We'll implement a Stack as a list, so let's use a type synonym to treat them equivalently. This means we can pattern-match on a Stack as though it were a List.

```
type Stack a = [a]
```

The empty Stack is the empty list.

```
emptyStack :: Stack a
emptyStack = []
```

Popping a stack gives you the stack after popping, as well as the element popped. Use a tuple to return those two things.

```
pop :: Stack a -> (Stack a, a)
pop (x:stk) = (stk, x)
```

It's an error to pop an empty list.

```
pop [] = error "Cannot pop an empty list"
```

Peeking at the top of a stack returns the element and leaves the rest of the Stack unchanged.

```
peek :: Stack a -> a
peek (x:_) = x
```

We may push something onto a Stack, which creates a new Stack.

```
push :: a -> Stack a -> Stack a
push x stk = (x:stk)
```

To be able to check if a Stack contains a certain item requires us to be able to test them for equality, so we put a type constraint here.

```
stackContains :: (Eq a) => a -> Stack a -> Bool
```

The empty stack contains nothing. We could use "emptyStack" instead of "[]" but the patterns would overlap.

```
stackContains x [] = False
```

We might think about pattern-matching the head of the stack against the item being checked, as in this code example:

```
stackContains x (x:_) = True
```

But Haskell won't let you pattern-match two things into the same variable like this. Instead we must use guards.

```
stackContains x (y:stk)
   | x == y    = True
```

If the head of the stack doesn't contain the item, check the rest of the stack.

```
   | otherwise = stackContains x stk
```

Here's a sample test. If you push something onto a stack it should not be empty.

```
testPush = peek (push 3 emptyStack) == 3
```

We'll use this stack [3,4,5] for our next few tests.

```
example :: Stack Int
example = push 3 (push 4 (push 5 emptyStack))
```

Peeking should give us the last thing pushed onto the stack.

```
testPeek = peek example == 3
```

If we pop 3 off the Stack, peeking at the result should give us 4.

```
testPop =
   let (restOfStack, head) = pop example
   in head == 3 && (peek restOfStack) == 4
```

Thie following code example is how we could run all the tests at once…

```
testAll = testPush && testPeek && testPop
```

… the following is equivalent. It returns True if everything in the list evaluates to True, and false otherwise.

```
testAll = and [testPush, testPeek, testPop]
```

You can load this file in GHCi like any normal .hs file. If you want to generate a .pdf you can install pandoc and use the following command: pandoc Stacks.lhs -o Stacks.pdf