Dart for Everybody

Brylie Christopher Oxley

2023-11-13

Table of contents

1 Introduction

Embarking on the Coding Adventure: The Pragmatic and the Philosophical

1.0.1 Welcome to the World of Coding

Ahoy, future coder! Buckle up for a wild ride into the world of programming. It's a bit like Hogwarts, but instead of waving wands and uttering Latin spells, you'll wield a keyboard and summon magic with lines of code. Like a well-brewed potion, coding combines logic, creativity, and just a dash of audacity to manifest digital wonders. So, ready to swap your quill for a keyboard?

1.0.2 Why Dart?

"But why Dart?" you might wonder, with the digital world buzzing with so many coding languages. Well, Dart is the dashing, versatile hero of our coding saga. Simple to learn, flexible to use, and potent enough to create anything from mobile apps to web wizardry. If you've heard of Flutter, the super-framework for building eye-popping user interfaces, yep, that's Dart's baby.

1.0.3 The Programmer's Mindset

Coding isn't just about talking to machines; it's about cultivating a mental toolkit every bit as important as your coding skills. Think of it as learning to brew your potion - it involves precision, experimentation, and a fair share of spilled ingredients. Coding is a marathon, not a sprint. Well, more like a fun run where you get to design the track, the backdrop, and the finish line.

1.0.4 Embracing Errors & Iterative Improvement

Imagine casting a new spell. Do you get it right the first time? Probably not. You'll likely turn your teacher into a toad or something. Coding, much like spellcasting, involves a lot of trials, a ton of errors, and an infinite capacity for laughter. You'll make mistakes, and that's

not just okay—it's fantastic! Each error is a clue, leading you on a treasure hunt to a better solution.

1.0.5 Coding, Creativity, and Community

Coding isn't a lonely wizard locked in a tower; it's a magical feast at Hogwarts' Great Hall! It's where creativity meets collaboration. Programming communities are like a House Cup competition, but everybody wins! You'll code together, debug together, and create together. Every Hermione needs a Ron and Harry, after all.

1.0.6 What lies ahead?

This book is like your very own Marauder's Map. It will guide you from the fundamental spells (think: Dart syntax, data types) through the dark forests of control flow and functions to the enchanting world of classes. By the end, you'll be creating magic with code, just like Dumbledore, but with fewer beard issues.

So, are you ready to step onto platform 9 ¾ and board the Dart Express? There's a whole world of programming magic waiting to be discovered. Let the journey begin!

2 Building Blocks

3 Variables and Data Types

The Magician's Kit

3.1 Magic Potions and Spells: Dart Variables and Data Types

Welcome, future Dart wizards, to another enchanting chapter of our coding adventure. Today, we'll explore the mystical world of Dart variables and data types. You'll make variables dance to your tunes and morph data types like a true sorcerer by the end.

3.2 What's in a name? Everything, when it's a Variable!

First up, let's meet our shapeshifter friend - Variables. They're the magical containers in programming, storing, and changing their content as you command. They're like the Room of Requirement in Harry Potter – they can become anything you need them to be. Need to remember a user's name? Variables to the rescue. Want to calculate the score of your cheese-rolling game? Variables have your back!

3.3 The Wonderful World of Data Types

Now, imagine trying to stuff an elephant into a suitcase or pour an ocean into a teacup; impossible, right? That's where data types come into play. There are different kinds of boxes in Dart - sorry, I mean variables - to store different kinds of data. You have:

- **Numbers:** Dart gives you two types of numeric containers, `int` for whole numbers (like the number of cats you own) and `double` for fractional ones (like the exact amount of coffee you drink daily in liters).

```
int numberOfCats = 5; // You have five adorable cats
double litersOfCoffee = 0.75; // You drink 0.75 liters of coffee a day, cheers!
```

Strings: These are for text. Remember the name of your first pet or your favorite quote? Those are strings, and they are always written between 'single quotes' or "double quotes."

```dart
String petName = 'Fluffy'; // Your pet's name is Fluffy, aww!
String favoriteQuote = "Coffee first, schemes later."; // Wise choice
```

Booleans: These are the simplest of data types, storing only **`true`** or **`false`**. They're like the light switches of the coding world, deciding the path of your conditionals.

```dart
bool lovesCoding = true; // You love coding, right?
```

3.4 Declaring and Assigning Variables - Let's Get the Party Started!

Now that we have our Dart-flavored containers, let's fill them up. Declaring a variable is like sending a party invitation to a specific data type. You're telling it to reserve a spot in the memory and wait for the party to start.

```dart
String party; // Declared a variable named 'party' of type 'String'
```

Assigning a variable, on the other hand, is like kicking off the party by filling that memory spot with a value.

```dart
party = 'Started'; // The party has started!
```

You can also do both at once - send the invitation and start the party:

```dart
String party = 'Started'; // Declared and assigned in one line
```

3.5 Final, Const, Var, and Late - The Special Keywords

Think of these as the VIPs of the Dart world.

- **`final`**: This keyword is used when you have a variable you don't want to change after it's assigned. It's like a stubborn cat that found a comfortable spot on the couch and won't move, no matter what.

```dart
final String favoriteFood = 'Pizza'; // Pizza is love, Pizza is life
```

`const`: This is for values you know are constant and won't change even before runtime, like a day has 24 hours or a minute has 60 seconds.

```dart
const int hoursInDay = 24; // Time is constant... unless you're Doctor Who
```

var: Dart's version of "Whatever!". If you're feeling a bit lazy and want to avoid defining the data type, Dart's got your back. It will figure out the data type based on the value you assign.

```dart
var lazyVar = 'I am a string'; // Dart knows this is a string
```

late: This is Dart's way of saying, "Better late than never". Use this when you want to declare a variable but can't immediately assign it a value.

```dart
late String lateToTheParty; // We'll assign this variable later
```

So, that's your introductory crash course on Dart variables and data types. Remember, practice is key to mastering these concepts. So, start brewing your magic potions and casting your spells, and remember to have fun while you're at it!

4 Operators

4.1 Conjuring Magic: Operators in Dart

Gather 'round, code sorcerers! It's time to expand your Dart-spelling repertoire by adding a new chapter of incantations - the Operators. Just as you'd expect, these aren't your run-of-the-mill wand-waving tricks. These special symbols wield the power to perform actions and operations on our code.

Think of operators as the action heroes of programming. They swing, leap, and even backflip over your variables and data, performing everything from magical math transformations to logical labyrinths. And, Dart, being the fantastic magical realm it is, gives you a plethora of operators, each serving a unique purpose. Based on their stunts, we can categorize Dart operators into different troops.

Before we dive into this magical whirlpool, remember: with great power comes great responsibility. So, prepare to command these operators with wisdom and valor, or at least with a cup of coffee and a slice of pizza! Stay tuned for the upcoming action-packed performance from our brave Dart operators. Trust me; it'll be spellbinding!

Arithmetic

Assignment

Bitwise

Comparison

Logical

Type

5 Arithmetic

5.0.1 Mathemagic with Arithmetic Operators

Ladies and Gentlemen, it's time to pull out your magic wands - I mean, your keyboards - for our first act, the incredible, the unbelievable, the math-defying Arithmetic Operators!

Remember those characters you've met back in your Math class? Yes, the ones that liked adding apples and subtracting oranges! They're here, and they've got their game face on. In Dart, they've got some cool tricks up their sleeves:

- **+** The **Addition Operator**: This is like the super-friendly operator who's always eager to bring things together. Have a couple of numbers? The addition operator will happily unite them for you.

```
int cats = 5;
int dogs = 3;
int totalPets = cats + dogs; // You have 8 lovely pets now.
```

- The **Subtraction Operator**: This operator is the detective of the group, always finding the difference. Feed it two numbers, and it'll tell you how far apart they are.

```
int cakes = 10;
int eatenCakes = 3;
int remainingCakes = cakes - eatenCakes; // Still got 7 delicious cakes.
```

***** The **Multiplication Operator**: Imagine you have a magic spell to duplicate things. That's what the multiplication operator does, making your numbers multiply!

```
int hours = 24;
int days = 7;
int hoursInWeek = hours * days; // Abracadabra, there are 168 hours in a week.
```

/ The **Division Operator**: The peacekeeper of the group, the division operator evenly distributes one value among another, just like cutting a pie into equal pieces.

```
int pie = 10;
int friends = 2;
double piePerFriend = pie / friends; // Each friend gets 5.0 pieces of pie.
```

% The **Modulus Operator**: This sneaky operator is all about the leftovers. Divide one number by another, and the modulus will tell you what's remaining.

```
int totalCandies = 11;
int numberofKids = 3;
int leftoverCandies = totalCandies % numberofKids; // 2 candies left after sharing among 3
```

Just like that, our mathemagicians are making wonders happen with your code. Now, it's your turn. Grab those operators and start crafting your own spells. Remember, every great wizard starts with the basics!

6 Assignment

6.0.1 Charm Casting with Assignment Operators

Lend me your ears, magical coders, for the next spectacular act of our thrilling Dart journey - the mesmerizing, the enchanting, Assignment Operators!

Assignment operators are like the ultimate multitaskers of the Dart world. They juggle operations and assignments all at once, giving your fingers a well-deserved break from all that typing.

- **=** The **Basic Assignment Operator**: This operator is the team's workhorse. It's just here to assign values to variables, no questions asked.

```
int cupcakes = 12; // You've got 12 cupcakes!
```

+= The **Addition Assignment Operator**: This operator adds a dash of magic to the plain old assignment. It adds a value to a variable and then assigns the result back to the variable. Imagine a potion that grows sweeter with each spoon of sugar you add.

```
int sugar = 5; // 5 spoons of sugar
sugar += 3; // Now it's 8, sweetness overload!
```

-= The **Subtraction Assignment Operator**: This operator is here to balance the sweetness. It subtracts a value from a variable and reassigns the result back to the variable.

```
int coffee = 10; // Start with 10 cups of coffee
coffee -= 2; // You drank 2, so you have 8 left
```

***=** The **Multiplication Assignment Operator**: Like a charm that multiplies whatever it touches, this operator multiplies a variable by a value and assigns the result back to the variable.

```
int cats = 2; // 2 cats
cats *= 3; // Magic spell! Now you've got 6 cats!
```

/= The **Division Assignment Operator**: This operator believes in equal sharing. It divides a variable by a value and reassigns the result back to the variable.

```
double pizza = 12.0; // 12 slices of pizza
pizza /= 4; // You share with 3 friends, so everyone gets 3 slices
```

• **%=** The **Modulus Assignment Operator**: This operator is all about the leftovers. It applies modulus to a variable and assigns the result back to the variable.

```
int candies = 15; // Start with 15 candies
candies %= 4; // You share with 3 friends, and you have 3 candies left
```

With these charmed Assignment Operators at your disposal, the magic of Dart is now yours to command. Practice these incantations and make your own magic. Code on, wizards!

7 Bitwise

7.0.1 The Binary Circus: Bitwise Operators

Ladies and gentlemen, robots and AI, prepare yourselves for a dazzling dive into the digital depths of Dart. A realm where 0s and 1s juggle, tumble, and twist into patterns of pure logic. Welcome to the grand show of Bitwise Operators!

Imagine a grand binary ballet, where each 0 and 1 pirouettes gracefully at the command of these mystical operators. There are six primary dancers in this ballet:

- `&` The **Bitwise AND Operator**: It's like the picky eater of the group. This operator looks at the matching bits of two numbers and only takes a bite when both are 1. Otherwise, it starves (returns 0).

```
int firstNumber = 170;  // Binary: 10101010
int secondNumber = 240; // Binary: 11110000
int result = firstNumber & secondNumber; // Result: 160 (Binary: 10100000)
```

`|` The **Bitwise OR Operator**: The exact opposite of our AND operator, this one's the foodie. If either of the matching bits is 1, it chomps down, satisfied.

```
result = firstNumber | secondNumber; // Result: 250 (Binary: 11111010)
```

`^` The **Bitwise XOR Operator**: XOR is the rebel - it thrives on difference. It takes a peek at the matching bits, and only when they're different it munches (returns 1).

```
result = firstNumber ^ secondNumber; // Result: 90 (Binary: 01011010)
```

`~` The **Bitwise NOT Operator**: This is the dramatic one. It takes a binary number and flips all its bits in a grand gesture.

```
result = ~firstNumber; // Result: -171 (Binary: 01010101)
```

`<<` The **Left Shift Operator**: Imagine a marching band where every time the drum rolls, everyone takes a step to the left. That's what the left shift operator does, but with bits.

```
result = firstNumber << 2; // Result: 680 (Binary: 1010100000)
```

>> The **Right Shift Operator**: As you'd guess, this operator is all about marching to the right. A drumroll, and all bits step right.

```
result = firstNumber >> 2; // Result: 42 (Binary: 00101010)
```

With these bitwise operators, you can choreograph your own binary ballets, creating more intricate patterns and magical performances within your Dart code. So, take the stage, and let the binary ballet begin!

8 Comparison

8.0.1 The Court of Comparisons: Comparison Operators

Ladies and gentlemen, let's pay a visit to the esteemed Court of Comparisons, where the diligent Dart Comparison Operators preside. These operators work tirelessly, adjudicating the nuances of numbers, always seeking justice (or "just is," as they like to call it).

They are not just mere judges but also guides who illuminate the path of your code with the light of truth (or false, because sometimes you need to venture down the dark path to find the cheese in the labyrinth). Each judgment they make (true or false) is equally important, guiding your program along its destined path.

Now, let's meet our sagacious sentinels:

- **==** The **Equality Judge**: Always checks if two values are identical twins. If yes, it proclaims, "True! A perfect match."

```
bool areWeEqual = (10 == 10); // This indeed is true. 10 and 10, you may now high-five!
```

!= The **Inequality Judge**: This one sniffs out differences. If two values are not alike, it declares, "True! As different as chalk and cheese."

```
bool areWeDifferent = (10 != 5); // This indeed is true. 10 and 5, you're as different as
```

< The **Less Than Usher**: It humbly checks if one value is less than another.

```
bool amILess = (5 < 10); // True it is! 5, you may take your seat in the "Less than 10" se
```

> The **Greater Than Usher**: The robust sibling, always ensuring that one value deserves the "greater than" badge.

```
bool amIGreater = (10 > 5); // True, 10, you're indeed greater than 5. Your badge is waiti
```

<= The **Less Than or Equal To Magistrate**: A kind soul, checks if a value is less than or just equal to another.

```
bool lessOrEqual = (5 <= 5); // True, indeed! 5, you're as equal to 5 as peas in a pod.
```

>= The **Greater Than or Equal To Magistrate**: A balanced judge, checking if a value is more than or just equal to another.

```
bool greaterOrEqual = (10 >= 5); // True! 10, you stand tall, greater or equal to 5.
```

8.1 Why the parentheses?

Now, you might be wondering why we swaddle our comparisons in parentheses like newborn babies. It's not just because they look cute that way (though they absolutely do)! Parentheses in programming help us establish a clear order of operations—ensuring our diligent judges make their proclamations at the right time, every time. So wrap those comparisons up snug and tight, for they have a big role to play in your Dart code!

And thus, we conclude our visit to the Court of Comparisons. Remember, in the court of Dart Comparison Operators, truth and falsity are equally important. They're the yin and yang that keep the cosmos of your code balanced. Happy comparing!

9 Logical

9.0.1 Welcome to the Boolean Ballroom: Logical Operators

Are you ready to join the dance of decisions, the waltz of wisdom, the samba of sanity checks? Then, let's step into the Boolean Ballroom, where Dart's Logical Operators, the maestros of the Boolean Orchestra, make the magic happen. These operators weave together Boolean values (our darling dancers known as **`true`** and **`false`**) into a fantastic dance of logic and reason.

Here are our talented choreographers:

- **`&&`** The **Logical AND Duet**: This performance requires both dancers (Boolean values) to be true for the entire dance to be a success. If even one dancer forgets a step (becomes false), the dance is not the same.

```
bool perfectDuet = (true && true); // True! Both dancers nailed it!
bool stumbledDuet = (true && false); // False! One of the dancers missed a beat. Keep prac
```

`||` The **Logical OR Ensemble**: This is a more forgiving performance. The dance will still shine as long as at least one of our dancers performs flawlessly (is true).

```
bool amazingEnsemble = (false || true); // True! One dancer's incredible spin saved the da
```

`!` The **Logical NOT Solo**: The prima donna of logical operations, it only requires one dancer. This solo performance is a curious one, as the dancer always performs the opposite of what they rehearsed.

```
bool unexpectedSolo = !true; // False! Wait, we thought they were going to perform a true!
```

Remember, our Boolean Ballroom depends on perfect timing. So, ensure that you have your conditions and Boolean values in the right order. A misstep in logic can lead to an entirely different dance!

With that, our show in the Boolean Ballroom concludes. Don your top hat, dust off your tailcoat, and ready your Boolean dancers for the next performance in the Dart Disco!

10 Type

10.0.1 All Aboard the Type Test Express: `is` and `is!`

Buckle up, coding comrades! We're about to embark on a thrilling journey aboard the Type Test Express, the one-stop solution to avoid the pitfalls of type confusion. Dart's type test operators are like the friendly conductors checking tickets, making sure everyone (or in our case, every variable) is in the right place.

There are two conductors in our train analogy, **`is`** and **`is!`**, and they handle our passengers - variables - with care and precision.

- **`is`** The **Identity Inspector**: This operator, like an attentive conductor, checks if a variable is of a certain type. If it is, it responds with a courteous nod (returns **`true`**). But if it's not, it shakes its head (returns **`false`**).

```
var aTrainTicket = 'Choo Choo!';
print(aTrainTicket is String); // True! This ticket is indeed a string.
```

`is!` The **Non-Identity Inspector**: This operator has an edgier approach. Instead of checking if a variable is of a certain type, it checks if it is not of a certain type. If the variable's type does not match, it gives a triumphant thumbs-up (**`true`**). But if it does match, it frowns in disappointment (**`false`**).

```
print(aTrainTicket is! int); // True! Our ticket is no integer, it's a string!
```

These conductors, **`is`** and **`is!`**, are here to keep our code on track, making sure that all variables are precisely what we expect them to be. Just remember, when in doubt, give a shout out to **`is`** and **`is!`**.

So, as we pull into our next station, the Land of Control Flow, remember to keep your variables in check and your types in order!

11 Documentation

The importance of comments and how to write good documentation for your code is an often overlooked art.

This chapter will be written soon…

12 Steering the Code

12.1 Flow Control and Functions

Welcome, thrilled carnival-goers and spellbound wizards, to the enchanting spectacle of "Steering the Code"! Much like orchestrating a grand carnival or casting a powerful spell, coding too requires choreography and craft. In this section, we'll delve into two magical aspects of coding, Flow Control and Functions, the carnival rides and enchantments that lead you through the mesmerizing maze of your code.

Flow Control is your guide through the bustling carnival of programming; it's the lively carnival barker who decides which thrilling rides to experience based on various circumstances. Whether it's choosing between the swirling Ferris Wheel or the exciting Roller Coaster (our conditional statements), or going for a second spin on the carousel (our loops), flow control lets your code adapt and react to all the lively happenings of the carnival.

Then we have **Functions**, the magical spells of our programming world. These are enchanting incantations that perform specific tricks again and again, without the need to recite the entire spell every time. Just like a magic spell can turn a frog into a prince with a single word, functions can streamline your code, making it efficient, readable, and easier to debug.

So, step right up folks! Join us as we dive into the whirl of carnival rides and the realm of magical enchantments in steering our code with Flow Control and Functions. Grab your carnival tickets and magic wands, fellow programmers, for an unforgettable coding adventure awaits you!

Flow Control

Functions

13 Flow Control

14 The Flow Control Fiesta

Ladies and gentlemen, hold on to your keyboards because we're about to embark on a wild, thrilling ride through the Dart programming language's carnival of control flow! Here, we have an amazing array of attractions that help us steer the logic of our programs.

14.1 Conditional Statements: The Mystical "if-else" Labyrinth

Imagine stepping into a labyrinth where every turn you take is based on a true-or-false condition. That's our dazzling **`if-else`** maze!

Here's our brave contestant **`weatherIsNice`**, set to either **`true`** or **`false`**.

```
bool weatherIsNice = true; // It's a sunny day!
```

Based on this adventurous variable, our contestant will decide which turn to take:

```
if (weatherIsNice) {
    print("Let's go for a picnic!"); // If the weather is nice, we're going on a picnic!
} else {
    print("Let's stay home and code."); // If the weather is not nice, more time for codin
}
```

Just remember, our **`if-else`** labyrinth never leads you astray!

14.2 Loop-a-Loop Roller Coaster: The Fabulous "for" Loop

Next, brace yourself for the exhilarating Loop-a-Loop roller coaster, where repetition reigns supreme!

```
for (int rideCount = 1; rideCount <= 5; rideCount++) {
    print("This is ride number $rideCount!");
}
```

The **`for`** loop is like a roller coaster ride with three important parts:

- **The Seat Belt Check (`int rideCount = 1`):** This is where we prepare for the ride. We set our `rideCount` to 1, indicating we're about to embark on the first ride.
- **The Safety Bar (`rideCount <= 5`):** This is our safety constraint. We continue riding as long as `rideCount` is less than or equal to 5. Once we exceed 5 rides, safety regulations dictate that it's time for a break!
- **The Exciting Loop (`rideCount++`):** This is the thrilling part where we go around the loop. After each ride, we increment `rideCount` by 1, marking off another exhilarating loop on our ride.

Strap in as you watch `rideCount` go from 1 to 5 in the blink of an eye, taking you through the loop each time!

14.3 Waiting in line: the for-in loop

Now, let's step deeper into the magical world of Dart looping and meet `for-in`, a spellbinding variation of `for` that's more commonly known as `each` in other programming lands. The `for-in` loop graciously invites every item in a list to a fabulous dance, one at a time.

```
var rollerCoasterRiders = ['Anna', 'Bob', 'Charlie'];

for (var rider in rollerCoasterRiders) {
    print('$rider is ready to ride the Loop-a-Loop roller coaster!');
}
```

Our `for-in` loop is like a carnival barker, calling each eager roller coaster rider from the `rollerCoasterRiders` list to take their turn on the Loop-a-Loop roller coaster. Isn't it just enchanting? Dart certainly knows how to host a spectacular code carnival!

14.4 The Never-ending Carousel: The Whirling "while" Loop

Don't forget to take a whirl on our Never-ending Carousel! As long as a condition holds true, you're going for another spin!

```
int lollipopCount = 10;
while (lollipopCount > 0) {
    print("Eating a lollipop...");
    lollipopCount--;
}
print("All lollipops are gone. Time for another ride!");
```

Our hero here keeps devouring lollipops until there are none left, making for a delicious and dizzying journey.

14.5 Do...While Loops: the at-least-once loop

Just like how some carnival games might keep you hooked until you win a prize, a do-while loop performs a task at least once, and then keeps doing it as long as a certain condition holds true. Let's put this in a Dart context:

```
int gamesPlayed = 0;
do {
  print("Playing game number ${gamesPlayed + 1}... Let's win that stuffed unicorn!");
  gamesPlayed++;
} while (gamesPlayed < 5);
```

In this example, **gamesPlayed** keeps track of the number of games we've played. We're committed to playing at least one game (gotta win that stuffed unicorn!), so the code inside the **do** block executes first. After each game, the **gamesPlayed** counter increases. If the number of games is less than 5, we go for another round, if not, we're finally out of the do-while funhouse!

14.6 Break: The Safety Hatch

The **break** statement is like a shortcut to escape the dizzying rides. If you're riding the carousel and start feeling dizzy, you would want to hop off right away, even if the ride isn't over. Let's check this out with some Dart code:

```
for (int carouselRides = 1; carouselRides <= 10; carouselRides++) {
  if (carouselRides > 5) {
    print("Oh no! We've had too much spinning on the carousel. Time to stop.");
    break;
  }
  print("Enjoying carousel ride number: $carouselRides");
}
```

In this example, **carouselRides** starts at 1 and, for each loop, we add another ride. But when we've ridden more than 5 times, the world starts spinning a bit too much! This is where the **break** statement comes in, allowing us to jump off the carousel right away.

14.7 Continue: Next Ride Please

The **continue** statement is like skipping a ride you don't like. If you come across a ride that you're not particularly fond of (like that too-scary haunted house), you can skip it and continue to the next one. Here's how we can do that in Dart:

```
List<String> carnivalRides = ['Ferris Wheel', 'Carousel', 'Haunted House', 'Roller Coaster

for (var i = 0; i < carnivalRides.length; i++) {
  var currentRide = carnivalRides[i];
  if (currentRide == 'Haunted House') {
    print("Oh, no! The Haunted House is too scary. Let's skip it.");
    continue;  // If Haunted House is encountered, skip to the next iteration
  }
  print("Enjoying the $currentRide!");
}
```

In this example, we're looping through a list of **carnivalRides**. When we encounter the 'Haunted House', we decide it's a little too spooky and skip it, moving straight to the next ride.

14.8 The Magical "Switch" House of Mirrors

Last but not least, step into the mysterious House of Mirrors where the magical **switch** statement takes the center stage. Each mirror reflects a different case, and our reflection hops from one mirror to another based on its value!

```
String dartLevel = 'beginner';

switch (dartLevel) {
    case 'beginner':
        print('Welcome, young apprentice! Your journey has just begun.');
        break;
    case 'intermediate':
        print('You have come a long way, young warrior!');
        break;
    case 'expert':
        print('Ah, Master Coder, you have mastered the Dart side of the Force!');
        break;
    default:
```

```
        print('Hmm, an unidentified entity you are. Keep practicing Dart, you must.');
}
```

14.9 Wrap-up

Well folks, that was a fantastic journey through Dart's Flow Control Fiesta! Each of these attractions, from the **`if-else`** Labyrinth to the **`switch`** House of Mirrors, helps guide the story of our Dart code. So, practice hard, and remember - in the magical world of Dart, you're the ride operator!

Stay tuned for our next fun-filled chapter, and until then, code on amigos and amigas!

15 Functions

15.1 The Fabulous World of Functions

Once upon a time, in the enchanted land of Dart, there existed a magical guild known as "Functions." This extraordinary league of spells was capable of performing marvelous feats, often transforming inputs into outputs in the blink of an eye!

15.1.1 Meet the Functions

Let's imagine functions as friendly magical creatures with various abilities. You call them by name, give them a task (your inputs), and *poof!* they work their magic, bringing you back a result (your output). For example, there could be a creature named **sum**, and if you ask **sum** to add 2 and 3, it'll return 5 with a twinkle in its eye. The code for such a function would look like this:

```
int sum(int a, int b) {
  return a + b;
}
```

Here, **sum** is the name of the function, **int a** and **int b** are the parameters (the tasks), and **a + b** is the magical result. The **int** in front of **sum** tells us that our magical creature always brings back an integer as a result. Remember, types are important in Dart!

15.1.2 A Function of Many Hats

Sometimes, a function may take on different roles depending on the task it is given. In Dart, these are known as optional and named parameters. Picture a magical creature who can juggle, dance, and sing — but it only performs the tasks you specifically ask for. Check this out:

```
String introduce(String name, {String talent = 'nothing'}) {
  return '$name can do $talent';
}
```

If you only tell this function your name, it will assume you can do nothing (how rude!). However, if you specify your talent, it'll sing your praises!

15.1.3 Anonymous Functions: The Unknown Hero

In the Dart magical universe, there are anonymous functions too. They're like the masked superheroes of the coding world - no names, just action! You'll often find them sneaking around in the world of higher-order functions (more on that later) or fluttering about in the Flutter framework, bringing life to buttons and various interactive elements. Here's how our anonymous hero might look:

```
var anonymousHero = (task) => 'Completing $task at lightning speed!';
```

15.1.4 Recursive Functions: The Magic Mirror

In the hall of magic mirrors, a special kind of spell echoes endlessly. These are the recursive functions — functions that call themselves in their own definition. It's a bit like a magical creature casting a spell to clone itself, then the clone casting the same spell, and so on, until a certain condition is met. Here's an example:

```
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

It calculates the factorial of a number (n!) by continually multiplying `n` by the factorial of `n - 1`, until `n` is 1.

15.1.5 Higher-Order Functions: The Superstars

Lastly, we have the superstars of the function world: higher-order functions. These are functions that have learned to tame other functions. They can take other functions as gifts (parameters), or they can spawn new functions to do their bidding (return a function).

```
Function makeAdder(int addBy) {
  return (int i) => addBy + i;
}
```

his function creates a new function that can add any number to `addBy`.

So, there you have it! In the enchanted land of Dart, functions aren't just ordinary lines of code. They are magical creatures, full of wonder and whimsy, ready to transform inputs into outputs, work magic anonymously, echo in eternity, and even control other functions. As you continue your journey, these magical creatures will be your most loyal companions, always ready to perform feats of code at your behest!

Next up, we'll be delving into the realm of Dart's mythical beasts: the data structures. Stay tuned, brave adventurer!

16 Working With Data

17 Data Structures

17.1 Conjuring and Commanding Data Structures

17.1.1 Introduction to Data Structures

Think of data structures as the puzzle boxes of the programming world. These clever contraptions hold our precious data, much like a dragon hoards its treasure. But fear not! There's no fire-breathing beast here, only some deceptively simple yet extremely powerful tools that you'll soon wield like a seasoned wizard.

17.1.2 Basic Data Structures

There are a few magical containers in Dart's magic toolbelt: Arrays, Lists, Sets, and Maps.

Arrays are like a line of enthusiastic concertgoers — they keep your data in a specific order. Each element in an array has a ticket with a number (index) that indicates its place in the line.

Lists are very similar to Arrays; they are like an extensible line of fans at a pop idol meet-and-greet. Not only can the idol remember who was first, but they can also let the line grow as more fans show up.

Sets are a bit like a magic trick where duplicates disappear. If you put in 'rabbit', 'hat', 'rabbit', and poof! The duplicate 'rabbit' vanishes, leaving you with just 'rabbit' and 'hat'.

Maps, on the other hand, are like a magical dictionary. They hold key-value pairs. Imagine you have a key named 'unicorn' and it leads you straight to a fabulous image of a unicorn. Isn't that neat?

17.1.3 Working with Data Structures

Manipulating these magical containers is where the fun begins. Just like a witch brewing a potion, you can add or remove ingredients (elements), find the one that adds a certain sparkle, or sort them to have the rarest ingredient on top. All it takes is the right incantation (method)!

17.1.4 Custom Data Structures

Sometimes, more than the provided magical containers are needed. That's where you can concoct your own data structure using classes. Imagine creating a magic bag that, whenever you reach in, always gives you the object you need the most at that moment. With custom data structures, you can create just about anything your magical heart desires!

17.1.5 Choosing the Right Data Structure

Picking the right magical container is like choosing the right spell in a duel - the wrong choice could lead to disastrous consequences or just some unnecessary hassle. In our quest to become programming wizards, we must learn to consider the trade-offs of each data structure, such as memory use and performance.

17.1.6 Practical Applications of Data Structures

Understanding data structures is like having a magical map to navigate the enchanted forest of programming. Whether it's organizing the high scores in a game, creating a social network of wizards and witches, or even plotting the quickest path to the nearest pizza parlor, data structures are the key.

17.1.7 Complex Data Structures

Lastly, we have the elder dragons of the data structures world: trees and graphs. They might seem intimidating at first, but fear not! They're just a different kind of puzzle box. For those brave enough to venture into this advanced territory, there are many resources to guide you. Just remember, every master wizard started as an apprentice.

And with that, we conclude our magical journey through data structures. But remember, every ending is also a new beginning. With your newfound knowledge, you're ready to conjure and command data structures in ways you never thought possible! Now, go forth and code!

18 Collections

****A Roller Coaster Ride through Lists, Maps, Sets, and Queues****

Buckle up, dear reader! It's time to navigate the looping tracks and dramatic dips of Dart's data structures—Lists, Maps, and Sets. Picture these as thrilling rides in our programming carnival, bringing both order and excitement to our code.

18.1 Lists: The Ferris Wheel of Dart

In Dart, a List is like a gigantic Ferris wheel, where each gondola carries a specific value or 'passenger.' Much like how Ferris wheels can carry all sorts of passengers—from the thrill-seekers, the lovebirds, to those just looking to enjoy the view—Dart lists can hold a variety of data types: numbers, strings, booleans, and yes, even other lists!

```
var shoppingList = ['eggs', 'milk', 'chocolate'];
var listOfMyMistakes = ['forgot semicolon', 'used = instead of ==', 'infinite loop'];
var listofMistakesIllProbablyMakeAgain = listOfMyMistakes;
var paradoxicalList = ['this sentence is false', 'list of lists not containing themselves'
```

The Dart list is a programmer's dream-come-true Ferris wheel. No matter how high you want to go, there's a gondola ready to take you there! Each gondola is accessed by an index, starting from `0` (yes, we programmers start counting from zero—another charming quirk to add to the 'list of programming eccentricities').

18.2 Maps: The Treasure Hunt of Dart

Now, let's move on to another exhilarating ride: Maps. A Map is like an exciting treasure hunt. Each treasure (value) in a Dart map is assigned a unique clue (key). To find the treasure, you simply follow the clue!

```
var pirateTreasure = {
  'golden coins': 1000,
  'precious gems': ['ruby', 'emerald', 'sapphire'],
```

```
  'mysterious map': 'leads to another adventure'
};
```

In our treasure hunt, **`'golden coins'`** is a clue that leads to a treasure of **`1000`**. Pretty cool, eh? Maps are superb when we want to link values together, just like connecting clues with treasures. Who knew coding could feel so much like a pirate adventure?

18.3 List and Map Methods: The Bag of Tricks

While we've introduced Lists and Maps and their superpowers, it's time to take a deep dive into their toolboxes. You see, Lists and Maps in Dart aren't just storage units; they're like that handy multi-tool you take on a camping trip.

Let's consider a list **`shoppingList`** of items you want to buy.

```
List<String> shoppingList = ['toothpaste', 'shampoo', 'sponge', 'bananas'];
```

If you suddenly remember that you need bread, instead of creating a new list, you can use the **`add()`** method:

```
shoppingList.add('bread');
```

And voila! Your shopping list now includes bread. But let's say you've picked up a bread-making hobby, and you no longer need bread in your list. Simply use **`remove()`**:

```
shoppingList.remove('bread');
```

Similarly, Maps aren't just simple old treasure maps. They are more like GPS systems that come with a range of features. Given a map **`favoriteFoods`**:

```
Map<String, String> favoriteFoods = {
  'Alice': 'Pizza',
  'Bob': 'Pasta',
};
```

If we find out that Charlie loves Tacos, we don't need a new map. Just use **`putIfAbsent()`**:

```
favoriteFoods.putIfAbsent('Charlie', () => 'Tacos');
```

And there you have it! Charlie and his love for Tacos have officially made it to the **`favoriteFoods`** map!

18.4 Sets: The Unique Snowflakes

Sets are a lot like the attendees of an exclusive party: no one likes duplicates! A Set is a collection of unique items. Think of it as a box where you put snowflakes. You wouldn't want two identical snowflakes now, would you?

In the following code, **Sets** are like that exclusive VIP lounge. There's no particular order, and no duplicate guests are allowed. Alice won't get in twice, even if she changes her disguise!

```
void main() {
  Set<String> vipLounge = {'Alice', 'Bob', 'Charlie', 'Alice'};

  print(vipLounge.length); // 3 party-goers in the VIP!
  print(vipLounge.contains('Alice')); // Alice made it into the VIP? Yes, she did!
  vipLounge.add('Dave'); // Dave's on the list, too!
  print(vipLounge); // Check out who's in the VIP lounge now!
}
```

18.4.1 Set operations

Sets have some special powers that make them a bit different than other collections.

Imagine we're at a food court area now, and we have two friends, Alice and Bob. Each of them has a set of foods they want to try at the carnival.

Let's explore the union operation, which gives us a set of all unique items in both sets. In the context of our carnival, this will give us all unique food items that either Alice or Bob wants to try.

```
void main() {
  // Foods Alice wants to try
  Set<String> aliceFoods = {'Popcorn', 'Cotton Candy', 'Churros'};

  // Foods Bob wants to try
  Set<String> bobFoods = {'Hotdog', 'Popcorn', 'Nachos'};

  // Foods that either Alice or Bob wants to try
  Set<String> unionFoods = aliceFoods.union(bobFoods);

  print('Either Alice or Bob wants to try these foods: $unionFoods');
  // Outputs: Either Alice or Bob wants to try these foods: {Popcorn, Cotton Candy, Churro
}
```

As you can see, the **union** method returns a new Set that contains all the elements of both the original Sets. In this case, it's all the foods that either Alice or Bob wants to try. Notice how 'Popcorn' only appears once in the union set, despite both Alice and Bob wanting to try it. That's because Sets only contain unique items.

I hope this example gives you a better understanding of the **union** operation on Dart Sets. Enjoy your carnival treats!

18.5 Queues

Now, imagine a **Queue** as the line for the hot dog stand. In this line, it's a strict "first come, first served" policy. The first one to line up will be the first one to get their hot dog!

```
import 'dart:collection';

void main() {
  Queue<int> hotDogLine = Queue();

  hotDogLine.addLast(1); // Customer No.1 joins the queue!
  hotDogLine.addLast(2); // Customer No.2 is next!
  hotDogLine.addLast(3); // Customer No.3 takes the last spot!

  print(hotDogLine.removeFirst()); // Customer No.1 gets his hot dog!
  print(hotDogLine); // Let's see who's still waiting for their hot dog!
}
```

18.6 Iterables and Loops: The Theme Park Ride

Remember the roller coaster ride of loops we discussed earlier? Now, combine that with Lists and Maps, and you have an entire theme park of data to explore. You can loop over Lists and Maps, accessing and manipulating data as you whizz by!

All these tricks and tools make Lists, Maps, and Sets not just containers for storing data but multifunctional devices that can be manipulated and traversed in many ways. After all, programming isn't just about storing values; it's about playing with them!

18.7 Multi-Dimensional Lists, Sets, and Maps: The Rabbit Holes

Just like in Alice in Wonderland, things can get much more complex in the world of Lists and Maps. Lists can store other lists, and maps can store other maps. Picture a giant matryoshka doll situation, except it's with data structures.

18.8 Conclusion

Just like that, at our Dart Programming Carnival, we ensure a fun and fair time for all! The use of collections like `List`, `Map`, `Set`, and `Queue` can provide a structured way to juggle data, akin to juggling colorful juggling balls at this grand carnival!

In the land of Dart, Lists, Maps, and Sets are the kings and queens. They provide a multitude of options to store, access, and manipulate data. They're the powerful allies you want by your side as you embark on your programming journey. So keep practicing, and in no time, you'll become a data structure wizard! Remember, it's all just a bit of hocus-pocus with a dash of data.

19 Supercharging Your Dart

20 Asynchronous Programming

20.1 Sip, Savor, and Async - Asynchronous Programming in Dart

Welcome to our next exciting chapter. Remember the last time you and your friends decided to hit your favorite café? You walked into the smell of freshly ground coffee beans, the sound of the espresso machine, and an overwhelming choice of tempting treats.

Programming, my dear friend, is no different than this café. Just like you've got multiple things going on, from ordering your food to catching up with your friends, a computer program often needs to do several things at once. So, let's get caffeinated and dive right into the world of synchronous and asynchronous programming!

20.1.1 Synchronous vs Asynchronous Programming

Imagine this scenario. You and your friends step into the café. You approach the counter to order. "One cappuccino, please," you say. The barista starts preparing your coffee right away. But until your coffee is ready, nobody else can order. Your friends have to wait until your coffee is done. Sounds pretty annoying, right? Well, this, my friends, is an example of synchronous programming.

In contrast, asynchronous programming would go a little like this: You place your order, the barista nods, and then you step aside. Your friends can place their orders too. The barista works on the orders as they came in. Everybody gets a buzzer that will notify them when their order is ready. In the meantime, you're all free to find a seat, chat, or scroll through your social media. Much better, isn't it?

20.1.2 Futures

In Dart, the concept of a "future" is central to understanding async programming. In our café analogy, a future is like your buzzer. When you place an order, you're handed a buzzer. This buzzer is a promise that your order will be ready at some point in the future.

In Dart, when we have an operation that might take a while to complete (like querying a database, downloading a file, or in this case, preparing a delicious cappuccino), we represent

this operation as a **Future**. When the **Future** completes, it'll either hold the result of the operation or an error, in case something went wrong.

```
Future<String> makeCoffee(String order) async {
    var coffee = await barista.prepareCoffee(order);
    return 'Your $coffee is ready!';
}
```

20.1.3 Async and Await

The **async** and **await** keywords are the superheroes of Dart's asynchronous programming. You can think of them as the digital version of your buzzer and you waiting for your coffee.

```
void main() async {
    print('Placing order for cappuccino...');
    var coffee = await makeCoffee('cappuccino');
    print(coffee);
}
```

The **async** keyword tells Dart that some asynchronous operations are happening within that function, and **await** tells Dart to wait for a **Future** to complete and extract its value.

20.1.4 Error handling in Async programming

Sometimes, things don't go according to plan. Maybe the café ran out of your favorite cinnamon rolls, or perhaps the barista accidentally spilled your coffee. When things go awry in asynchronous programming, you need to handle these errors gracefully.

```
Future<String> makeCoffee(String order) async {
    try {
        var coffee = await barista.prepareCoffee(order);
        return 'Your $coffee is ready!';
    } catch (error) {
        throw 'Oh no, there was a problem making your $order!';
    }
}
```

20.1.5 Streams

Lastly, let's talk about streams. Imagine you're sitting at your table, and you see your barista making drinks one after another, each one a different order. In Dart, a stream is a way to receive a sequence of events, one after another. You can think of it as having a front-row seat to watching the barista in action, with each completed order being an event in the stream.

```
class Order {
    String personName;
    String drinkName;
    Duration preparationTime;

    Order(this.name, this.drinkName, this.preparationTime);
}

class Barista {
    Future<Order> makeOrder(Order order) async {
        print('Preparing {$order.drinkName} for ${order.personName}...');
        await Future.delayed(order.preparationTime);
                print('Order is prepared.');
        return order;
    }
}

Stream<Order> baristaMakingOrders(Barista barista, List<Order> orders) async* {
    for (var order in orders) {
        yield await barista.makeOrder(order);
    }
}

void main() {
    var barista = Barista();
    var orders = [
        Order('Alice', 'Latte', Duration(minutes: 2)),
        Order('Bob', 'Mocaccino', Duration(minutes: 3)),
        Order('Charlie', 'Espresso', Duration(minutes: 1)),
    ];

    var subscription = baristaMakingOrders(barista, orders).listen(
        (Order order) {
            print('${order.beverageName} is ready for ${order.personName}');
        },
```

```
        onError: (err) {
            print('Oh no, there was a problem: $err');
        },
        onDone: () {
            print('The barista has finished all the orders!');
        }
    );
}
```

The code above models a café scenario where a barista is preparing drinks for a series of orders. It leverages Dart's asynchronous programming capabilities to simulate the process of making a drink, which takes a certain amount of time. This process is modeled using a **`Future`**, and the sequence of drink orders is represented as a **`Stream`**. Here is a more detailed breakdown of the different parts of this code:

The `Order` class: This class represents an order placed by a person in the café. Each **`Order`** object has a **`personName`** (the name of the person who placed the order), a **`drinkName`** (the type of drink ordered), and a **`preparationTime`** (the amount of time it takes to prepare the drink).

The `Barista` class: This class represents the barista who is preparing the drinks. The **`Barista`** class has a single method, **`makeOrder`**, which takes an **`Order`** as input. This method returns a **`Future<Order>`**, simulating the asynchronous process of making a drink. The **`makeOrder`** method first prints a message saying it's preparing the drink, then "waits" for a duration equal to the order's preparation time using the **`Future.delayed`** function. After the delay, it prints a message saying the order is prepared and returns the **`Order`**.

The `baristaMakingOrders` function: This is an asynchronous generator function that takes a **`Barista`** and a **`List<Order>`** as input. For each **`Order`** in the list, it waits for the **`Barista`** to finish making the order, then yields the **`Order`**. Because it uses the **`yield`** keyword, this function returns a **`Stream<Order>`**, which emits each completed order as soon as it's ready.

The `main` function: This function sets up the scenario and runs the simulation. It first creates a **`Barista`** and a list of **`Order`**s. Then, it calls the **`baristaMakingOrders`** function and subscribes to the resulting **`Stream<Order>`**. The **`listen`** method is called on the **`Stream`** to handle the **`Order`** events it emits. This function takes three arguments:

- A callback function that gets called whenever a new **`Order`** is ready. This function simply prints a message saying the drink is ready for the person who ordered it.
- An **`onError`** callback that gets called if there's an error while preparing an order. This function prints an error message.
- An **`onDone`** callback that gets called when all orders have been completed. This function prints a message saying the barista has finished all the orders.

This code provides a practical and relatable demonstration of asynchronous programming in Dart, showing how **`Future`**s and **`Stream`**s can be used to manage time-consuming tasks and sequences of events.

21 Regular Expressions

21.1

The Treasure Hunt of The Secret Code Society

"Welcome, brave ones, to the Secret Code Society, a mysterious world where each symbol and character holds its own secret power. Regular expressions are our magic spells, weaving characters together to find the hidden treasures amidst the sea of text. So, put on your detective caps, and let's unravel the enigma of Dart's regular expressions."

21.2 The Basics: The Map to the Treasure

```
void main() {
  var re = RegExp(r'\b[A-Z]\w*\b');
  print(re.allMatches('In The Secret Code Society, Every Clue Counts').length); // 8
}
```

This, my friends, is the map to your first treasure. Here, `\b[A-Z]\w*\b` is the cryptic instruction set by the society. It translates to "find every word starting with a capital letter". `\b` is the boundary between a word and a non-word, `[A-Z]` is any uppercase letter, and `\w*` is any word character (letter, number, or underscore) appearing zero or more times.

The magic spell, `re.allMatches`, reveals the number of treasures hidden in the sentence. Here, it unearths seven!

21.3 The Number Riddle: Decoding Numerical Clues

```
void main() {
  var re = RegExp(r'\d+');
  print(re.hasMatch('Find The 7 Hidden Treasures')); // true
}
```

Here's a number riddle for you! Sometimes, the clues are hidden in the form of numbers. The spell `\d+` means "find one or more digits". It's like the society's secret number detector, sniffing out numerical secrets. `hasMatch` then checks if our riddle, 'Find The 7 Hidden Treasures', indeed contains such a secret. It joyfully returns `true` - yes, a hidden number was found!

21.4 Word Puzzles: The Jigsaw of Characters

```
void main() {
  var re = RegExp(r'cl\w*');
  final match = re.firstMatch('In the Secret Code Society, every clue counts')!;
  print(match[0]);
}
```

At times, you'll encounter a jigsaw of characters. Here, `cl\w*` decodes to "find any word starting with 'cl' ". It's like having a jigsaw piece and looking for the rest of the puzzle. `firstMatch` then tells us the first word it found to fit this puzzle in our sentence. And lo and behold, it found 'clue'!

21.5 The Laughing Sphinx: Lighter Moments in The Society

Why don't secret agents use regular expressions?

Because they don't like anything that *captures*!

Ah, a little humor to brighten our cryptic journey! Remember, while regular expressions can appear baffling, they're nothing but a fun, coded treasure hunt. Each symbol, each character is part of the magic, forming patterns that unlock the mysteries of text.

So, keep your wits about you, Secret Code Society members, and enjoy this journey of unraveling Dart's regular expressions. Onwards to the next treasure!

22 Numeric Computing

23 Chapter: Numeric Computing in Dart - The Celestial Harmony

> "As an astronomer, I don't just explore the universe through a telescope. I explore it through numbers. I am the composer of the cosmos, writing symphonies in algorithms and equations."

Welcome, future space explorers! In this chapter, we will journey across the cosmic plains of numeric computing in Dart. With the `dart:math` library as our spaceship, we'll uncover the mysteries of the universe and the harmonic symphony it plays.

23.1 The Basics: Cosmic Constants and Astronomical Arithmetic

```
import 'dart:math' as math;

void main() {
  print('Pi: ${math.pi}');
  print('Euler\\'s number: ${math.e}');
  print('Golden ratio: ${(1 + math.sqrt(5)) / 2}');
}
```

Our universe dances to the tunes of some significant constants. Pi, the ratio of a circle's circumference to its diameter, and Euler's number, the base of natural logarithms, are the melodies that govern the elliptical orbits of planets around stars. The golden ratio? It's everywhere, from spiral galaxies to the arrangement of sunflower seeds!

23.2 Spherical Trigonometry: A Harmonious Trio of Sin, Cos, and Tan

```
double radianToDegree(double radian) => radian * (180 / math.pi);

void main() {
  double angle = 60; // Let's say the angle is 60 degrees

  double angleInRadians = math.radians(angle); // Convert it to radians

  print('Sin(60): ${math.sin(angleInRadians)}'); // 0.86602540378
  print('Cos(60): ${math.cos(angleInRadians)}'); // 0.5
  print('Tan(60): ${math.tan(angleInRadians)}'); // 1.73205080757
}
```

The universe's orchestra plays in the language of trigonometry. When mapping the night sky or calculating the path of a comet, `sin`, `cos`, and `tan` help us describe the universe's spherical geometry. They're the basis of our cosmic sheet music.

23.3 3. Celestial Harmonics: Power and Logarithms

```
void main() {
  print('2^10: ${math.pow(2, 10)}'); // 1024.0
  print('Log(100): ${math.log(100)}'); // 4.605170185988092
}
```

Ever wonder why star brightness is measured on a logarithmic scale? It's because the energy they produce, seen as light, is enormously varied. By using logarithms, we can make sense of this vast spectrum. As for `pow`, well, it's how we calculate the incredible energies involved when stars go supernova!

23.4 4. Random Constellations: Dart's Random Class

```
void main() {
  var rng = math.Random();
  for (var i = 0; i < 10; i++) {
    print(rng.nextInt(100)); // Generate a random integer between 0 and 99
  }
```

```
}
```

Just like how we discover unexpected patterns in the night sky, Dart's `Random` class adds an element of surprise to our code. Remember, while the universe might seem chaotic, there's always an underlying pattern or principle — a hidden harmony.

23.5 For all humanity

Whether you're calculating the gravitational interaction of galaxies or plotting the lifecycle of a star, Dart's numeric computing capabilities and the `dart:math` library are like your scientific calculators. Just remember, every time you crunch a number, you're playing a note in the grand symphony of the universe. So, don't just code. Compose! Create your cosmic concerto with Dart and uncover the hidden harmonies in the cosmos.

24 Organizing Your Code

Organizing Your Space(ship): A Journey in Code Cleanliness

Chapter: Organizing Your Space(ship): A Journey in Code Cleanliness

Imagine this: You're the proud pilot of a state-of-the-art spaceship, bristling with impressive technology, darting through the interstellar vastness. Now, remember that this spaceship is not just a means of transport. It's your home, your lab, and your workstation, all at the same time. It's a lot like your code. Each part of your ship needs to be carefully organized, otherwise, you'd be floating in a mess of loose nuts, bolts, and ill-placed astronaut ice-cream!

1. The Toolshed (Variables and Functions)

Starting off, your spaceship has a toolshed where you store all your nifty tools, just like the variables in your code. You wouldn't leave your hyper-wrench floating around the ship, would you? Likewise, it's best to keep your variables confined to the part of the code where they're needed. If a variable is used only inside a function, don't let it float freely in the spaceship; keep it safe and organized inside the function!

2. The Control Room (Control Structures)

The control room is the heart of your spaceship, where you pilot your vessel. Like if-else statements and loops, each button and lever in the control room serves a specific purpose. Just as you'd group related controls together (say, navigation on one panel, life-support on another), it's important to keep related pieces of logic together in your code. Grouping related logic makes your code easier to navigate and understand.

3. Living Quarters (Objects and Classes)

The living quarters are where the crew lives and works, much like objects in your code. Each room is designed for a specific person with all the amenities they need - just like a class in your code, designed for a specific purpose with all the variables and methods it needs. This organization helps avoid clutter and confusion.

4. The Engine Room (Modules and Libraries)

The engine room is filled with heavy-duty machinery, powering your spaceship, much like the libraries and modules in your code. Each machine performs a specific function. Instead of

letting these machines sprawl all over the spaceship, you keep them in the engine room, right? Similarly, keep your code that performs specific tasks within its own module or library.

5. The Logbook (Comments)

Finally, the ship's logbook is an essential tool to remember what each switch does or what each part of the ship is for. It's a lot like comments in your code. Proper commenting is like leaving a clean logbook for the next shift or even for future-you! Remember, the most confusing code is code that you wrote six months ago.

Organizing your spaceship, much like organizing your code, is a vital part of being an interstellar explorer or a stellar programmer. Having a well-organized ship/codebase will make your journey more enjoyable and less of a stellar headache!

25 Object-Oriented Programming

****Dart of Justice - The League of OOP Superheroes****

In our journey through the programming universe, we've met many characters - from loops to conditionals, from data structures to functions. Now, it's time to meet the League of Superheroes - the Object-Oriented Programming (OOP) team.

25.0.1 What is OOP?

Picture this: an Avengers movie, but instead of heroes like Thor, Black Widow, or Black Panther, you have... wait for it... Classes and Objects. They are our real superheroes, the true Avengers in the programming universe!

OOP is a paradigm where everything revolves around Classes and Objects, just like a comic book centers around its superheroes and their mighty exploits. It allows us to encapsulate data (properties of superheroes, perhaps?) and methods (their superpowers, obviously) into reusable and interactive packages known as classes. Sounds exciting, right? Let's dive in.

25.0.2 Classes and Objects

Imagine a superhero template, a blueprint, if you will, where you just fill in the characteristics and powers and voila, you have your hero. That's exactly what a **class** is - a blueprint from which objects are created.

```
class Superhero {
  String name;
  String power;

  Superhero(this.name, this.power);
}
```

The real superheroes are **objects**, instances of the class. For example, Spiderman is an instance of the `Superhero` class.

```
var spiderman = Superhero('Spiderman', 'Web-slinging');
```

25.0.3 Constructors, and the `this` keyword

When a superhero is born (or, ahem, created), they come with their own unique characteristics. This happens in our class via a **constructor** - a special method that brings our object to life.

In Dart, constructors share the same name as the class and use the **`this`** keyword to refer to the current instance of the class. **`this`** is like our superhero pointing to their own chest saying, "Hey, this is me!"

```
class Superhero {
  String name;
  String power;

  Superhero(this.name, this.power);
}
```

25.0.4 Getters and Setters

Think of getters and setters as the superhero's PR team. They control how the public gets information about the hero (getter) and how they can change it (setter). If a setter isn't set, no one can change the information. For instance, no one but Peter Parker can change the fact that he is Spiderman!

```
class Superhero {
  String name;
  String power;

  Superhero(this.name, this.power);

  String get identity => name;
  set identity(String identity) {
    name = identity;
  }
}
```

25.0.5 Inheritance and the `super` keyword

Just like how superheroes sometimes pass on their mantle to successors (think Batman -> Robin), classes can inherit properties and methods from other classes. The class being inherited from is the superclass, and the inheriting class is the subclass. The **`super`** keyword is used to refer to the superclass.

```
class Speedster extends Superhero {
  int speed;

  Speedster(String name, this.speed) : super(name, 'Super Speed');
}
```

25.0.6 Polymorphism

In a superhero team-up, each hero uses their powers in their own unique way, right? This is exactly what polymorphism is. The same method behaves differently in different classes. We call this method overriding.

```
class Speedster extends Superhero {
  //...
  void showPower() {
    print('$name runs at $speed mph!');
  }
}
```

25.0.7 Composition

Composition is like our superhero stepping into a high-tech chamber, with various traits and abilities available at the touch of a button. Each trait is represented by a different class, and our superhero can pick and choose the ones they want to embody. From super hearing to chameleon camouflage, each super-trait becomes part of the hero's whole toolkit. Let's see it in action:

```
class SuperSpeed {
  void runFast() {
    print('Running at lightning speed!');
  }
}

class SuperStrength {
  void liftHeavy() {
    print('Lifting a mountain, no big deal!');
  }
}
```

```
class SuperSneeze {
  void superSneeze() {
    print('A-choo! There goes the neighborhood...');
  }
}

class CustomHero {
  SuperSpeed speed;
  SuperStrength strength;
  SuperSneeze sneeze;

  CustomHero(this.speed, this.strength, this.sneeze);

  void showOff() {
    speed.runFast();
    strength.liftHeavy();
    sneeze.superSneeze();
  }
}

var myHero = CustomHero(SuperSpeed(), SuperStrength(), SuperSneeze());
myHero.showOff();
```

In the world of OOP, composition lets us make our code more dynamic, just like our custom superhero who can now run faster than a cheetah, lift heavier than a Hercules, and… sneeze harder than a gale-force wind? Well, we never said all superpowers had to be serious!

25.1 Design patterns

The Superpowers of our League

Design patterns are just like the special abilities of our superheroes - tried and true techniques that come in handy when faced with recurring coding challenges. They are templates or blueprints that can be reused in many different situations. Let's take a look at a few.

25.1.1 Singleton: The Immortal One

This pattern restricts a class from instantiating multiple objects. There's only one instance of this class across your application, much like there's only one Immortal One!

```
class Singleton {
  static final Singleton _singleton = Singleton._internal();

  factory Singleton() {
    return _singleton;
  }

  Singleton._internal();
}

void main() {
  var s1 = Singleton();
  var s2 = Singleton();

  print(identical(s1, s2));  // prints: true
}
```

Here, the `Singleton` class has a private constructor, which means no other class can instantiate it. When we call `Singleton()`, we get the same instance every time, thus ensuring the immortality of the singleton object.

25.1.2 Factory: The Shapeshifter

Factory pattern is about creating objects without specifying the exact class of object that will be created, just like the Shapeshifter can morph into anything!

```
abstract class Shape {
  void draw();
}

class Circle implements Shape {
  void draw() {
    print('Drawing a circle');
  }
}
```

```
class Square implements Shape {
  void draw() {
    print('Drawing a square');
  }
}

Shape shapeFactory(String type) {
  if (type == 'circle') return Circle();
  if (type == 'square') return Square();
  throw 'Can\\'t create $type';
}

void main() {
  var circle = shapeFactory('circle');
  circle.draw(); // prints: Drawing a circle

  var square = shapeFactory('square');
  square.draw(); // prints: Drawing a square
}
```

Here, `shapeFactory` returns a `Shape` object but the actual type is hidden from the client, allowing the Shapeshifter to transform into a `Circle` or `Square`.

25.1.3 Observer: The Mind Reader

This pattern allows an object (the observer) to watch another object (the subject), and be notified when the subject changes state. Just like the Mind Reader is always in the know!

```
class Subject {
  List<Observer> observers = [];

  void addObserver(Observer observer) {
    observers.add(observer);
  }

  void notify(String message) {
    for(var observer in observers) {
      observer.update(message);
    }
  }
}
```

```
class Observer {
  void update(String message) {
    print('Received: $message');
  }
}

void main() {
  var subject = Subject();
  var observer1 = Observer();
  var observer2 = Observer();

  subject.addObserver(observer1);
  subject.addObserver(observer2);

  subject.notify('Something happened!');
}
```

Here, `Observer` classes get notified when the `Subject` calls `notify()`, keeping them up to date with the latest happenings. That's Mind Reader for you!

With these design patterns in your superhero toolkit, you're ready to face even the most villainous coding challenges head-on. Remember, with great power comes great responsibility, so use them wisely!

25.2 Mixins

While we covered composition briefly in our chapter on object-oriented programming, there is a more conventional way of mixing and matching new combinations of code called Mixins.

Mixins are a way of reusing a class's code in multiple class hierarchies. Think of it as a magical tome of spells that any wizard can use, regardless of their school of magic. Here's an example:

```
mixin Agility {
  void dodge() {
    print("Dodged the attack!");
  }
}

class Player {
  void attack() {
```

```
    print("Attack!");
  }
}

class Rogue extends Player with Agility { }

void main() {
  var rogue = Rogue();
  rogue.attack();
  rogue.dodge();
}
```

In this code, the **Rogue** class has all the methods of the **Player** class and also the **Agility** mixin.

25.3 Generics

Generics are a way of writing flexible, reusable code that works with different data types. They're like a magical key that can open any lock in our game! Here's a simple example:

```
class TreasureBox<T> {
  T content;

  TreasureBox(this.content);

  void open() {
    print("You've found $content!");
  }
}

void main() {
  var goldBox = TreasureBox<int>(500);
  goldBox.open(); // You've found 500!

  var gemBox = TreasureBox<String>('a rare gem');
  gemBox.open(); // You've found a rare gem!
}
```

In this example, **TreasureBox** is a generic class that can contain any type of content. **T** is a placeholder that you replace with the actual type when you create an instance of **TreasureBox**.

These are some of the advanced topics in Dart. Each of these topics can be a chapter in itself. Understanding these advanced features of Dart can make your code more robust, flexible, and efficient. As you continue your journey into Dart programming, be sure to explore each of these topics in more depth!

25.4 Let's wrap this up!

So, just like how every superhero adds value to the Avengers, every concept in OOP is crucial in programming. And remember, with great coding power comes great bug-fixing responsibility! Next up, we'll be exploring the cosmic realms of Asynchronous Programming. Stay tuned!

26 Advanced Dart

The following advanced topics will commonly be encountered in the first years of programming. Each concept is given a corresponding sub-chapter since they have many details. We keep the chapters short, however, so the reader will have a basic understanding of each concept.

27 Error Handling

27.1 Error Handling: Beyond the Basics

In our Dart programming journey, we might stumble upon more complicated issues, like difficult and unexpected traps in our treasure hunt. Let's equip ourselves with more advanced error-handling tools!

27.1.1 Custom Exceptions

In Dart, we can create our own custom exceptions to represent specific error conditions. Let's say, for instance, you're trying to open a treasure chest, but it's locked. In the programming world, this could be a "ChestLockedException". Let's see how we could create and use this custom exception:

```
class ChestLockedException implements Exception {
  String cause = "The chest is locked!";
}

void main() {
  try {
    throw ChestLockedException();
  } catch(error) {
    print(error.cause); // Will output: The chest is locked!
  }
}
```

Here's what's happening in this code:

1. We define a class **`ChestLockedException`** that implements the **`Exception`** interface. This class has a string field **`cause`** that contains the message "The chest is locked!".
2. In the **`main`** function, we use a **`try-catch`** block to throw and catch this exception. Inside the **`try`** block, we throw a new instance of **`ChestLockedException`**.
3. The **`catch`** block then catches the thrown exception. Since we know that our exception has a **`cause`** field, we can access it using **`error.cause`** and print it out.

So, in plain language: we are simulating a scenario where we're trying to open a locked chest. As we expect, an error occurs (the chest is locked), so we throw a specific **`ChestLockedException`**. We then catch this exception and print out the reason the error occurred (the chest is locked).

27.1.2 Stack Traces

When an error occurs, Dart gives you a stack trace, which is like a breadcrumb trail leading back to the origin of the error. This helps you track down the source of the error and fix it. Here's how you might see it:

```
void functionThatThrows() {
  throw Exception('This is an exception');
}

void main() {
  try {
    functionThatThrows();
  } catch(error, stackTrace) {
    print(error);
    print(stackTrace);
  }
}
```

In the code above, **`stackTrace`** is a StackTrace object that you can print out to see the sequence of method calls leading to the exception.

Here's a hypothetical example of what the **`stackTrace`** could look like:

```
Exception: This is an exception
#0      functionThatThrows (...dart_project/main.dart:2:3)
#1      main (...dart_project/main.dart:7:5)
#2      _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:309:32)
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:184:12)
```

The output of the **`stackTrace`** starts with the most recent method call (the source of the exception) and follows the path of execution back to the first method call.

Each line of the **`stackTrace`** typically includes:

- The index of the stack frame (starting with 0),
- The name of the method that was called,
- The path to the file where the method was defined,

- The line number and column where the method was called in that file.

In this case, the **`functionThatThrows`** (which is in your **`main.dart`** file on line 2, column 3) is the first method listed because it's where the exception was thrown. The following line is the **`main`** method (also in your **`main.dart`** file, but on line 7, column 5), which is where **`functionThatThrows`** was called. The last two lines are part of Dart's internal workings and show the steps Dart took to start your program.

This kind of traceback is incredibly useful in debugging because it tells you where the error occurred and how the program got there.

Next:

Cascade Notion Operator

28 Cascade Notion Operator

28.0.1 Cascade Notation Operator

It's time to meet one of Dart's handy wizards - the Cascade Notation Operator (`..`). This operator might seem like just two little dots, but don't let that fool you - it has a mighty power in Dart.

The Cascade Notation Operator allows you to perform a series of operations on a single object. It's like having a magical multitasking wand that can quickly perform several spells on the same item.

Let's imagine we're organizing a grand feast for all the brilliant programmers across the globe, and we're using a fabulous recipe app we've created using Dart. Let's add a few tasks to our recipe with the cascade operator.

```
class Recipe {
  String name = '';
  List<String> ingredients = [];
  List<String> steps = [];

  void showRecipe() {
    print('Recipe: $name');
    print('Ingredients: $ingredients');
    print('Steps: $steps');
  }
}

void main() {
  var feast = Recipe()
    ..name = 'Global Programmer\'s Feast'
    ..ingredients.addAll(['Code snippets', 'A dash of creativity', 'Cupfuls of inclusivity
    ..steps.addAll(['Mix well', 'Compile without errors', 'Serve hot with a side of fun'])
    ..showRecipe();
}
```

Isn't that neat? The cascade operator (`..`) allows us to call multiple methods or set multiple properties on our **`Recipe`** object, all in one go, without needing to reference the **`feast`** variable

multiple times. It's like our code is multitasking! With the cascade operator, we're not just preparing a feast - we're serving a masterclass in efficiency.

That's the power of Dart's Cascade Notation Operator. It's all about performing many tasks in a neat, streamlined manner. It's like having a team of magical sous chefs all working in perfect harmony to prepare the ultimate coding cuisine. Bon appétit!

Take time for quiet contemplation in:

Isolates

29 Isolates

In the realm of Dart, each bit of code resides within an 'Isolate', an entity that is, quite aptly, isolated from all others. These Isolates don't share memory. It's as if each one is a different planet in our Dart galaxy, each having its own atmosphere (memory) and life-forms (code).

```
void broadcastMessage(String message) {
  print(message);
}

void main() {
  Isolate.spawn(broadcastMessage, 'Greetings from Planet Dart!');
}
```

In the stellar code above, **`Isolate.spawn`** forms a new planet (creates a new isolate) and broadcasts the **`broadcastMessage`** signal from that planet (runs the function in that isolate). The second argument to **`Isolate.spawn`** is the interstellar message transmitted to the new planet (isolate).

Think of it this way: You're creating a new planet and then broadcasting an interstellar message: 'Greetings from Planet Dart!'.

And the cool part? Each planet (isolate) exists in its own corner of the galaxy, unaffected by any cosmic disasters (errors) that might befall its neighbors. Isn't it fascinating? Welcome to the cosmic journey of Dart Isolates!

Be more meta with:

Metaprogramming

30 Metaprogramming

Alright, let's board the Hogwarts Express and delve into the meta-magical world of Dart metaprogramming!

In the mystic world of Dart, the `@` character is our wand to cast annotations, a potent charm to attach magical properties (metadata) to parts of your incantations (code). The **`magicSpell`** you see in your code is an annotation - a magical spell you cast on your code.

These annotations can be wielded by wizarding tools and enchanted libraries to transform the way the incantations are processed or interpreted. For instance, some spells (annotations) might be used to conjure additional code, provide hints to the all-knowing prophecy orbs (static analysis tools), or alter the code's behavior in the runtime realm.

To cast the **`magicSpell`**, it must be summoned from the depths of your code. This might look like a simple incantation like this:

```
class magicSpell {
  const magicSpell();
}
```

In this example, **`magicSpell`** is a magical class with a single spell-casting command (constructor) that takes no magical ingredients (arguments). This is a common pattern for annotations. You can think of **`magicSpell`** as a magical badge that we're placing on another spell (function).

For example, when you use **`@magicSpell`** above **`doSomething`**, you apply that magical badge to the **`doSomething`** spell.

```
@magicSpell
void doSomething() {
  // Do something magical...
}
```

At this point, the **`magicSpell`** annotation is just a magical badge that does nothing itself. But other spells—like a metaprogramming tool or enchanted library—could read your incantations, discover the **`magicSpell`** annotation, and do something magical with the **`doSomething`** function because of it. That magic could be anything from conjuring additional code to modifying the function's behavior in the runtime realm.

It's like walking into a potions class, where the outcome can be as surprising as it is magical! Welcome to the wondrous journey of Dart metaprogramming!

Extend your capabilities with:

Extension Methods

31 Extension Methods

Alright, let's spin this with a dance party theme! Extension methods are like adding new dance moves and expression to your repertoire - you're expanding your abilities, and adding a spark to the dance floor.

```
extension on String {
  String addSomeEnthusiasm() => toUpperCase() + '!!!';
}

void main() {
  print('let us dance'.addSomeEnthusiasm()); // LET US DANCE!!!
}
```

In the above code, we're busting out a new move, **`addSomeEnthusiasm`**, on the **`String`** class. Now, every string can add enthusiasm to the dance floor by converting the string to upper case and adding three exclamation marks - making your statements loud, proud, and dance-ready!

Think of extension methods as learning a new dance style to spice up your routine. You're not changing the fundamental steps - walking is still walking, and a String is still a String - but you're adding some flair to your execution. So get ready to shake things up and make the code dance to your rhythm with extension methods!

Allow specific values with:

Enumerated Types

32 Enumerated Types

Enumerated types, also known as enums, are a special type of class in Dart used to represent a fixed number of simple values. Enums are incredibly useful when describing a value that can only take one out of a small set of possible values.

Imagine you're writing a program that deals with the days of the week. Sure, you could represent the days of the week as strings or integers. Still, then you'd constantly have to remember whether "Monday" is represented by "1" or "0", and there's a risk of accidentally assigning "8" to a variable that represents a day. Here's where enums come in.

```
enum DaysOfTheWeek {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday,
}
```

With this enum definition, you can now create variables that can only hold a valid day of the week:

```
DaysOfTheWeek today = DaysOfTheWeek.Monday;
```

Dart will prevent you from assigning any other value to **`today`**. This way, we avoid the risk of typo errors and make the code safer and easier to read. Enums improve code clarity and prevent errors by ensuring that variables can only take one of a fixed set of values.

Note that the enum's values (like **`DaysOfTheWeek.Monday`**) are full-fledged objects in their own right, with a **`index`** property that gets automatically assigned based on their position (starting from 0):

```
print(DaysOfTheWeek.Monday.index); // Output: 0
print(DaysOfTheWeek.Friday.index); // Output: 4
```

You can also retrieve a list of all the values in an enum using the **`values`** property:

```
print(DaysOfTheWeek.values);
// Output: [DaysOfTheWeek.Monday, DaysOfTheWeek.Tuesday, DaysOfTheWeek.Wednesday, DaysOfTh
```

This can be useful for situations where you need to loop over all possible values of an enum.

And that's an introduction to enumerated types in Dart! As you can see, they're a simple yet powerful feature that can make your code safer and easier to understand.

Avoid interstellar surprises with:

Sound Null Safety

33 Sound Null Safety

Let's hop on an intergalactic adventure.

In the infinite expanse of the coding universe, a lurking danger often disrupts the peace of coding civilizations: the infamous NullReferenceException, also known as the "Billion-Dollar Mistake." This shady character is known to make programs implode, causing catastrophic system failures.

But fear not! In the high-tech world of Dart, the Coding Council came up with a mighty force field known as Sound Null Safety. This innovative force field protected the world's code, ensuring that no variable could contain a null, a vacuum in this case unless the coder declared it explicitly.

```
int energyLevel = null; // Alert! A variable of type 'int' can't be assigned null.
int? fuelLevel = null; // This is acceptable; fuellevel can contain null (or vacuum) becau
```

In this scenario, the variable `energyLevel` is like a spaceship refusing to accept a vacuum in its energy core. But `fuelLevel` is designed to handle a vacuum because of the '?' symbol, making it a more versatile spacecraft.

Sound Null Safety is like the advanced AI system on a spaceship, it ensures that all variables act like vacuum-resistant spaceships unless told otherwise by the '?' symbol.

But what happens when you need to use a variable that could potentially be a vacuum? The Coding Council had foreseen this too. You are required to "check" the vacuum. Imagine it as a protective mechanism on the spaceship, allowing operations only if they're not going to create a vacuum:

```
int? potentialVacuum = null;
int guaranteedFuel = potentialVacuum ?? 0; // If potentialVacuum is null (or a vacuum), th
```

Thanks to Sound Null Safety, the coding universe of Dart is well-protected against the disruptive effects of the NullReferenceException. This has made coding exploration much safer and has ensured continued prosperity and progress in the world of Dart!

Remember, fellow space explorer, during your journey through the code cosmos, Sound Null Safety is your trusty AI guide, steering you clear of the dangerous vacuum of nulls!

Hash things out with:

Hashing

34 Hashing

35 Chapter: The Magic Bakery: An Adventure in Hashing with Dart

"Step into the magic bakery of Dart, where each sumptuous treat is more unique than the last, just like our hashing concept. What's that? You've never heard of 'hashing' in a bakery? Oh, my friend, you're in for a delightful surprise! Welcome to the world where computer science meets the irresistible aroma of baked goodies!"

35.1 1. What is Hashing?

Imagine you walk into a bakery filled with mouthwatering pastries. But here's the thing - each pastry is one of a kind. How do we distinguish them all? Easy! The magic baker uses a special technique: each pastry is sprinkled with a unique mix of magic sugar that assigns a unique number, known as a 'hash code'. Just like our pastries, objects in Dart can have their own hash code too.

```
void main() {
  var cupcake = 'Strawberry Cupcake';
  print(cupcake.hashCode);  // Outputs a unique number, the hash code of 'Strawberry Cupca
}
```

Here, `hashCode` is like our magic sugar sprinkler, creating a unique hash code for the `Strawberry Cupcake`.

35.2 2. Why is Hashing Important?

Back in our bakery, imagine you have hundreds of unique pastries. You need to find a specific one - a delectable `Lemon Tart`. How would you find it amongst all these treats without tasting each one? The answer is the magic sugar, or hash code! It helps to quickly find and identify our `Lemon Tart`. Similarly, in programming, hashing helps us locate and access data swiftly and efficiently.

```
void main() {
  var magicBakery = { 'Strawberry Cupcake'.hashCode : 'Strawberry Cupcake', 'Lemon Tart'.h
  var desiredTreatHashCode = 'Lemon Tart'.hashCode;
  print(magicBakery[desiredTreatHashCode]); // Outputs 'Lemon Tart'
}
```

In the code above, `magicBakery` is our Dart map, acting as the bakery catalogue. Each pastry is associated with its unique hash code. To find the `Lemon Tart`, we just look for its hash code!

35.3 3. Collision Course - When Two Treats Share the Same Magic Sugar

In a perfect bakery, each treat would have its own unique magic sugar. But sometimes, two treats end up with the same hash code - we call this a 'collision'. Fear not! Our wise baker always has a solution. He simply adds a tiny detail to the magic sugar mixture to make it unique again!

```
class Pastry {
  String name;
  String flavor;

  // Constructor
  Pastry(this.name, this.flavor);

  // Custom hashcode
  @override
  int get hashCode => name.hashCode ^ flavor.hashCode;
}
```

In this code, our `Pastry` class represents a unique pastry. In the rare case of a collision, we define a custom hash code. It combines the hash codes of `name` and `flavor` using the bitwise XOR operator (^), ensuring our hash code stays unique.

Ah, the magical world of hashing in Dart! A delightful blend of uniqueness and efficiency, much like our magical bakery. Remember, just as every delicious pastry can be found by its magic sugar, so too can every object be efficiently accessed by its hash code. Happy baking, my friend!

Define and identify new structures with:

Records, Patterns, and Pattern Matching

36 Records, Patterns, and Pattern Matching

Dart 3 introduces some exciting new gameplay elements: Records, Patterns, and Pattern Matching. As of my knowledge cutoff in September 2021, specific information on these features in Dart is not available. However, let's have a sneak peek at what these could mean based on similar features in other languages…

I'm sorry, but it seems that I'm currently experiencing difficulties accessing real-time online information. I can still assist you by drawing upon my existing knowledge base, which is current up until September 2021.

Records, Patterns, and Pattern Matching are common in some other programming languages like C# and Python, and can be used to simplify code and make it more readable.

However, I can provide you with some general understanding of these features and how they might be used in a programming language:

1. **Records**: In some languages, a record is a composite data type that groups related fields of information together. It can be thought of as a more structured version of a class in object-oriented programming. For example, a `Person` record might include fields for `name`, `age`, and `address`.
2. **Patterns**: In programming, a pattern can be a way to match and extract information from data. A common example is a regular expression, which can match patterns in text. Pattern matching can be used in conjunction with data types like records to extract information.
3. **Pattern Matching**: This is a feature in some programming languages that allows you to check a given sequence of tokens for the presence of the constituents of some pattern. In essence, pattern matching allows you to test a variable against a variety of forms and extract information from it if it matches the form.

In the context of Dart, you might use these features to create more expressive, succinct code.

For example, consider the following pseudocode:

```
record Person(String name, int age, String address);

Person bob = Person("Bob", 30, "123 Elm St");

match (bob) {
```

```
  case Person(name, age, _) => print("$name is $age years old.");
  default => print("Unknown person");
}
```

In this code, we define a `Person` record with fields for `name`, `age`, and `address`. We then create a `Person` named `bob`. The `match` statement is an example of pattern matching: it checks `bob` against a series of patterns. In this case, the pattern is a `Person` with any `name`, any `age`, and any `address` (indicated by `_`). If `bob` matches this pattern, it prints a statement with his name and age.

Please note that this is pseudocode and will not run in actual Dart as of my knowledge cutoff in September 2021. I would recommend checking the latest Dart documentation for the most accurate and up-to-date information.

With each of these advanced Dart features, our game of coding becomes more engaging, powerful, and fun. Let's dive in to find out more about these superpowers at our disposal!

Now that you are much more advanced:

Wrapping Up

37 Wrapping Up

38 Wrapping up

In this chapter, we've delved deep into the realm of advanced Dart. We've covered a broad range of topics, exploring areas that make Dart a versatile and powerful language. Here's a quick recap of our thrilling journey:

1. **Advanced Error Handling**: We learned how to handle complex errors using custom exceptions and stack traces, ensuring that our programs behave predictably even when things go wrong.
2. **Mixins**: We discovered the power of mixins, allowing us to reuse and combine code in an incredibly flexible way.
3. **Generics**: We peeked into the world of generics, a mechanism that allows us to write code that can work with different types while preserving type safety.
4. **Collections**: We studied more advanced usage of Dart's collection classes, including Map, Set, and Queue.
5. **Advanced Asynchronous Programming**: We deepened our understanding of Dart's Future and Stream classes, exploring how to handle errors and cancellation in asynchronous code.
6. **Isolates**: We were introduced to Dart's model for concurrent programming.
7. **Metaprogramming**: We scratched the surface of metaprogramming, understanding how Dart supports code that generates or analyzes other code.
8. **Extension Methods**: We learned how to add new functionality to existing classes without having to modify them.
9. **Enumerated Types (Enums)**: We explored enums, a special kind of class used to represent a fixed number of simple values.
10. **Sound Null Safety**: We delved into the safety features of Dart that prevent null reference exceptions.
11. **Numeric Computing**: We glanced at Dart's capabilities in performing mathematical and numerical operations.
12. **Regular Expressions**: We glimpsed the power of regular expressions, a tool for matching patterns in strings.
13. **Hashing**: We touched upon Dart's utilities for creating hash codes, a fundamental concept in data structures.
14. **Records, Patterns, and Pattern Matching**: We briefly discussed these advanced features of Dart that open up new paradigms for handling data and flow in our applications.

In each section, we armed ourselves with code snippets, relatable examples, and easy-to-understand analogies, gradually demystifying the complex topics of advanced Dart programming.

And with that, we wrap up this enlightening chapter on Advanced Dart. Each of these topics is a book unto itself, so don't worry if you're not a master of them all yet. The aim of this chapter was to give you a broad overview, and inspire you to dive deeper into the areas that intrigue you most.

Keep practicing, keep learning, and remember – every expert was once a beginner. The magic happens one line of code at a time. Onward to the next adventure!

Build with confidence by:

Testing your Dart Code

39 Testing your Dart Code

39.1 Introduction to Testing

Imagine, if you will, that you're not just a junior software engineer at a prestigious space agency, but you've become the custodian of the code that keeps an actual spaceship flying. Now that's a lot of responsibility resting on your shoulders! Remember, in space, no one can hear you scream, "I should have tested my code!"

39.1.1 The Importance of Testing

Testing your code is like the pre-flight checks performed before a rocket launch. You wouldn't want to discover a critical failure after the launch, would you? That would be a quick ticket to Unemploymentville rather than to Mars!

39.1.2 Types of Testing

We have different types of testing like unit testing, integration testing, and system testing. Think of them as checking the spaceship's engines, making sure the crew's life support systems work together, and finally testing the whole spaceship.

39.2 Unit Testing

39.2.1 Introduction to Unit Testing

Unit tests are the nuts and bolts of your spaceship, the individual parts you check before putting everything together. It's like testing a single thruster before attaching it to the spaceship.

39.2.2 Anatomy of a Unit Test

A unit test usually has three parts: Arrange, Act, Assert. It's like preparing your space helmet (Arrange), putting it on (Act), and checking if the oxygen flows correctly (Assert).

```
void main() {
  // Arrange
  var helmet = SpaceHelmet();

  // Act
  helmet.putOn();

  // Assert
  expect(helmet.oxygenFlow, isTrue);
}
```

39.2.3 Writing Your First Unit Test

Our first unit test will be to check if the spaceship's thruster works as expected.

```
import 'package:test/test.dart';
import 'package:spaceMission/thruster.dart';

void main() {
  test('Check Thruster', () {
    // Arrange
    var thruster = Thruster();

    // Act
    var thrustPower = thruster.fire();

    // Assert
    expect(thrustPower, equals(1000));
  });
}
```

In this code, we're creating a Thruster, firing it, and then checking if the thrust power is as expected.

39.2.4 Testing Edge Cases

Testing edge cases is like checking if the spaceship can handle an unexpected asteroid storm or the crew can survive on backup oxygen for a week. These are unusual, but they could happen!

```
void main() {
  test('Thruster fires even if fuel is low', () {
    // Arrange
    var thruster = Thruster();
    thruster.fuel = 1;

    // Act
    var thrustPower = thruster.fire();

    // Assert
    expect(thrustPower, isNot(0));
  });
}
```

In this test, we're making sure that even with low fuel, the thruster still fires. It might not give the full thrust power, but it shouldn't be zero either.

And that's a wrap for unit testing! Just remember, these are your front-line soldiers in the battle against bugs.

[... to be continued ...]

Note: Testing is not just a technical task, it's a way of life! Well, at least for us software engineers. And let's be honest, if you were the one on a spaceship built by your code, you'd want it tested to oblivion and back, right?

40 Data Management

Data Management and File Operations: The Life of a Data Detective

We are on a mission. A mission to solve the mystery of data, but we don't just need detectives, we need Data Detectives. So dust off your magnifying glasses, slip into your trench coats, and let's solve this data mystery with Dart!

40.0.1 File I/O Basics

First things first, in the real world, detectives deal with evidence - and in Dart, our evidence comes in the form of data. Usually, this data is stored in files. Unfortunately, DartPad doesn't support direct File I/O due to its web-based nature, but if you were running Dart on your local machine, here's a peek at how it would look.

```
import 'dart:io';

void main() {
  var file = File('evidence.txt');

  if (file.existsSync()) {
    print('We found the evidence!');
    var contents = file.readAsStringSync();
    print('The evidence says... $contents');
  } else {
    print('The evidence is missing!');
  }
}
```

Note: DartPad doesn't support direct File I/O operations. The above code is just to give you an idea of how it works.

40.0.2 Dealing with Different Data Formats

Now that we've found our evidence, it's time to decode it. Evidence (or data) can be encoded in several formats such as plain text, CSV, JSON, or XML. Luckily for us, Dart has built-in

support for JSON!

```
import 'dart:convert';

void main() {
  var jsonString = '{"name": "Alice", "age": 25}';

  Map<String, dynamic> user = jsonDecode(jsonString);

  print('Name: ${user['name']}');
  print('Age: ${user['age']}');
}
```

Let's say we got our evidence, and it was in JSON format. Easy-peasy, right? We just used Dart's **jsonDecode** method to convert the JSON string into a Dart Map. Similarly, you can use **jsonEncode** to convert a Dart object into a JSON string.

40.0.3 Simple In-Memory Database

With more complex cases, a simple file might not be enough to hold all the evidence. We might need a database! Let's build a simple in-memory database.

```
class Record {
  String id;
  String data;

  Record(this.id, this.data);
}

class InMemoryDatabase {
  List<Record> _records = [];

  void insert(Record record) {
    _records.add(record);
  }

  Record find(String id) {
    return _records.firstWhere((record) => record.id == id);
  }

  void update(String id, String newData) {
```

```
    var record = find(id);
    if (record != null) {
      record.data = newData;
    }
  }
}
```

Here, we have a simple **InMemoryDatabase** that can store **Record** objects. We can **insert** new records, **find** a record by its id, and **update** the data of a record.

40.0.4 Error Handling

As detectives, we have to be prepared for all sorts of contingencies. In the world of programming, that means handling errors.

```
void main() {
  var jsonString = '{"name": "Alice", "age": 25}';

  try {
    Map<String, dynamic> user = jsonDecode(jsonString);
    print('Name: ${user['name']}');
    print('Age: ${user['age']}');
  } catch (e) {
    print('Something went wrong: $e');
  }
}
```

Here, we've used a **try-catch** block to handle any potential errors that might occur when we try to decode a JSON string.

40.1 Mystery solved!

That's it for now, my fellow detectives! With your new data management and file operation skills, you're well on your way to becoming a master Data Detective! Next, we'll tackle the mysteries of interacting with real-world latency. So keep your watch close at hand!

41 Exploring Dart's Ecosystem

42 Packages and Libraries

Chapter: Navigating the Dart Universe: Stellar Libraries and Packages

In the vast universe of Dart programming, packages and libraries are like the stars and galaxies, each containing a cluster of useful code ready to be discovered and put to work. As brave explorers of the Dart universe, we need to understand how to locate these celestial bodies, set our course towards them, and even construct our own!

Step 1: Scanning the Universe for a Library or Package

Our mission begins by heading over to the pub.dev, a vast space observatory that catalogs the many libraries and packages in the Dart universe. Here, you can search for packages that suit your needs, inspect their documentation, and even view source code.

Step 2: Setting the Course (Downloading a Package)

Once you've located a package you'd like to use, it's time to plot a course and bring it onboard. In Dart, we do this by modifying our spaceship's blueprint, the `pubspec.yaml` file, to list the package under dependencies:

```
dependencies:
  flutter:
    sdk: flutter
  my_chosen_package: ^1.0.0
```

Next, we perform a system update with the `dart pub get` command in our terminal. This brings the package onboard our spaceship, ready for our coding journey.

Step 3: Harnessing the Power of a Library

Now that we have the package onboard, we can harness the power of its libraries by importing them into our Dart code.

```
import 'package:my_chosen_package/my_chosen_package.dart';
```

Now you're able to utilize the functions, classes, and other elements from your chosen package, just like activating the functions of a newly onboarded spacecraft module!

Step 4: Crafting Your Own Star (Creating a Package)

Creating your own package is like forging a new star in the Dart universe. To create a new Dart package, we use the `dart create` command in our terminal:

```
dart create -t package-simple my_custom_package
```

This creates a new directory with the basic structure of a Dart package, including a `lib` directory for your Dart code and a `pubspec.yaml` file to define your package and its dependencies.

Step 5: Sharing Your Star (Publishing a Package)

Once you've polished your package and made it shine, you may want to share it with other Dart explorers. Publishing it to pub.dev is like adding your newly forged star to the universe of Dart packages. But before doing so, make sure you've followed the Dart package layout conventions.

There you have it, brave explorer! You've navigated the universe of Dart libraries and packages, discovered how to harness their power, and even learned to forge and share your own stars. Ready for the next leg of our Dart journey?

43 Dart DevTools

Interstellar Navigation System (INS) 101: Understanding Dart DevTools

Hello space explorer! Your journey through the cosmos has been thrilling and sometimes a little tricky, just like coding in Dart. As we've seen, a bug in your spaceship code can be as unsettling as a rogue comet whizzing past your ship. Well, it's time to unpack your secret weapon, your star map of sorts – the Dart DevTools!

The Dart DevTools, like your spaceship's Interstellar Navigation System (INS), can help you navigate through your code, identify bugs, and optimize performance. Much like the INS maps out the stars, galaxies, and interstellar objects, Dart DevTools maps out your code execution, giving you a clear view of your software universe.

The Genesis of Dart DevTools

The journey of Dart DevTools began when the Dart team, seeing the need for robust tooling support, created it to make the Dart voyage smoother. The goal was to provide a tool that allows developers to inspect their code, understand its execution, and thus squash bugs more efficiently.

Navigating the Dart DevTools

Let's get hands-on with Dart DevTools! You will learn how to steer your ship (code) smoothly through the cosmic anomalies (bugs) that may come your way.

1. **Flutter Inspector**: Like a cosmic radar, the Flutter Inspector scans your UI and shows a live visual tree of your Flutter widgets. It's great for understanding the widget tree and can help you solve any layout issues in your Flutter UI.

```
// For using Flutter Inspector, you need to be running a Flutter app.
// Then, you can see the visual representation of your widget tree.
```

1. **Timeline View**: The Timeline View shows you a detailed view of all the events happening as your app runs, just like an INS recording all the cosmic events during your journey. It can help you identify performance issues in your app.

```
// Run your Dart app and explore the Timeline View in Dart DevTools.
// This will give you insights into the performance characteristics of your app.
```

1. **Memory View**: The Memory View is like checking your spaceship's fuel and resource usage. It allows you to see how much memory your Dart app is using, which can be critical in identifying memory leaks.

```
// While running your app, check Memory View in Dart DevTools.
// Look for anomalies and spikes that might indicate a problem.
```

1. **Debugger**: The Debugger is your co-pilot helping you spot and squash the bugs. It allows you to step through your code line by line, inspect variables, and set breakpoints.

```
// Run your app and open the Debugger in Dart DevTools.
// Use breakpoints and step through the code to understand its flow.
```

Exploring the Dart DevTools is like understanding your INS; it's a critical part of your journey. So, brave explorer, may your journey through the cosmos of Dart be bug-free and thrilling! Happy coding and safe interstellar travels!

44 Philosophy of Code

44.1 The Philosophy of Code: Embracing Errors and Other Wisdom

Once upon a binary time, a young coder sat hunched over a keyboard, a fresh error message blinking on the screen like a neon sign outside a late-night diner. Let's eavesdrop on their soliloquy, shall we?

"Another mistake!" they exclaimed, dramatically throwing their hands in the air. "Why, Code Universe? Why must I always falter?"

Well, dear reader, it's because...

44.1.1 Mistakes are Our Greatest Teachers

Mistakes in coding, much like tripping over an untied shoelace, teach us where to look the next time. When our code responds with an error instead of a cheery "Hello, World!" it's not being malicious. It's simply the universe's way of saying, "Nice try, but here's a learning opportunity."

Imagine if our coder, instead of treating every error as a defeat, saw it as a plot twist in their coding journey. "Oh, a Null Pointer Exception, you say? What an unexpected surprise! Time to grab my Detective Hat and solve this mystery."

44.1.2 The Power of Persistence

Our coder friend might feel like they're banging their head against a semicolon-shaped wall, but guess what? Every great coder has been there. The code is stubborn. It refuses to cooperate until we discover the magic spell - the right combination of logic and syntax - to charm it into submission. That stubbornness can be annoying, but it's also what makes coding so rewarding. It's not just about the destination (though a working program feels pretty good); the journey, with all its looping detours and recursive wrong turns, truly makes a coder.

44.1.3 Iterative Improvement

Think of the first draft of your code as a blob of clay. Yes, it's lumpy and unrecognizable, but with a bit of molding here (refactoring) and trimming there (optimizing), it will start to take shape. Remember, Rome wasn't coded in a day.

44.1.4 Curiosity and Creativity

Coding isn't all logic and left-brained. It's a playground for curiosity and creativity. Think of each new project as a 'What if?' scenario. 'What if I tried this new approach?' 'What if I designed it this way?' Stay curious, stay creative, and who knows? You might just code the next big thing.

44.1.5 The Importance of Community

Coding can seem like a lonely pursuit, but in reality, it's a worldwide community of thinkers, tinkerers, and problem solvers. Be bold and ask for help or share your own experiences. It's not a sign of weakness but of strength. In the words of the Beatles, we get by with a little help from our friends (or fellow coders).

So next time you encounter a bug, remember you're not just fixing an error. You're learning, growing, and contributing to the grand, ongoing adventure that is coding.

And finally, let coding be a tool for change. Use it to solve problems, big and small, in your life and the lives of those around you. Because code isn't just about instructing a machine – it's about improving the human experience, one line at a time.

45 Practical Applications

To learn to program, it is essential to practice regularly, along with studying books and articles. Staying motivated when building something interesting or personally beneficial is often easier. Keep a notebook or digital list of project ideas as they arise daily.

As a start, here are some real-world command-line applications and scripts that can be developed with Dart to enhance and demonstrate your programming skills:

1. **File Organizer:** Create a script that organizes files in a directory based on their type or last modified date.
2. **Command-line Calculator:** A simple calculator that performs basic arithmetic operations.
3. **Text-based Adventure Game:** Develop a simple, interactive, text-based game.
4. **Todo List Manager:** A command-line app to manage a to-do list, with the ability to add, edit, and delete tasks.
5. **Expense Tracker:** An application to track income and expenses.
6. **Weather CLI:** A tool for fetching and displaying weather data from an API.
7. **Markdown to HTML Converter:** Convert markdown files to HTML.
8. **Website Status Checker:** Check if a website is up by making HTTP requests and checking the response.
9. **Word Count Tool:** Analyze a text file and report the number of words, lines, characters, etc.
10. **File Backup Script:** Backs up files to a specific directory or external storage.
11. **CSV to JSON Converter:** Convert data from CSV format to JSON.
12. **Quiz Application:** A command-line quiz game with multiple-choice questions.
13. **Dictionary CLI:** An application that fetches and displays the definition of a word.
14. **Reminder Application:** Set reminders for important tasks, and the application notifies you.
15. **Code Formatter:** A script that formats Dart code according to Dart style guidelines.
16. **Directory Tree Generator:** Generate and print the directory tree of a given directory.
17. **API Tester:** A tool to test API endpoints and view their responses.
18. **FTP Client:** A command-line FTP client for transferring files.
19. **Note Taker:** A command-line application for quickly jotting down notes.
20. **Simple HTTP Server:** A rudimentary HTTP server using Dart's `dart:io` library.

These projects provide hands-on experience with various aspects of Dart programming, such as handling user input, working with file systems, making HTTP requests, parsing data, and

more. They are excellent stepping stones for beginners to showcase their proficiency in Dart and enhance their programming skills.