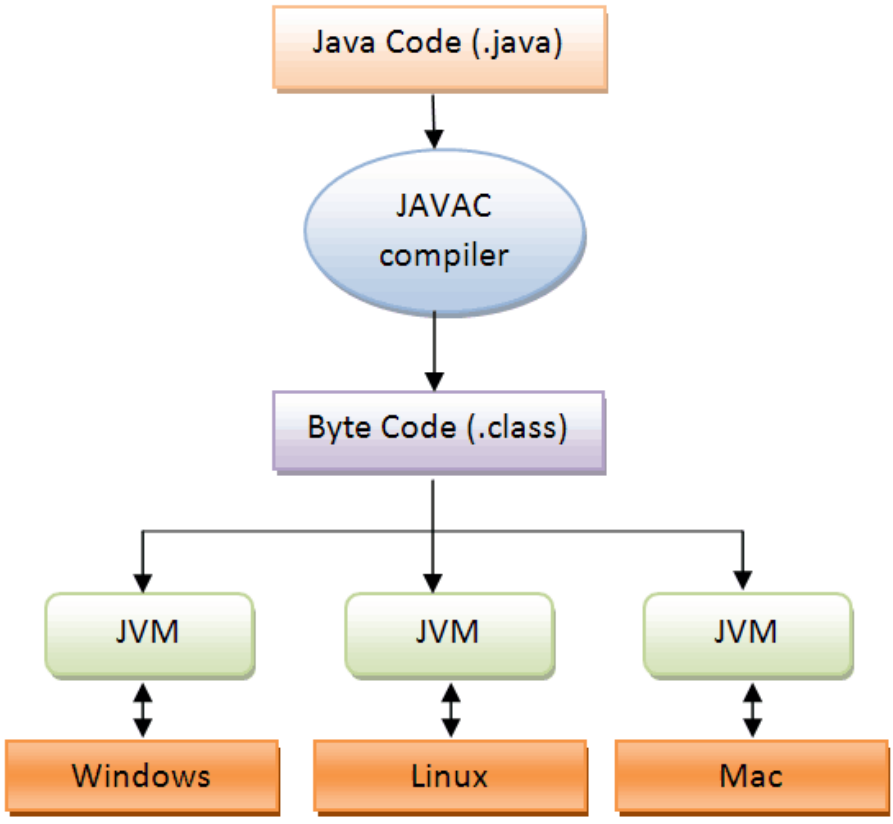
Introduction to Java

Java is a set of computer software and specifications developed by James Gosling at Sun Microsystems, which was later acquired by the Oracle Corporation, that provides a system for developing application software and deploying it in a cross-platform computing environment. Java is used in a wide variety of computing platforms from embedded devices and mobile phones to enterprise servers and supercomputers. Java's dominant place in today's industry was bolstered by Google's choice of the language as it's platform for the Android phone.

Writing in the Java programming language is the primary way to produce code that will be deployed as byte code in a Java virtual machine (JVM)

A Java virtual machine (JVM) is an abstract (virtual) computer that enables a computer to run Java programs, as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.

Write Once:
Write once, run anywhere (WORA), or sometimes Write once, run everywhere (WORE), was a slogan created by Sun Microsystems to illustrate the cross-platform benefits of the Java language. Ideally, this meant that a Java program could be developed on any device, compiled into a standard bytecode, and be expected to run on any device equipped with a Java virtual machine (JVM). The installation of a JVM or Java interpreter on chips, devices, or software packages became an industry standard practice.



The catch is that since there are multiple JVM implementations, on top of a wide variety of different operating systems, there could be subtle differences in how a program executes on each JVM/OS combination, possibly requiring an application to be tested on each target platform. This gave rise to a joke among Java developers: Write Once, Debug Everywhere.

A programmer could develop code on a PC and expect it to run on Java-enabled mobile phones, as well as on routers and mainframes equipped with Java, without any adjustments. This was intended to save software developers the effort of writing a different version of their software for each platform or operating system they intend to deploy on.

Byte code compilers are also available for other languages, including Ada, JavaScript, Python, and Ruby. In addition, several languages have been designed to run natively on the JVM, including Clojure, Groovy, and Scala. Java syntax borrows heavily from C and C++, but object-oriented features are modeled after Smalltalk and Objective-C. Java eschews certain low-level constructs such as pointers and has a very simple memory model where objects are allocated on the heap and all variables of object types are references. Memory management is handled through integrated automatic garbage collection performed by the JVM.

Java is a high-level language. Other high-level languages you may have heard of include Python, C and C++, Ruby, and JavaScript. Before they can run, programs in high-level languages have to be translated into a low-level language, called “machine language”. This translation takes some time, which is a small disadvantage of high-level languages. But high-level languages have two advantages:

1. It is much easier to program in a high-level language. Programs take less time to write, they are shorter and easier to read, and they are more likely to be correct.

2. High-level languages are portable, meaning they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Two kinds of programs translate high-level languages into low-level languages: **interpreters** and **compilers.** An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. The program is re-interpreted each time it is executed. This factors in to the slower speed of interpreted languages, but also aids in the portability. Figure 1.1 shows the structure of an interpreter. Because of this re-interpretation interpreted programs are traditionally slower than compiled programs. With today’s modern hardware this is becoming less of an issue.

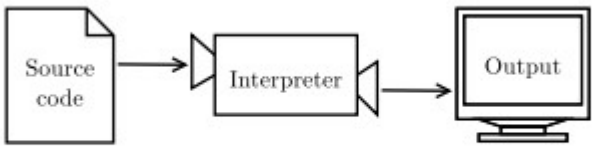


Figure 1.1: How interpreted languages are executed.

In contrast, a compiler reads the entire program and translates it completely before the program starts running. In this context, the high-level program. is called the **source code,** and the translated program is called the **object code, byte code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. As a result, compiled programs often run faster than interpreted programs. Java is both compiled and interpreted. Instead of translating programs directly into machine language, the Java compiler generates **byte code**. Similar to machine language, byte code is easy and

fast to interpret. But it is also portable, so it is possible to compile a Java program on one machine, transfer the byte code to another machine, and run the byte code on the other machine. The interpreter that runs byte code is called a “Java Virtual Machine” (JVM). Javas portability comes from the existence of many different JVM implementations; one for each major operating system. Figure 1.2 shows the steps of this process. Although it might seem complicated, these steps are automated for you in most program development environments. Usually you only have to press a button or type a single command to compile and run your program. On the other hand, it is important to know what steps are happening in the background, so if something goes wrong you can figure out what it is.

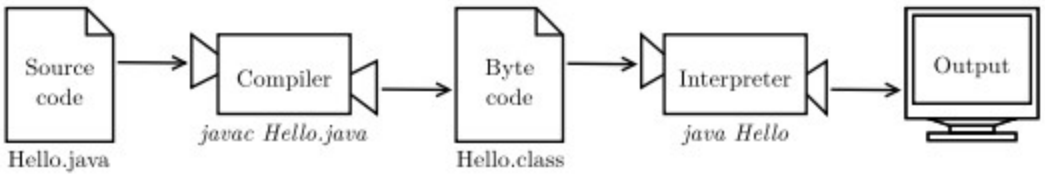


Figure 1.2: The process of compiling and running a Java program.

The tools that you see above come with the language download and are described below

1. **javac:** The Java compiler consumes Java source code and emits Java byte-code
2. **java:** The Java interpreter consumes Java byte-code and emits machine code appropriate for whatever OS the machine is installed on.

In the lab for this module you will download, install and configure Java for your operating system.

Java is Strictly Typed

Unlike Python, which is dynamically typed, Java requires explicit data typing when defining a variable. The Java compiler needs to know whether a variable will be an integer, floating point, String or some type of custom object. If the types are not expressed when creating a variable the code will not compile (more on this later)

- **Java code to define an integer variable:** int x = 12;
- **Python code to define an integer variable:** x = 12

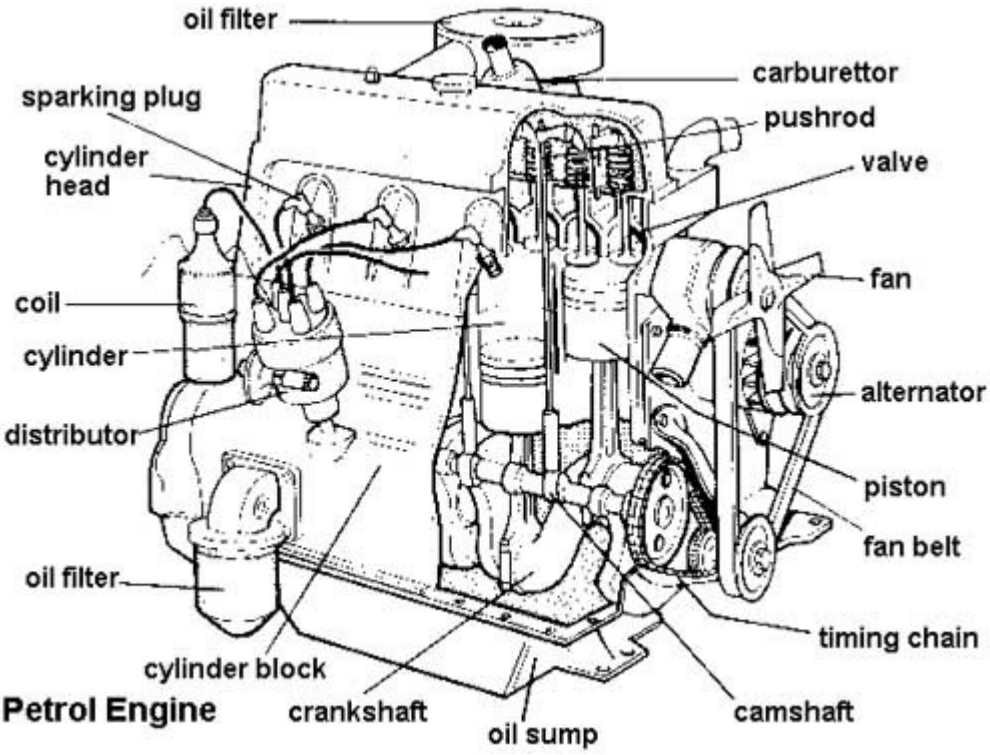
Java is Object Oriented

Object-oriented programming (OOP) is a programming paradigm based on the concept of interacting "objects".

So what is an object? Just as in the real world, an object is any thing whatsoever. An object can be a physical thing, such as a Car, or a mental thing, such as an Idea. It can be a natural thing, such as an Animal, or an artificial, human-made thing, such as an ATM. A program that manages an ATM would involve BankAccounts and Customer objects. A chess program would involve a Board object and ChessPiece objects. A card game would involve Card objects, Deck objects and Hand objects. A

Dungeons and Dragons program would involve Characters, Items, Dungeons (subdivided into rooms) and Maps. An object can literally be anything that your imagination can envision.

Interacting Objects: An OO application is one that contains many objects. These objects interact with each other. Taking the example of a complex machine like a Car we can see how these objects interact. A Person object enters the vehicle through the Door object. The Door is controlled by a Handle object which changes the state of the Door from "opened" to "closed". The Person starts the Car by placing a key in the Ignition object. Turning the key causes the Ignition object to send signals to the Starter object which then starts the Engine object. The Person then uses a lever to send signals to the Transmission object. These signals will tell the Transmission which state to change to: Drive, Reverse or Neutral. The Person would then communicate with the Engine by pressing a Pedal object which sends signals to the Engine to increase or decrease speed. The Person communicates with the Brake objects by pressing a different pedal which tells the Brakes how much force to apply. This analogy can be extended to excruciating detail, all the way down to the bolt objects that are used to hold things together. How many distinct objects do you think comprise a single engine? Each one of these objects works in concert with other objects to create a combustion engine. Each one of these objects is designed as a self-contained unit that can receive messages, telling it what to do. With this process, a single fan could be used in many different types of engines, or not in engines at all. Objects are self-contained re-usable components.



This object interaction is the fundamental concept of object oriented programming. Each one of these objects would be defined as a code module called a **class**. The class would define the characteristics and behaviors of each object, as well as the way in which other objects could interact. This method of interaction is called the object's **interface.** For example, we increase the speed of the engine by pressing a pedal. This pedal is the objects interface. We increase the volume of the radio by turning a

knob or pressing a button. These controls (interfaces) allow humans to interact with and control these systems.

Benefits of Object Oriented Programming

Modularity for easier troubleshooting
Something has gone wrong, and you have no idea where to look. Is the problem in the Widget file, or is it the WhaleFlumper? Will you have to trudge through that "sewage.java" file? Hope you commented your code! When working with object-oriented programming languages, you know exactly where to look. "Oh, the fuel pump in your car has stopped working? The problem must be in the FuelPump class!" You don't have to muck through anything else. That's the beauty of encapsulation. Objects are self-contained, and each bit of functionality does its own thing while leaving the other bits alone. Also, this modality allows an team to work on multiple objects simultaneously while minimizing the chance that one person might duplicate functionality.

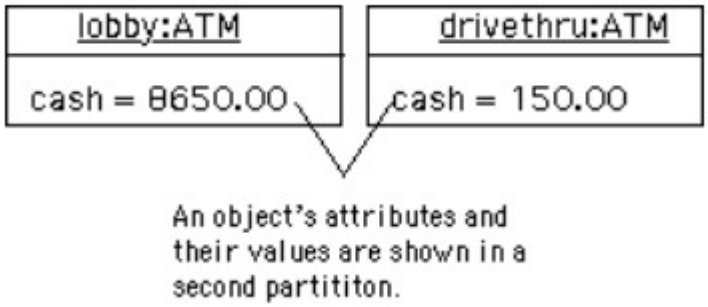
Reuse of code through inheritance
Suppose that in addition to your Car object, one colleague needs a RaceCar object, and another needs a Limousine object. Everyone builds their objects separately but discover commonalities between them. In fact, each object is really just a different kind of Car. This is where the inheritance technique saves time: Create one generic class (Car), and then define the subclasses (RaceCar and Limousine) that are to inherit the generic class's traits. Of course, Limousine and RaceCar still have their unique attributes and functions. If the RaceCar object needs a method to "fireAfterBurners" and the Limousine object requires a Chauffeur, each class could implement separate functions just for itself. However, because both classes inherit key aspects from the Car class, for example the "drive" or "fillUpGas" methods, your inheriting classes can simply reuse existing code instead of writing these functions all over again. What if you want to make a change to all Car objects, regardless of type? This is another advantage of the OO approach. Simply make a change to your Car class, and all car objects will simply inherit the new code.

Flexibility through polymorphism
Riffing on this example, you now need just a few drivers, or functions, like "driveCar," driveRaceCar" and "DriveLimousine." RaceCarDrivers share some traits with LimousineDrivers, but other things, like RaceHelmets and BeverageSponsorships, are unique. This is where object-oriented programming's sweet polymorphism comes into play. Because a single function can shape-shift to adapt to whichever class it's in, you could create one function in the parent Car class called "drive" — not "driveCar" or "driveRaceCar," but just "drive." This one function would work with the RaceCarDriver, LimousineDriver, etc. In fact, you could even have "raceCar.drive(myRaceCarDriver)" or "limo.drive(myChauffeur)."

Natural Problem Solving
Object-oriented programming is often the most natural and pragmatic approach, once you get the hang of it. OOP languages allows you to break down your software into bite-sized problems that you then can solve — one object at a time. This isn't to say that OOP is the One True Way. However, the advantages of object-oriented programming are many. When you need to solve complex programming challenges and want to add code tools to your skill set, OOP is your friend — and has much greater longevity and utility than Pac-Man or parachute pants.

Objects are reusable: An important aspect of OOP is that these objects are reusable. We can define a single blueprint for a Starter and then implement that Starter in infinite Cars. A Pedal object from a single design can be mass produced and used in any type of vehicle. Many different Cars use the same Engine design . . . etc. Think about these complex systems. You will be asked to describe a system in these terms. Also think about the way us humans interact with these complex systems.

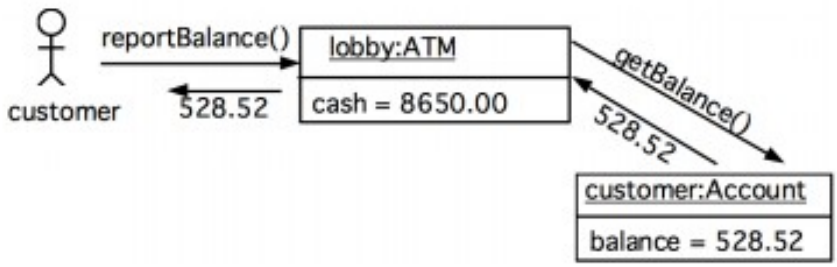
Attributes and Values: Just as with real objects, the objects in our programs have certain characteristic attributes. For example, an ATM object would have a current amount of cash that it could dispense. A ChessPiece object might have a pair of row and column attributes that specify its position on the chess board. Notice that an object's attributes are themselves objects. A Dungeons and Dragons character would have an attribute to store it's health, as well as attributes to store the items that a character possesses. These attributes are implemented with variables of different data types. The figure below shows two ATM objects and their respective attributes. As you can see, an object's attributes are listed in a second partition of the UML diagram. Notice that each attribute has a value. So the lobby:ATM has a $8650.0 in cash, while the drivet hru:ATM has only $150.0 in cash.



To continue with the Car analogy, a Car would have the attributes: make, model, year, color , fuel_level . . . etc.

Actions and Messages: In addition to their attributes, objects also have characteristic actions or behaviors. As we have already said, objects in programs are dynamic. They do things or have things done to them. In fact, programming in Java is largely a matter of getting objects to perform certain actions for us. For example, in a chess program the ChessPieces have the ability to ***moveTo()*** a new position on the chess board. Similarly, when a customer pushes the "Current Balance" button on an ATM machine, this is telling the ATM to ***report()*** the customer's current bank balance. (Note how we use parentheses to distinguish actions from objects and attributes.) The actions that are associated with an object can be used to send messages to the objects and to retrieve information from objects. A message is the passing of information or data from one object to another.

The figure below shows a customer interacting with an ATM. The ATM object interacts with the Account object in order to retrieve the customer's balance.



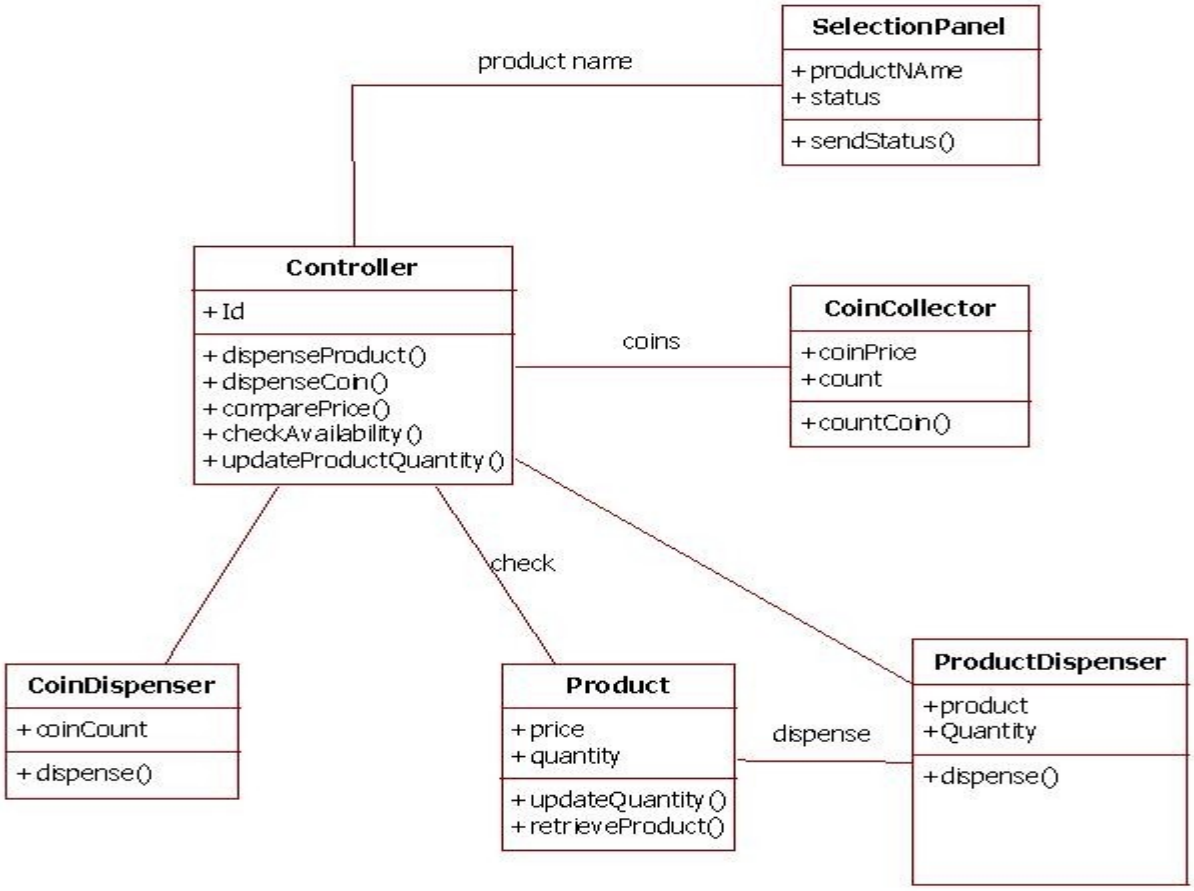
To continue with the Car analogy, a Car would have the behaviors: accelerate( ), decelerate( ), start( ), stop( ) . . . etc.

What is a Class? A class is a template for an object. A class encapsulates the attributes and actions that characterize a certain type of object. In an object-oriented program, classes serve as blueprints or templates for the objects that the program uses. We say that an object is an instance of a class. A good analogy here is to think of a class as a recipe. The recipe defines how a pie is constructed. We can construct infinite pies from a single recipe.

The recipe is the class. The pie is the object. We can say that the pie is an instance of the recipe.

Writing an object-oriented program is largely a matter of designing classes and writing definitions for those classes in Java. Designing a class is a matter of specifying all of the attributes and behaviors that are characteristic of that type of object. For example, suppose we are writing a drawing program. One type of object we would need for our program is a rectangle. A Rectangle object has two fundamental attributes, a length and a width. Given these attributes, we can define characteristic rectangle actions, such as the ability to calculate its area and the ability to draw itself and the ability to move or rotate. Identifying an object’s attributes and actions is the kind of design activity that goes into writing an object oriented program.

Unified Modeling Language (UML): Modeling is the designing of software applications before coding. Modeling is an Essential Part of large software projects, and helpful to medium and even small projects as well. A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make. Surveys show that large software projects have a huge probability of failure - in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than that it will succeed. If you're running one of these projects, you need to do all you can to increase the odds for success, and modeling is the only way to visualize your design and check it against requirements before your crew starts to code. UMLis an industry standard modeling tool that allows programmers to think and design in high levels of abstraction, before the first character is typed. Here is an example of a UML diagram for a Vending Machine. The diagram illustrates the objects, attributes, behaviors and interactions.



Principles of Object-Oriented Design As we have discussed, an object-oriented program is composed of many objects communicating with each other. The process of designing an object-oriented program to solve some problem or other involves several important principles:

Divide-and-Conquer Principle. Generally, the first step in designing a program is to divide the overall problem into a number of objects that will interact with each other to solve the problem. Thus, an object oriented program employs a division of labor much as we do in organizing many of our real-world tasks. This divide-and-conquer approach is an important problem-solving strategy. If we were building a Poker program we identify Card, Deck and Hand objects each with their own unique attributes and behaviors.

Composition Principle. Objects can contain other objects as their attributes. For instance a Deck consists of 52 card objects, a hand may consist of 5 card objects. This is the essence of composition. Composition factors heavily into code reuse. We could plug Card and Deck object into any type of card game . . . it does not have to be poker.

Encapsulation Principle. Once the objects are identified, the next step involves deciding, for each object, what attributes it has and what actions it will take. The goal here is to encapsulate within each object the expertise needed to carry out its role in the program. Each object is a self-contained module with a clear responsibility and the tools (attributes and actions) necessary to carry out its role. Just as a

dentist encapsulates the expertise needed to diagnose and treat a tooth ache, a well-designed object contains the information and methods needed to perform its role.

Interface Principle. In order for objects to work cooperatively and efficiently, we have to clarify exactly how they should interact, or interface, with one another. An object's interface should be designed to limit the way the object can be used by other objects. Think of how the different interfaces presented by a digital and analog watch determine how the watches are used. In a digital watch, time is displayed in discrete units, and buttons are used to set the time in hours, minutes and seconds. In an analog watch, the time is displayed by hands on on a clock face, and time is set, less precisely, by turning a small wheel. If we wanted to have a deck object deal a card we would need to know how to tell the object to do this. The deck object may expose a ***deal()*** method that we could call. This would be part of the Deck objects interface.

Information Hiding Principle. In order to enable objects to work together cooperatively, certain details of their individual design and performance should be hidden from other objects. To use the watch analogy again, in order to use a watch we needn't know how its time keeping mechanism works. That level of detail is hidden from us. Hiding such implementation details protects the watch's mechanism, while not limiting its usefulness. We should not be able to change the value of a cards suit or rank . . . these attributes should be hidden from general consumption.

Generality Principle. To make objects as generally useful as possible, we design them not for a particular task but rather for a particular kind of task. This principle underlies the use of software libraries. As we will see, Java comes with an extensive library of classes that specialize in performing certain kinds of input and output operations. For example, rather than having to write our own method to print a message on the console, we can use a library object to handle our printing tasks.

Extensibility Principle. One of the strengths of the object-oriented approach is the ability to extend an object's behavior to handle new tasks. This also has its analogue in the everyday world. If a company needs sales agents to specialize in hardware orders, it would be more economical to extend the skills of its current sales agents instead of training a novice from scratch. In the same way, in the object-oriented approach, an object whose role is to input data might be specialized to input numeric data.

Abstraction Principle. Abstraction is the ability to focus on the important features of an object when trying to work with large amounts of information. For example, if we are trying to design a floor plan for a kitchen, we can focus on the shapes and relative sizes of the appliances and ignore attributes such as color, style, and manufacturer. The objects we design in our Java programs will be abstractions in this sense because they ignore many of the attributes that characterize the real objects and focus only on those attributes that are essential for solving a particular problem.