Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x produces the same result f(x) each time. Here are some of the key concepts of functional programming: **Functions as first-class citizens:** In functional programming, functions are treated as first-class citizens, which means they can be passed around as values, stored in variables, and returned from other functions. **Pure functions:** A pure function is a function that always returns the same output for the same input. Pure functions have no side effects, which means they do not modify any global state. **Immutability:** Immutable data is data that cannot be changed once it is created. Immutability is a key feature of functional programming, as it makes code more reliable and easier to reason about.

Lets look at an example (this one uses named functions - means functions that have names)

```
func main() {
	// Create a channel to send and receive numbers.
	numbers := make(chan int)
	// Create a goroutine to generate numbers.
	Go generateNumbers(numbers)
	// Create a goroutine to sum the numbers.
	go sumNumbers(numbers)
	// Wait for the goroutines to finish.
	<-numbers
	<-numbers
}
func generateNumbers(numbers chan int) {
	for i := 0; i < 10; i++ {
		numbers <- i
	}
	close(numbers)
}
func sumNumbers(numbers chan int) {
	var sum int
	for number := range numbers {
	sum += number
	}
	fmt.Println("The sum of the numbers is", sum)
}
```

Step by Step explanation:

Functions as first-class citizens means that functions can be passed around as values, stored in variables, and returned from other functions. This makes it easy to compose functions and create complex programs from simple building blocks.

In this program, the **generateNumbers()** and **sumNumbers()** functions are both passed around as values, stored in variables, and returned from other functions.

Pure functions are functions that always return the same output for the same input. Pure functions have no side effects, which means they do not modify any global state. This makes pure functions easier to reason about and test.

In this program, the **generateNumbers()** and **sumNumbers()** functions **are both pure functions**. They do not modify any global state, and they always return the same output for the same input.

Immutable data is data that cannot be changed once it is created. Immutable data makes code more reliable and easier to reason about.

In this program, the **numbers channel is immutable**. Once it is created, it cannot be changed.

Here is a step-by-step explanation of how the program works:

1. The **main() function** creates a channel to send and receive numbers.
2. The **main() function** creates two goroutines: one to generate numbers and one to sum the numbers.
3. The **generateNumbers() goroutine** generates numbers and sends them to the numbers channel.
4. The **sumNumbers() goroutine** receives numbers from the numbers channel and sums them.
5. *The main() function waits for the goroutines to finish.*
6. *The main() function prints the sum of the numbers.*

Please note: go lang is not a functional programming language but you can do numerous things by following functional programming style. It makes the program more reliable.

Another example:

Let's consider a system for a car rental company, and we need to **filter out cars based on different conditions**. For example, you might want to find all cars that are available for rent, or all cars of a specific model. Here's how you could do that in Go:

Step 1: Define the Car structure and a slice of Cars.

```
type Car struct {
        Model string
        IsRented bool
}
cars := []Car{
{"Toyota", true}, {"Honda",
false}, {"BMW", false},
{"Audi", true}, {"Mercedes",
false},
}
```

Step 2: Define the Predicate functions. These functions will return true if a car is available for rent and if a car is a specific model, respectively.

```
isAvailable := func(car Car) bool {
        return !car.IsRented }
        isHonda := func(car Car) bool {
        return car.Model == "Honda"
}
```

Step 3: Define the Filter function. This function takes a slice of Cars and a predicate function, and returns a new slice containing only the cars that satisfy the predicate.

```
filterCars := func(cars []Car, predicate func(Car) bool) []Car
{
        result := []Car{}
        for _, car := range cars {
        if predicate(car) {
        result = append(result, car)
        }
}
return result
}
```

Step 4: Use the Filter function to filter out available cars and Hondas.

```
availableCars := filterCars(cars, isAvailable)
hondaCars := filterCars(cars, isHonda)
```

Step 5: Print the results.

```
fmt.Println(availableCars)
// prints: [{Honda false} {BMW false} {Mercedes false}]
fmt.Println(hondaCars) // prints: [{Honda false}]
```

This code is using the same basic filter function concept as before, but applied to a slice of a more complex type (Car structs) and using more meaningful predicate functions. This example shows how you could create a flexible system for finding cars that meet different criteria without having to write separate functions for each possible criterion. By using higher-order functions and treating functions as first-class values, you can create more modular, reusable, and maintainable code.

When we write our programs, we will use a mix of regular functions and some pure functions but it is always good to think of functions as first class entities.
Generally when students graduate ... they are used to passing variables as reference or as value but they are not used to passing functions around. Functional programming shows how you can pass functions around. Under the hood, passing functions around means ..passing a pointer to the function (C++ lingo).

Another Example: let's consider a real-world e-commerce scenario where a website needs to process a large number of orders. For each order, the system needs to verify inventory, process the payment, and update the order status. We can simulate this with goroutines and channels. Each order can be processed independently in its own goroutine, and channels can be used to communicate the status of each step. Lets look at the pipeline of tasks:

```
package main import ( "
        fmt"
        "time"
)
```

```
type Order struct {
	ID int
	IsCompleted bool
}
func checkInventory(order Order, inventoryChan chan Order) {
// simulate inventory check delay
	time.Sleep(1 * time.Second)
	 inventoryChan <- order
 }
func processPayment(inventoryChan chan Order, paymentChan chan Order) {
	order := <-inventoryChan
	// simulate payment processing delay
	 time.Sleep(1 * time.Second)
	paymentChan <- order
}
func updateOrderStatus(paymentChan chan Order, statusChan chan Order) {
	order := <-paymentChan
	// simulate status update delay
	time.Sleep(1 * time.Second)
	order.IsCompleted = true
	statusChan <- order
}
func main() {
	inventoryChan := make(chan Order)
	paymentChan := make(chan Order)
	statusChan := make(chan Order)
	orders := []Order{{1, false}, {2, false}, {3, false}}
	for _, order := range orders {
		go checkInventory(order, inventoryChan)
		go processPayment(inventoryChan, paymentChan)
		go updateOrderStatus(paymentChan, statusChan)
		}
	for range orders {
	order := <-statusChan
	fmt.Printf("Order %d has been completed\n", order.ID)
	}
}
```

In this example:

- processPayment and updateOrderStatus are waiting for orders from their respective input channels (inventoryChan and paymentChan), and that's achieved by the <- operator in the functions processPayment and updateOrderStatus. This operator is used to receive data from the channel. If there is no data available on the channel, it blocks and waits until data is available.
- checkInventory, processPayment, and updateOrderStatus are launched as separate goroutines for each order. The first function checkInventory doesn't wait and pushes the data into inventoryChan which is caught in processPayment function, then processPayment pushes data into paymentChan which is then caught by updateOrderStatus function and finally, the order's status is updated and it's pushed into statusChan.
- After launching all the goroutines, we receive from the statusChan in a loop, printing a message for each completed order.

So the order of operations is:

1. Launch goroutines.
2. checkInventory puts an order in inventoryChan.
3. processPayment takes an order from inventoryChan, processes it, and puts it in paymentChan.
4. updateOrderStatus takes an order from paymentChan, processes it, and puts it in statusChan.
5. Main function takes an order from statusChan and prints it.

This sequence forms a pipeline where each stage is a goroutine that receives from one channel, performs an operation, and sends to another channel. Each stage of the pipeline blocks until it receives the data it needs, does its processing, and passes the result on. The blocking nature of channel operations (both send and receive) is what synchronizes the goroutines.

Note: It is very important to understand how channels work ... CHANNELS can be used very effectively to communicate between go routines. One go Routine can SEND data to the CHANNEL, and the second go Routine can RECEIVE data from the channel. Channels can be buffered or unbuffered. Understand the difference properly else you will not understand the above program.