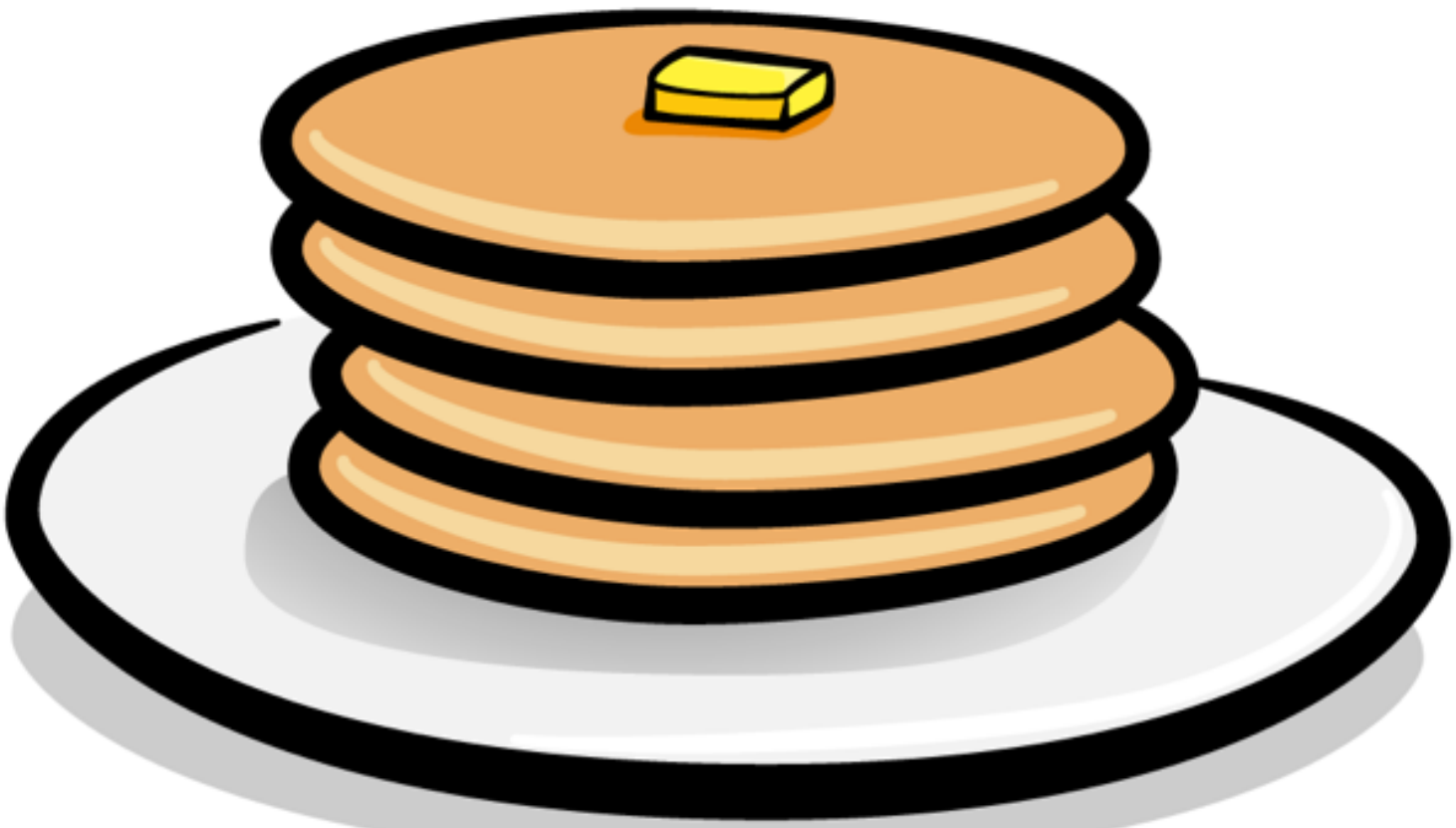
4 Stack Data Structures Written by Márton Braun

Stacks are everywhere. Some common examples of things you might stack:

- Pancakes.
- Books.
- Paper.
- Cash, especially cash. :]

The **stack** data structure is identical, in concept, to a physical stack of objects. When you add an item to a stack, you place it on top of the stack. When you remove an item from a stack, you always remove the top-most item.



Good news: A stack of pancakes. Bad news: You may only eat the top-most pancake.

Stack operations

Stacks are useful, and also exceedingly simple. The main goal of building a stack is to enforce how you access your data. If you had a tough time with the linked list concepts, you'll be glad to know that stacks are comparatively trivial.

There are only two essential operations for a stack:

- **`push`**: Adding an element to the top of the stack.
- **`pop`**: Removing the top element of the stack.

This means that you can only add or remove elements from one side of the data structure. In computer science, a stack is known as the **LIFO** (last-in first-out) data structure. Elements that are pushed in last are the first ones to be popped out.

To get this in code, open the starter project, create the **Stack.kt** file in the **stack** package, and write the following code:

```
interface Stack<T : Any> {
  fun push(element: T)
  fun pop(): T?
}
```

> **Note**: The previous `Stack` interface is different from the `Stack` class provided by Kotlin (or Java) which extends the `Vector` class and provides methods we don't need here.

Stacks are used prominently in all disciplines of programming, such as:

- Android uses the *fragment stack* to push and pop fragments into and out of an Activity.
- Memory allocation uses stacks at the architectural level. Memory for local variables is also managed using a stack.
- *Search and conquer* algorithms, such as finding a path out of a maze, use stacks to facilitate backtracking.

Implementation

You can implement your Stack interface in different ways and choosing the right storage type is important. The `ArrayList` is an obvious choice since it offers constant time insertions and deletions at one end via `add` and `removeAt` with the last index as a parameter. Usage of these two operations will facilitate the **LIFO** nature of stacks.

In the same **Stack.kt** file you can then start your implementation with the following code:

```
class StackImpl<T : Any> : Stack<T> {
  private val storage = arrayListOf<T>()

  override fun toString() = buildString {
    appendLine("----top----")
    storage.asReversed().forEach {
      appendLine("$it")
    }
    appendLine("-----------")
  }
}
```

You define a private property of type `ArrayList` for the data and you override the `toString` method in order to display its content for debug purposes. With this code, you'll get some errors because of the missing implementation of the `push` and `pop` operations but you're going to fix this soon.

push and pop operations

Add the following two operations to your `Stack`:

```
override fun push(element: T) {
  storage.add(element)
}

override fun pop(): T? {
  if (storage.size == 0) {
    return null
  }
  return storage.removeAt(storage.size - 1)
}
```

In the the `push` method you just append the value passed as parameter to the end of the `ArrayList` using its `add` method. In the `pop` method you simply return `null` if the storage is empty or you remove and return the

last element you have inserted.

It's time to see the stack at work. Open **Main.kt** and write this code in `main()`:

```
"using a stack" example {
  val stack = StackImpl<Int>().apply {
    push(1)
    push(2)
    push(3)
    push(4)
  }
  print(stack)
  val poppedElement = stack.pop()
  if (poppedElement != null) {
    println("Popped: $poppedElement")
  }
  print(stack)
}
```

You'll see the following output:

```
---Example of using a stack---
----top----
4
3
2
1
-----------
Popped: 4
----top----
3
2
1
-----------
```

`push` and `pop` both have an ***O*(1)** time complexity.

Non-essential operations

Next, you'll add some nice-to-have operations that make stacks easier to use.

In **Stack.kt**, add the following code to the `Stack` interface :

```
fun peek(): T?

val count: Int
  get

val isEmpty: Boolean
  get() = count == 0
```

`peek` is an operation that's often attributed to the stack interface. The idea of `peek` is to look at the top element of the stack without mutating its contents. The `count` property returns the number of element in the `Stack` and it's used to implement the `isEmpty` property.

You now need to add the implementation to the `StackImpl` class with this code:

```
override fun peek(): T? {
  return storage.lastOrNull()
}

override val count: Int
  get() = storage.size
```

This allows you to have cleaner code, changing the implementation of the `pop` method like this:

```
override fun pop(): T? {
  if (isEmpty) {
    return null
```

```
  }
  return storage.removeAt(count - 1)
}
```

Less is more

You may have wondered if you could adopt the Kotlin collection interfaces for the stack. A stack's purpose is to limit the number of ways to access your data, and adopting interfaces such as `Iterable` would go against this goal by exposing all of the elements via iterators. In this case, less is more!

You might want to take an existing list and convert it to a stack so that the access order is guaranteed. Of course, it would be possible to loop through the array elements and `push` each element. However, you can write a **static factory method** that directly adds these elements to the Stack implementation.

Add the following code to `StackImpl` class:

```
  companion object {
    fun <T : Any> create(items: Iterable<T>): Stack<T> {
      val stack = StackImpl<T>()
      for (item in items) {
        stack.push(item)
      }
      return stack
    }
  }
```

Now, add this example to the `main()` function:

```
"initializing a stack from a list" example {
  val list = listOf("A", "B", "C", "D")
  val stack = StackImpl.create(list)
  print(stack)
  println("Popped: ${stack.pop()}")
```

```
}
```

This code creates a stack of strings and pops the top element `"D"`. Notice that the Kotlin compiler can type infer the element type from the list so you can use `Stack` instead of the more verbose `Stack<String>`.

You can go a step further and make your stack initializable by listing its elements, similar to `listOf()` and other standard library collection factory functions. Add this to **Stack.kt**, outside the `Stack` class definition:

```
fun <T : Any> stackOf(vararg elements: T): Stack<T> {
  return StackImpl.create(elements.asList())
}
```

Now, go back to `main()` and add:

```
"initializing a stack from an array literal" example {
  val stack = stackOf(1.0, 2.0, 3.0, 4.0)
  print(stack)
  println("Popped: ${stack.pop()}")
}
```

This creates a stack of `Double`s and pops the top value 4.0. Again, type inference saves you from having to specify the generic type argument of the `stackOf` function call.

Stacks are crucial to problems that *search* trees and graphs. Imagine finding your way through a maze. Each time you come to a decision point of left, right or straight, you can push all possible decisions onto your stack. When you hit a dead end, backtrack by popping from the stack and continuing until you escape or hit another dead end.

Challenges

A stack is a simple data structure with a surprisingly large amount of

applications. Complete the following challenges to see what it can do.

Challenge 1: Reverse a LinkedList

Given a linked list, print the nodes in reverse order. You should not use recursion to solve this problem.

Solution 1

One of the prime use cases for stacks is to facilitate backtracking. If you push a sequence of values into the stack, sequentially popping the stack will give you the values in reverse order:

```
fun <T : Any> LinkedList<T>.printInReverse() {
  val stack = StackImpl<T>()

  for (node in this) {
    stack.push(node)
  }

  var node = stack.pop()
  while (node != null) {
    println(node)
    node = stack.pop()
  }
}
```

Here's how it works:

1. Copy the content of the list into a stack, carefully putting the nodes on top of each other.
2. Remove and print the nodes from the stack one by one, starting from the top.

The time complexity of pushing the nodes into the stack is ***O*(*n*)**. The time complexity of popping the stack to print the values is also ***O*(*n*)**. Overall, the time complexity of this algorithm is ***O*(*n*)**.

Since you're allocating a container (the stack) inside the function, you also incur an ***O*(*n*)** space complexity cost.

Challenge 2: The parentheses validation

Check for balanced parentheses. Given a string, check if there are `(` and `)` characters, and return `true` if the parentheses in the string are balanced. For example:

```
// 1
h((e))llo(world)() // balanced parentheses

// 2
(hello world // unbalanced parentheses
```

Solution 2

To check if there are balanced parentheses in the string, you need to go through each character of the string. When you encounter an opening parenthesis, you'll push that into a stack. Then, if you encounter a closing parenthesis, you should pop the stack.

Here's the code:

```
fun String.checkParentheses(): Boolean {
  val stack = StackImpl<Char>()

  for (character in this) {
    when (character) {
      '(' -> stack.push(character)
      ')' -> if (stack.isEmpty) {
        return false
      } else {
        stack.pop()
      }
    }
  }
```

```
  return stack.isEmpty
}
```

Here's how it works:

1. Create a new stack and start going through your string, character by character.
2. Push every opening parenthesis into the stack.
3. Pop one item from the stack for every closing parenthesis, but if you're out of items on the stack, your string is already imbalanced, so you can immediately return from the function.
4. In the end, a balanced string is one that has popped all of the opening parentheses it's pushed (and not a single item more). That would leave the stack empty because you popped all the parentheses you pushed before.

The time complexity of this algorithm is ***O*(*n*)**, where *n* is the number of characters in the string. This algorithm also incurs an ***O*(*n*)** space complexity cost due to the usage of the `Stack` data structure.

Key points

- Despite its simplicity, the stack is a key data structure for many problems.
- The only two essential operations for the stack are the **push** method for adding elements and the **pop** method for removing elements.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.