

## Stacks

A stack is an abstract data type based on the principle of *Last In First Out* (LIFO). Just think about a stack of books or dishes. A stack supports the following operations:

- Push something on the top.
- Look at the top.
- Pop something off the top of the stack.

When we do not care about how these operations are implemented, and just want to express the fact that a stack must provide these operations, we make use of an *interface*. In Java, this looks as follows:

```
public interface Stack<AnyType> {
    public void clear();
    public boolean isEmpty();
    public AnyType top();
    public AnyType pop();
    public void push(AnyType x);
}
```

Since `Stack` provides only names and argument types for methods, it is obviously not enough to implement a stack. We will have to define a `class`, for instance `DsStack`, that *implements* the `Stack` interface.

In Java, we express that a class *implements* an interface like this:

```
public class DsStack<AnyType> implements Stack<AnyType> {
    // implementation
}
```

Classes that implement an interface must provide an implementation for all methods declared in the interface. So interfaces capture the common characteristics and behavior of the classes that implement the interface. Note that we can have various different implementations of the same interface — in our example, we might have a stack implemented as an array, or as a linked list, or in some other way.

In the following code we use a `Stack`:

```
Stack<Integer> stack = new DsStack<Integer>();
stack.push(5);
int k = stack.top();
stack.pop();
```

Since `Stack` is an `interface`, not a `class`, it is impossible to construct objects of type `Stack` (you get the error message “Class is abstract”). However, we can have variables of type `Stack`, and such a variable can reference any object that implements the `Stack` interface. In the example above, the variable `stack` is of type `Stack<Integer>`, and it references an object of type `DsStack<Integer>`.

## Stack frames

The Java runtime system stores information in two separate parts of memory: the stack and the heap.

The *heap* stores all objects and all static variables.

The *stack* stores all local variables of methods (and parameters of methods are just local variables).

When a method is called, the Java Virtual Machine creates a *stack frame* (also known as an *activation record*) that stores the parameters and local variables for that method. One method can call another, which can call another, and so on, so the JVM maintains a *stack* of stack frames.

The `main` method that was called to start your program is at the bottom of the stack, the method that is currently executing (at any point in time) is the one whose stack frame is on top. All the other stack frames represent methods waiting for the methods above them to return before they can continue executing.

When a method finishes executing, its stack frame is erased from the top of the stack, and its local variables are erased forever.

When a method calls itself recursively, the JVM’s internal stack holds two or more stack frames for that method. Only the top one can be accessed. For instance, calling `factorial(3)` for this method:

```

public static long factorial(int n)
{
    if (n <= 1)    // base case
        return 1;
    else
        return n * factorial(n - 1);
}

```

will lead to three stack frames for `factorial` being on the stack:

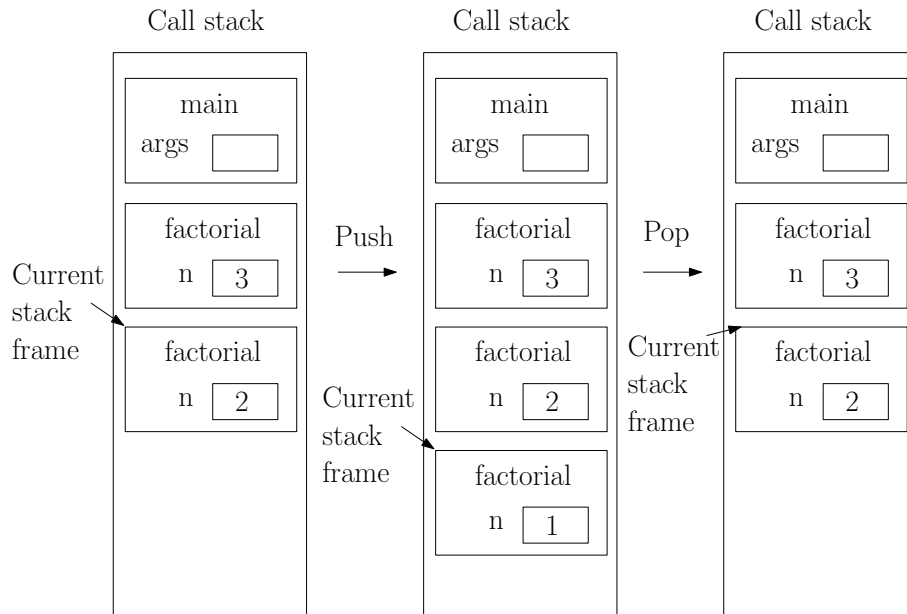


Figure 1: A call stack state: `factorial(3)`

When an exception is thrown, the Java runtime system uses the call stack to find the method where execution can continue. It pops stack frames until a stack frame is found that indicates that it catches the exception.

The `java.lang` package has a method `Thread.dumpStack` that prints a list of the methods on the stack (but it doesn't print their local variables). This method can be convenient for debugging—for instance, when you're trying to figure out which method called another method with illegal parameters that made it crash.

Most operating systems give a program enough stack space for a few thousand stack frames. If you use a recursive procedure to walk through a million-node list, Java will try to create a million stack frames, and the stack will run out of space. The result is a run-time error. You should use iteration instead of recursion when the recursion will be very deep.