Answers

EXERCISE 1.1

1. Programs
2. Memory
3. Machine language, Assembly language, High-level language
4. Compliers
5. Android
6. Commercial
7. Accelerometer

EXERCISE 1.2

1. Java
2. The javac
3. .java
4. .class
5. Bytecode

EXERCISE 1.3

1. Information Hiding
2. Classes
3. Object Oriented Analysis and Design (OOAD)
4. Inheritance
5. UML
6. Attributes

EXERCISE 1.4

1. Input Unit
2. Programming
3. Assembly Language
4. Output Unit
5. Memory and Storage
6. Arithmetic Logic Unit (ALU)
7. Arithmetic Logic Unit (ALU)
8. High Level
9. Machine Language
10. Control Unit

EXERCISE 1.5

1. Java
2. C
3. Transmission Control Protocols
4. C++

EXERCISE 1.6

1. Edit, Compile, Load, Verify, and Execute
2. Integrated Development Environment
3. Java Virtual Machine
4. Java Virtual Machine
5. Class Loader
6. Bytecode Verifier

EXERCISE 1.7

Explain the two compilation phases of Java programs.  
  
Java programs go through two distinct compilation phases before they can be executed. These phases are part of the process that allows Java to be platform-independent, ensuring that the same code can run on different machines. Here's the explanation:  
  
1. Source Code to Bytecode Compilation (First Phase):  
- In this phase, the Java source code, written in a .java file (e.g., MyProgram.java), is compiled into an intermediate form called bytecode.  
- The Java compiler, known as javac, performs this compilation. It translates the high-level Java code into a platform-independent set of instructions stored in a .class file (e.g., MyProgram.class).  
- Bytecode is not machine code; it’s a set of instructions that the Java Virtual Machine (JVM) can understand. This phase ensures that the code is syntactically correct and follows Java’s rules. If there are errors (like syntax mistakes), the compiler will report them, and no .class file will be generated until the errors are fixed.  
- This phase happens on the developer’s machine, and the resulting bytecode is portable—it can be transferred to any machine with a JVM.  
  
2. Bytecode to Machine Code Compilation (Second Phase):  
- In this phase, the bytecode in the .class file is translated into machine code that the specific hardware and operating system can execute.  
- This happens at runtime, when the java command is used to run the program (e.g., java MyProgram). The JVM interprets or compiles the bytecode into native machine code using one of two methods:  
- Interpretation: The JVM interprets the bytecode line by line and executes it directly. This is slower but allows for immediate execution.  
- Just-In-Time (JIT) Compilation: The JVM uses a JIT compiler to translate the bytecode into machine code all at once (or in chunks) before execution. This is faster and is the default approach in modern JVMs.  
- The JVM is platform-specific, meaning there’s a different JVM for Windows, macOS, Linux, etc. The JVM handles the translation of bytecode to machine code tailored for the specific platform, which is why Java is "write once, run anywhere."  
  
In summary, the first phase (source to bytecode) happens during development and produces portable bytecode, while the second phase (bytecode to machine code) happens at runtime on the target machine, enabling platform independence.

EXERCISE 1.8

 One of the world’s most common objects is a wrist watch. Discuss how each of the following terms and concepts applies to the notion of a watch: object, attributes, behaviors, class, inheritance (consider, for example, an alarm clock), modeling, messages, encapsulation, interface, and information hiding.  
  
This exercise asks us to apply object-oriented programming (OOP) concepts to a wrist watch. Let’s break it down term by term and relate each to a watch.  
  
1. Object:  
- An object is an instance of a class, representing a specific entity in the real world. A wrist watch on your hand is an object—it’s a tangible thing with specific characteristics and functions.  
- Example: My specific wrist watch, a "Casio G-Shock," is an object. It has its own unique serial number, color, and settings.  
  
2. Attributes:  
- Attributes are the properties or data that describe an object. For a wrist watch, these are the characteristics that define its state.  
- Example: A watch’s attributes might include its time (hours, minutes, seconds), date, color, brand, battery level, and size. For my Casio G-Shock, the time might be 3:45 PM, the date might be March 22, 2025, and the color might be black.  
  
3. Behaviors:  
- Behaviors are the actions or operations an object can perform, often represented as methods in a class. For a wrist watch, these are the things it can do.  
- Example: A watch’s behaviors include displaying the time, setting the time, setting the date, starting/stopping a stopwatch, and beeping for an alarm. My Casio G-Shock can display the time when I press a button or start a stopwatch for my morning run.  
  
4. Class:  
- A class is a blueprint for creating objects. It defines the attributes and behaviors that objects of that type will have.  
- Example: A Watch class would define the general structure for all watches. It might include attributes like time, date, and color, and behaviors like setTime() and displayTime(). My Casio G-Shock is an instance (object) of the Watch class.  
  
5. Inheritance (consider, for example, an alarm clock):  
- Inheritance allows a class to inherit attributes and behaviors from another class, creating a hierarchy. A more specialized class (subclass) can extend a more general class (superclass).  
- Example: An AlarmClock class could inherit from the Watch class. The Watch class has basic attributes like time and behaviors like displayTime(). The AlarmClock class inherits these but adds new attributes like alarmTime and behaviors like setAlarm() and soundAlarm(). So, an alarm clock is a specialized type of watch with extra features.  
  
6. Modeling:  
- Modeling is the process of representing a real-world entity (like a watch) as a class in a program, capturing its essential attributes and behaviors.  
- Example: To model a watch in a program, we’d create a Watch class with attributes like time and date, and methods like setTime() and displayTime(). This model simplifies the real watch but captures what’s necessary for the program’s purpose, like tracking time.  
  
7. Messages:  
- In OOP, objects communicate by sending messages, which are essentially method calls. A message tells an object to perform one of its behaviors.  
- Example: When I press a button on my watch to display the time, I’m sending a "message" to the watch object to execute its displayTime() method. In a program, this might look like myWatch.displayTime().  
  
8. Encapsulation:  
- Encapsulation is the bundling of data (attributes) and methods (behaviors) into a single unit (a class), while restricting direct access to some of the object’s components.  
- Example: In a Watch class, the time attribute might be private, meaning it can’t be accessed directly. Instead, public methods like getTime() and setTime() provide controlled access. This ensures that the time can only be modified in a valid way (e.g., setting hours between 0 and 23).  
  
9. Interface:  
- An interface defines a contract of behaviors (methods) that a class must implement, without specifying how they’re implemented. It’s the way an object exposes its functionality to the outside world.  
- Example: A watch might have an interface called TimeKeeper that requires methods like getTime() and setTime(). Any class (like Watch or AlarmClock) that implements TimeKeeper must provide these methods. The interface is like the buttons on a watch—it’s how I interact with it without knowing the internal details.  
  
10. Information Hiding:  
- Information hiding is a principle of encapsulation where the internal details of an object are hidden from the outside world, and only a public interface is exposed.  
- Example: In a watch, I don’t need to know how the internal gears or circuits work to set the time—I just press buttons (the public interface). In the Watch class, the time attribute might be private, and the internal logic for updating the time (e.g., ticking every second) is hidden. I can only interact with the time through methods like setTime().