply what we've seen so far.

```
void debugger(String message, int lineNum) {
  // ...
}
```

- To call that function, you must pass in a String and an int, in that order.

```
debugger('A bug!', 55);
```

Named parameters

- Dart supports **named parameters**. **Named** means that when you call a function, you attach the argument to a label.
- This example calls a function with two named parameters:

```
debugger(message: 'A bug!', lineNum: 44);
```

- Named parameters are written a bit differently. You wrap any named parameters in curly braces ({ }).

```
void debugger({String message, int lineNum}) {
// ...
}
```

Positional optional parameters

- You can pass positional parameters that are optional, using [ ]Named parameters are written a bit differently. You wrap any named parameters in curly braces ({ }).

```
int addSomeNums(int x, int y, [int z]) {
  int sum = x + y;
  if (z != null) {
    sum += z;
  }
  return sum;
}
```

```
addSomeNums(int x, int y, [int z = 5]) => x + y + z;
```

```
addSomeNums(5, 4);
addSomeNums(5, 4, 3);
```

Lexical scope

- Dart is lexically scoped. Every code block has access to variables “above” it.
- You can see what variables are in the current scope by following the curly braces outward to the top level.

```
String topLevel = 'Hello’;
void firstFunction() {
  String secondLevel = 'Hi’;
  print(topLevel);
  nestedFunction() {
    String thirdLevel = 'Howdy’;
    print(topLevel);
    print(secondLevel);
    innerNestedFunction() {
      print(topLevel);
      print(secondLevel);
      print(thirdLevel);
    }
  }
  print(thirdLevel); // Error!
}
void main() => firstFunction();
```

Class

- **Class** is used for representing an object in object-oriented programming.
- Defining class is almost the same as other OOP languages, such as C++.
- Member variable (property), member function (method)
- In Dart, we do not need to use the **new** keyword when creating new instances.
  - Using **new** is not recommended in Dart.

```
class Cat {
  String name;
  String color;
}
```

```
Cat nora = new Cat();
nora.name = 'Nora’;
nora.color = 'Orange';
```

```
Cat ruby = Cat();
ruby.name = 'Ruby’;
ruby.color = 'Grey';
```

Constructors

- You can give classes special instructions about what to do as soon as a new instance is created. These functions are called **constructors**.
- When you assign each property to the variable you passed to the constructor, the right code can be used.

```
class Animal {
  String name;
  String type;
  Animal(String name, String type) {
    this.name = name;
    this.type = type;
  }
}
```

```
class Animal {
  String name, type;
  Animal(this.name, this.type);
}
```

Inheritance

- If you want to inherit some class, you can use **extends** keyword.
- The concept of inheritance is the same as other OOP languages.
- The child class can have the properties and methods of the parent class.

```
// superclass
class Animal {
  String name;
  int legCount;
}

// subclass
class Cat extends Animal {
  String makeNoise() {
    print('purrrrrrr’);
  }
}
```

Inheritance

- If you want to inherit some class, you can use **extends** keyword.
- The concept of inheritance is the same as other OOP languages.
- The child class can have the properties and methods of the parent class.

```
// superclass
class Animal {
  String name;
  int legCount;
}

// subclass
class Cat extends Animal {
  String makeNoise() {
    print('purrrrrrr’);
  }
}
```

Inheritance

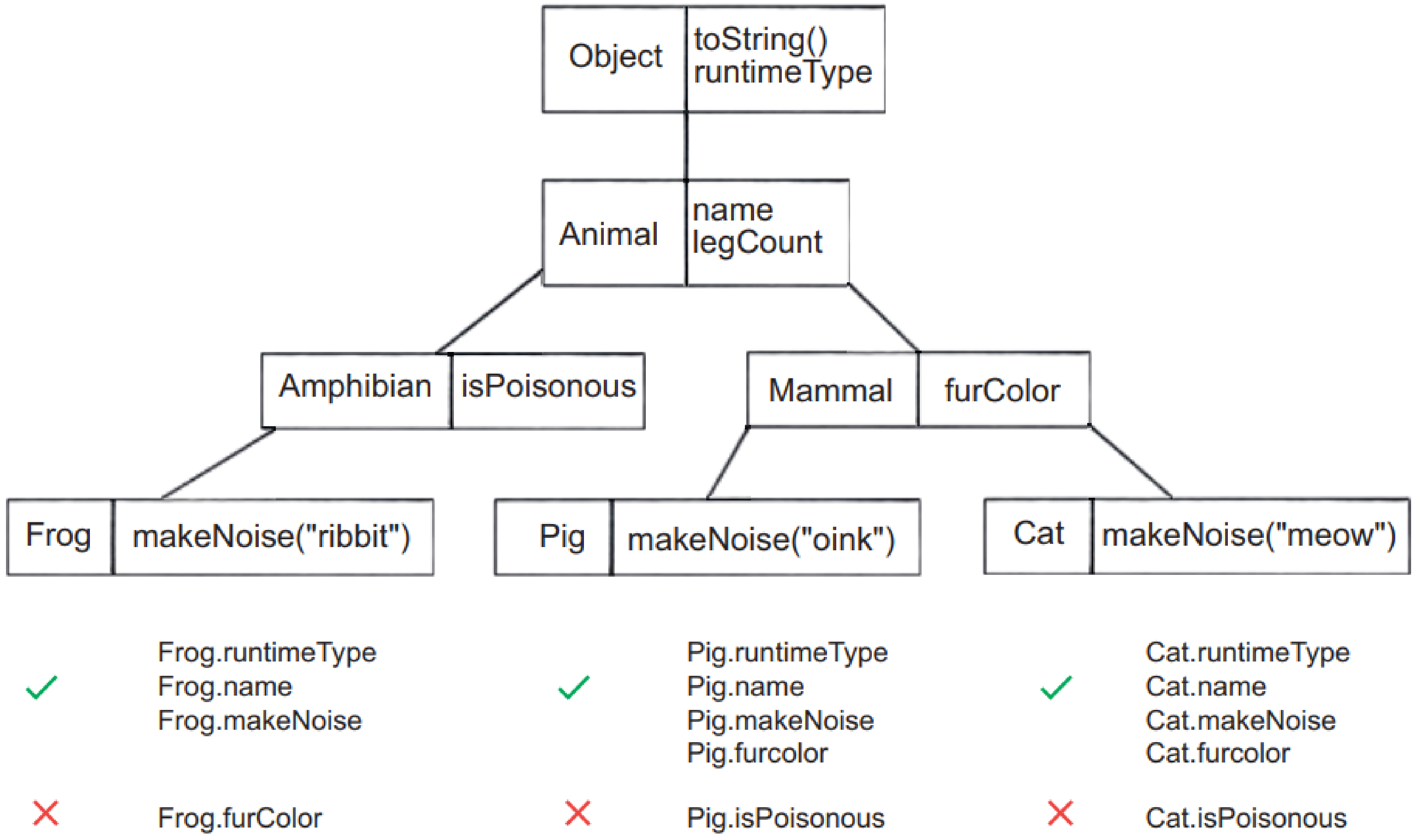


Figure 2.2 Object-oriented inheritance example