Part II: Software

Any sufficiently advanced technology is indistinguishable from magic.
—Arthur C. Clarke (1962)

To which we add: "*and any sufficiently advanced magic is indistinguishable from hard work, behind the scene*". In Part I of the book we built the hardware platform of a computer system named *Hack*, capable of running programs written in the Hack machine language. In Part II of the book we will transform this barebone machine into an advanced technology, indistinguishable from magic: a black box that can metamorphose into a chess player, a search engine, a flight simulator, a media streamer, or anything else that tickles your fancy. In order to do so, we'll unfold an elaborate behind-the-scene software hierarchy that will endow the Hack platform with the ability to execute programs written in high-level programming languages. In particular, we'll focus on *Jack*, a simple, Java-like, object-based programming language, described formally in Chapter 9. Over the years, Nand to Tetris readers and students have used Jack to develop *Tetris*, *Pong*, *Snake*, *Space Invaders,* and numerous other games and interactive apps. Being a general-purpose computer, Hack can execute all these programs, as well as any other program that comes to your mind.

Clearly, the gap between the expressive syntax of high-level programming languages, on the one hand, and the clunky instructions of low-level machine language, on the other, is huge. If you are not convinced, try developing a Tetris game using instructions like `@17` and `M=M+1`. Bridging this gap is precisely what Part II of the book is all about. We will build this bridge by developing gradually some of the most powerful and ambitious programs in applied computer science: a *compiler*, a *virtual machine*, and a basic *operating system*.

Our Jack compiler will be designed to take a Jack program, say Tetris, and produce from it a stream of machine language instructions that, when executed, makes the Hack platform deliver a Tetris game experience. Of course Tetris is just one example: the compiler that you will build will be capable of translating *any* given Jack program into machine code that effects the program's semantics on the Hack computer. The compiler, whose main tasks consist of *syntax analysis* and *code generation*, will be built in chapters 10 and 11.

As with programming languages like Java and C#, the Jack compiler will be *two-tiered*: the compiler will generate interim *VM code*, designed to run on an abstract *virtual machine*. The VM code will then be compiled further by a separate translator into the Hack machine language. *Virtualization* – one of the most important ideas in applied computer science – comes to play in numerous settings including program compilation, cloud

computing, distributed storage, distributed processing, and operating systems. We will devote chapters 7 and 8 for motivating, designing, and building our virtual machine.

Like many other high-level languages, the basic Jack language is surprisingly simple. What turns modern languages into powerful programming systems are *extensions* like mathematical functions, string processing, memory management, graphics drawing, user interaction handling, and more. Taken together, these services form a basic *operating system* (OS) which, in the Jack framework, is packaged as Jack's *standard class library*. This basic OS, designed to bridge many gaps between the high-level Jack language and the low-level Hack platform, will be developed in Jack itself. You may be wondering how software that is supposed to enable a programming language can be developed in this very same language. We'll deal with this challenge by following a development strategy known as *bootstrapping*, similar to how the Unix OS was developed using the C language.

The construction of the OS will give us an opportunity to present elegant algorithms and classical data structures that are typically used to manage hardware resources and peripheral devices. We will then implement these algorithms in Jack, extending the language's capabilities one step at a time. As you go through the chapters of Part II, you will deal with the OS from several different perspectives. In Chapter 9, acting as an *application programmer*, you will develop a Jack app and use the OS services abstractly, from a high-level client perspective. In Chapters 10 and 11, when building the Jack compiler, you will use the OS services as a low-level client, e.g. for various memory management services required by the compiler. In Chapter 12 you will finally don the hat of the OS developer, and implement all these system services, yourself.

II.1 A Taste of Jack Programming

Before delving into all these exciting projects, we'll give a brief and informal introduction of the Jack language. This will be done using two examples, starting with *Hello World*. We will use this example to demonstrate that even the most trivial high-level program has much more to it than meets the eye. We will then discuss a simple program named *PointDemo*. This program will be used to demonstrate the multi-class and object-based capabilities of the Jack language. Once we get a programmer-oriented taste of the high-level Jack language, we will be prepared to start the journey of realizing the language by building a virtual machine, a compiler, and an operating system.

Hello World, Again: We began this book with the iconic *Hello World* program that students often encounter as the first thing in introductory programming courses. Here is this trivial program once again, written in the Jack programming language:

```
// First example in Programming 101:
class Main {
    function void main() {
        do Output.printString("Hello World");
        return;
    }
}
```

Let's discuss some of the implicit assumptions that we normally make when presented with such programs. The first magic that we take for granted is that a bunch of plain characters, say `printString("Hello World")`, can cause the computer to actually display something on the screen. How does the computer figure out *what* to do? And even if the computer knew what to do, *how* will it actually do it? As we saw in Part I of the book, the screen is just a grid of pixels. If we want to display the letter "H" on the screen, we have to turn on and off a carefully selected subset of pixels that, taken together, render the desired letter image on the screen. Of course this is just the beginning. What about displaying this `"H"` legibly on screens that have different sizes and resolutions? And what about dealing with *while* and *for* loops, *arrays*, *objects*, *methods*, *classes*, and all the other goodies that high-level programmers are trained to use without ever thinking about how they work?

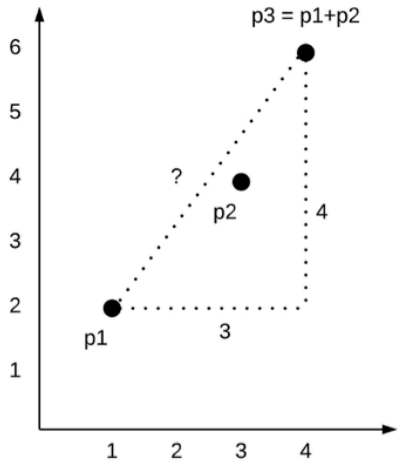
Indeed, the beauty of high-level programming languages, and that of well-designed abstractions in general, is that they permit using them in a state of blissful ignorance. Application programmers are in fact encouraged to view the language as a black-box abstraction, without paying any attention to how it is actually implemented. All you need is a good tutorial, a few code examples, and off you go.

Clearly though, at some point or another, *someone* must actually implement this language abstraction. Someone must develop, once and for all, the ability to efficiently compute square roots when the application programmer blissfully says `sqrt(173056)`, to elicit a number from the user when the programmer happily says `x = readInt()`, to find and carve out an available memory block when the programmer nonchalantly creates an object using `new`, and to perform transparently all the other abstract services that programmers expect to get without ever thinking about them. So, who are the good souls who turn high-level programming into an advanced technology indistinguishable from magic? They are the

software wizards who develop *compilers*, *virtual machines*, and *operating systems*. And that's precisely what *you* will do in Part II of the book.

You may be wondering why you have to bother about this elusive behind-the-scene scene. Didn't we just say that you can use high-level languages without worrying about how they work? There are at least two reasons why. First, the more you delve into low-level system internals, the more sophisticated high-level programmer you become. In particular, you learn how to write high-level code that exploits the hardware and the OS cleverly and efficiently, and how to prevent bugs that baffle many programmers, like memory leaks. Second, by getting your hands dirty and developing the system internals yourself, you will discover some of the most beautiful and powerful algorithms and data structures in applied computer science. Importantly, the ideas and techniques that will unfold in Part II of the book are not limited to compilers and operating systems. Rather, they are the building blocks of numerous software systems and applications that will accompany you throughout your career.

The PointDemo program: Suppose we want to represent and manipulate *points* in a two-dimensional space. Figure II.1 shows two such points, p_1 and p_2, and a third point, p_3, resulting from the vector addition $p_3 = p_1 + p_2 = (1,2) + (3,4) = (4,6)$. The figure also depicts the *Euclidean distance* between p_1 and p_3, which can be computed using the Pythagorean theorem. The code in the `Main` class illustrates how such algebraic manipulations can be done using the object-based Jack language.




```
/** Demo: working with Point objects.
 File name: Main.jack. */
class Main {
  function main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     do Output.printInt(p1.distance(p2));
     do Output.printInt(getPointCount());
     // Should print (4,6), 5, and 3.
     return;
  }
}
```

Figure II-1: Manipulating points in a two-dimensional space: example and Jack code

Some readers may wonder why Jack uses idioms like `var`, `let`, and `do`. At this point, we advise not to dwell on syntactic details. Instead, let's focus on the big picture, and proceed to

review how the Jack language can be used to implement the `Point` abstract data type (figure II-2).

```
/** Represents a two-dimensional point.
File name: Point.jack. */
class Point {
    // The coordinates of this point:
    field int x, y
    // The number of Point objects constructed so far:
    static int pointCount;

    /** Constructs a two-dimensional point and
     * initializes it with the given coordinates. */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }

    /** Returns the x coordinate of this point. */
    method int getx() { return x; }

    /** Returns the y coordinate of this point. */
    method int gety() { return y; }

    /** Returns the number of Point objects
     * constructed so far. */
    function int getPointCount() {
        return pointCount;
    }

    // Class declaration continues on the right.
```

```
    /** Returns a point which is this point plus
     the other point. */
    method Point plus(Point other) {
        return Point.new(x + other.getx(),
                         y + other.gety());
    }

    /** Returns the Euclidean distance between
     *  this and the other point. */
    method int distance(Point other) {
        var int dx, dy;
        let dx = x - other.getx();
        let dy = y - other.gety();
        return Math.sqrt((dx*dx) + (dy*dy));
    }

    /** Prints this point, as "(x,y)" */
    method void print() {
        do Output.printString("(");
        do Output.printInt(x);
        do Output.printString(",");
        do Output.printInt(y);
        do Output.printString(")");
        return;
    }

} // End of class Point.
```

Figure II-2: Jack implementation of the `Point` abstraction.

The code shown in Figure II-2 illustrates that a Jack class (of which `Main` and `Point` are two examples) is a collection of one or more *subroutines*, each being a *constructor*, *method*, or *function*. C*onstructors* are subroutines that create new objects, *methods* are subroutines that operate on the current object, and *functions* are static methods that operate on no particular object. (Object-oriented design purists may frown about mixing methods and functions in the same class; we are doing it here for illustrative purposes).

The remainder of this section is an informal overview of the `Main` and the `Point` classes. Our goal is to give a taste of Jack programming, deferring a complete language description to chapter 9. So, allowing ourselves the luxury of focusing on essence only, let's get started. The `Main.main` function begins by declaring three *object variables* (also known as *references*, or *pointers*), designed to refer to instances of the `Point` class. It then goes on to construct two `Point` objects, and refer the `p1` and `p2` variables to them. Next, it calls the `plus`

method, and refers `p3` to the `Point` object returned by that method. The rest of the `Main.main` function is self-explanatory, barring syntactic details.

The `Point` class begins by declaring that every `Point` object is characterized by two *field* variables (also known as *properties*, or *instance variables*). It then declares a *static variable*, i.e. a class-level variable associated with no particular object. The class constructor sets up the field values of the newly created object, and increments the number of instances derived from this class so far. Note that a Jack constructor must explicitly return the memory address of the newly created object, which, according to the language rules, is denoted `this`.

You may wonder why the result of the square root computed by the `distance` method is stored in an `int` variable – clearly a real-valued data type like `float` will make more sense. The reason for this peculiarity is simple: the basic Jack language features only three primitive data types: `int`, `boolean`, and `char`. Other data types can be implemented at will using class extensions, as we'll do in chapters 9 and 12.

The Operating System: The `Main` and `Point` classes make use of three OS functions: `Output.printInt`, `Output.printString` and `Math.sqrt`. Just like many other modern high-level languages, the basic Jack language is augmented by a *standard class library*, also referred to as OS, that provides all manner of frequently needed services and data types (the complete OS API is given in appendix 6). We will have much more to say about the OS services in chapter 9, where we'll use them abstractly in the context of Jack programming, as well as in Chapter 12, where we'll build the OS.

In addition to calling OS services for their effects directly from Jack programs, the OS comes to play in many other, less obvious ways. For example, consider the `new` operation, used to construct objects in object-oriented languages. How does the compiler know where in the host RAM to put the newly constructed object? Well, it doesn't. An OS routine is called to figure it out. When we'll build the OS in Chapter 12, you will implement, among many other things, a typical run-time memory management system. You will then learn, hands-on, how this system interacts with the hardware, from the one end, and with compilers, from the other, in order to allocate and reclaim RAM space cleverly and efficiently. This is just one example that illustrates how the OS bridges gaps between high-level applications and the host hardware platform.

II.2 Program Compilation

A high-level program is a symbolic abstraction that means nothing to the underlying hardware. Before executing a program, the high-level code must be translated into machine language. This translation process is called *compilation*, and the program that carries it out is called *compiler*. Writing a compiler that translates high-level programs into low-level machine instructions is a worthy challenge. Some modern languages, e.g. Java and C#, deal with this challenge by employing an elegant *two-tier* compilation model. First, the source program is translated into an interim, abstract VM code (called "bytecode" in Java and Python, and "IL" in C#/.NET). Next, using a completely separate and independent process, the VM code can be translated further into the machine language of any host hardware platform.

This attractive modularity is at least one reason why Java became such a dominant programming language. Taking a historical perspective, Java can be viewed as a powerful object-oriented language whose two-tier compilation model was the right thing in the right time, just when computers began evolving from a few predictable processor/OS platforms into a bewildering hodgepodge of numerous PC's, cellphones, mobile devices, and Internet of Things devices, all connected by a global network. Writing high-level programs that can execute on any one of these host platforms is a daunting challenge. One way to streamline this distributed, multi-vendor ecosystem (from a compilation perspective) is to base it on some overarching, agreed-upon Virtual Machine architecture. Acting as a common, intermediate runtime environment, the VM approach allows developers to write high-level programs that run almost as-is on many different hardware platforms, each equipped with its own VM implementation.

At this point you are probably wondering how a virtual machine looks like, and how it is implemented across different hardware platforms. We'll devote chapters 7 and 8 to the VM model, language, and implementation. Virtual machines play a crucial role in the history and practice of computer science, dating back to Alan Turing in the 1930's. Therefore, it's important to get to know them rigorously, as we are about to do.

The Road Ahead: In the remainder of the book we'll apply ourselves to developing all the exciting software technologies mentioned above. Our ultimate goal is creating a mechanism for turning high-level programs – *any* program – into executable code. The roadmap is shown in figure II-3.

Following the Nand to Tetris spirit, we'll pursue the Part II roadmap bottom up. To get started, we assume that we have a hardware platform, equipped with an assembly language. In chapters 7-8 we'll present a virtual machine architecture and a VM language, and implement this abstraction by developing a *VM Translator* that translates VM programs into Hack assembly programs. In chapter 9 we'll present the Jack high-level language, and use it to develop a simple computer game. This way, you'll get acquainted with the Jack language and operating system before setting out to build them. In chapters 10-11 we'll develop the Jack compiler, and in chapter 12 we'll build the operating system.

So, let's roll up our sleeves and go to work!

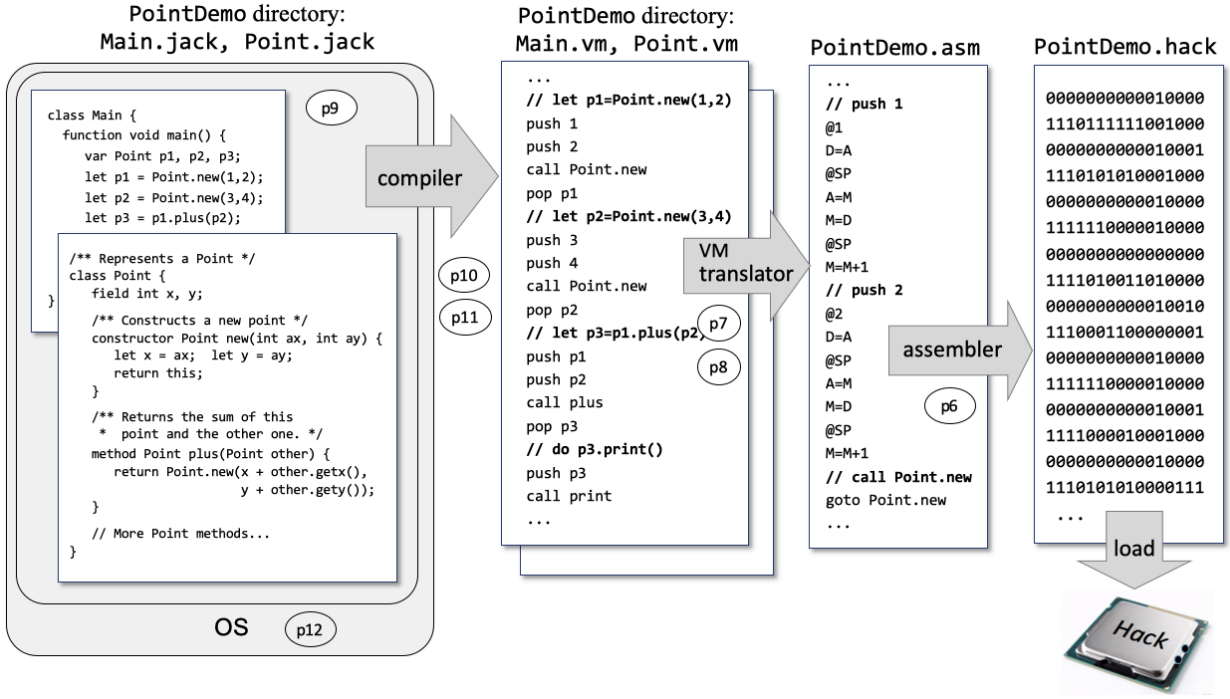


Figure II-3: Roadmap of Part II of the book (the assembler belongs to part I, and is shown here for completeness). The roadmap describes a translation hierarchy, from high-level, object-based, multi-class program to VM code, to assembly code, to executable binary code. The numbered circles stand for the projects that implement the compiler, the VM translator, the assembler, and the operating system. Project 9 focuses on writing a Jack application, in order to get acquainted with the language.