# 6.6 Method-Call Stack and Activation Records

To understand how Java performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's placed at the *top*—referred to as **pushing** the dish onto the stack. Similarly, when a dish is removed from the pile, it's removed from the top—referred to as **popping** the dish off the stack. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

# 6.6.1 Method-Call Stack

One of the most important mechanisms for computer science students to understand is the **method-call stack** (sometimes referred to as the **program-execution stack**). This data structure—working "behind the scenes"—supports the method-call/return mechanism. It also supports the creation, maintenance and destruction of each called method's local variables. Last-in, first-out (LIFO) behavior is *exactly* what a method needs in order to return to the method that called it.

# 6.6.2 Stack Frames

As each method is called, it may, in turn, call other methods, which may, in turn, call other methods—all *before* any of the methods return. Each method eventually must return control to the method that called it. So, somehow, the system must keep track of the *return addresses* that each method needs in order to return control to the method that called it. The method-call stack is the perfect data structure for handling this information. Each time a method calls another method, an entry is *pushed* onto the stack. This entry, called a **stack frame** or an **activation record**, contains the *return address* that the called method needs in order to return to the calling method. It also contains some additional information we'll soon discuss. If the called method simply returns instead of calling another method before returning, the stack frame for the method call is *popped*, and control transfers to the return address in the popped stack frame.

The beauty of the call stack is that each called method *always* finds the information it needs to return to its caller at the *top* of the call stack. And, if a method makes a call to another method, a stack frame for the new method call is simply *pushed* onto the call stack. Thus, the return address required by the newly called method to return to its caller is now located at the *top* of the stack.

# 6.6.3 Local Variables and

# Stack Frames

The stack frames have another important responsibility. Most methods have local variables—parameters and any local variables the method declares. Local variables need to exist while a method is executing. They need to remain active if the method makes calls to other methods. But when a called method returns to its caller, the called method's local variables need to "go away." The called method's stack frame is a perfect place to reserve the memory for the called method's local variables. That stack frame exists as long as the called method is active. When that method returns—and no longer needs its local variables—its stack frame is *popped* from the stack, and those local variables no longer exist.

# 6.6.4 Stack Overflow

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the method-call stack. If more method calls occur than can have their activation records stored on the method-call stack, a fatal error known as **stack overflow** occurs1—this typically is caused by infinite recursion (Chapter 18).

1. This is how the website `stackoverflow.com` got its name. This is a great website for getting answers to your programming questions.