**Read the list of international words**

Object, concept, code, message, communicate, technology, characteristics, function, method, animation, control, electronic, action, indicate, diagram, encapsulation, idea, design, system, component, interaction, functionality, parameter.

**Task 1**

*Read the text and analyze it*

**Object-Oriented Programming Concepts**

If you've never used an object-oriented language before, you need to understand the underlying concepts before you begin writing code. You need to understand what an object is, what a class is, how objects and classes are related, and how objects communicate by using messages.

**What Is an Object?**

Objects are keys to understanding *object-oriented* technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, your bicycle.

These real-world objects share two characteristics: They all have state and behavior. For example, dogs have state (name, color and breed, hungry) and behavior (barking, fetching, and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels and number of gears) and behavior (braking, accelerating, slowing down and changing gears).

Software objects are modeled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more *variables*. A variable is an item of data named by an identifier. A software object implements its behavior with *methods*. A method is a function (subroutine) associated with an object.

***Definition:*** An object is a software bundle of variables and related methods.

You can represent real-world objects by using software objects. You might want to represent real-world dogs as software objects in an animation program or a real-world bicycle as software object in the program that controls an electronic exercise bike. You can also use software objects to model abstract concepts. For example, an event is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard.

Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. A software object that modeled your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 10 mph, its pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables are formally known as *instance variables* because they contain the state for a particular bicycle object, and in object-oriented terminology, a particular object is called an instance.

In addition to its variables, the software bicycle would also have methods to brake, change the pedal cadence, and change gears. (The bike would not have a method for changing the speed of the bicycle, as the bike's speed is just a side effect of what gear it's in, how fast the rider is pedaling, whether the brakes are on, and how steep the hill is.) These methods are formally known as *instance methods* because they inspect or change the state of a particular bicycle instance.

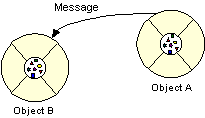
The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called *encapsulation*. This conceptual picture of an object-a nucleus of variables packaged within a protective membrane of methods-is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for. However, it's not the whole story. Often, for practical reasons, an object may wish to expose some of its variables or hide some of its methods. In the Java programming language, an object can specify one of four access levels for each of its variables and methods. The access level determines which other objects and classes can access that variable or method. Variable and method access in Java is covered in Controlling *Access to Members of a Class*. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

* *Modularity:* The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
* *Information hiding:* An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike to use it.

**What Is a Message?**

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your bicycle hanging from a hook in the garage is just a bunch of titanium alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interacts with it (pedal).

Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B



Sometimes, the receiving object needs more information so that it knows exactly what to do; for example, when you want to change gears on your bicycle, you have to indicate which gear you want. This information is passed along with the message as parameters.

These are the three components that comprise a message:

1. The object to which the message is addressed (YourBicycle)
2. The name of the method to perform (changeGears)
3. Any parameters needed by the method (lowerGear)

These three components are enough information for the receiving object to perform the desired method. No other information or context is required.

Messages provide two important benefits.

* An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.

Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

**Task 2**

*Agree or disagree*

Objects are keys to understanding *object-oriented* technology.

These real-world objects share ten characteristics.

Software objects are modeled after imaginary objects.

An object is a software bundle of variables and related methods.

Everything that the software object knows (state) and can do (behavior) is expressed by equations.

Software objects interact and communicate with each other by sending messages to each other.

There are the three components that comprise a message.

Messages provide two important benefits.

**Task 3**

*Answer the questions*

1. What is an object?
2. How can you characterize state and behavior of the object?
3. How is software object modeled?
4. What is a method?
5. How can you represent real-world objects?
6. What is a message?
7. What are the components that comprise a message?
8. What important benefits do messages provide?

**Task 4**

*Make a selective retelling of the text.*

**Task 5**

*Define various meanings of the following words and word combinations*

Current gear, current cadence, two wheels, behavior, change gears, brake, manufacturer, advantage, rectangles, employee records, allocate, invoke, blueprint, inheritance, entity, enforce, tracking number, retail.

**Task 6**

*Spot the international words in the text and guess their meaning*

**Task 7**

*Read the text and analyze it*

**What Is a Class?**

In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an *instance* of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles.

When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. A software blueprint for objects is called a *class*.

**Definition:**A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

The class for our bicycle example would declare the instance variables necessary to contain the current gear, the current cadence, and so on, for each bicycle object. The class would also declare and provide implementations for the instance methods that allow the rider to change gears, brake, and change the pedaling cadence.

After you've created the bicycle class, you can create any number of bicycle objects from the class. When you create an instance of a class, the system allocates enough memory for the object and all its instance variables. Each instance gets its own copy of all the instance variables defined in the class.

In addition to instance variables, classes can define *class variables*. A class variable contains information that is shared by all instances of the class. For example, suppose that all bicycles had the same number of gears. In this case, defining an instance variable to hold the number of gears is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a class variable that contains the number of gears. All instances share this variable. If one object changes the variable, it changes for all other objects of that type. A class can also declare *class methods*. You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance.

*Understanding Instance and Class Members* discusses instance variables and methods and class variables and methods in detail.

**Objects vs. Classes**

You probably noticed that the illustrations of objects and classes look very similar. And indeed, the difference between classes and objects is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects they describe: A blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "object" is sometimes used to refer to both classes and instances.

**What Is Inheritance?**

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes, and tandems are all *subclasses* of the bicycle class. Similarly, the bicycle class is the *superclass* of mountain bikes, racing bikes, and tandems.

Each subclass *inherits* state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: braking and changing pedaling speed, for example.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more special its behavior.

The Object class is at the top of class hierarchy, and each class is its descendant (directly or indirectly). A variable of type Object can hold a reference to any object, such as an instance of a class or an array. Object provides behaviors that are required of all objects running in the Java Virtual Machine. For example, all classes inherit Object's to String method, which returns a string representation of the object.

Inheritance offers the following benefits:

* Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
* Programmers can implement superclasses called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

**What Is an Interface?**

In English, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks. Within the Java programming language, an *interface* is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreed on behavior). In fact, other object-oriented languages have the functionality of interfaces, but they call their interfaces protocols.

The bicycle class and its class hierarchy define what a bicycle can and cannot do in terms of its "bicycleness." But bicycles interact with the world on other terms. For example, a bicycle in a store could be managed by an inventory program. An inventory program doesn't care what class of items it manages as long as each item provides certain information, such as price and tracking number. Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a protocol of communication. This protocol comes in the form of a set of constant and method definitions contained within an interface. The inventory interface would define, but not implement, methods that set and get the retail price, assign a tracking number, and so on.

To work in the inventory program, the bicycle class must agree to this protocol by implementing the interface. When a class implements an interface, the class agrees to implement all the methods defined in the interface. Thus, the bicycle class would provide the implementations for the methods that set and get retail price, assign a tracking number, and so on.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

* Capturing similarities among unrelated classes without artificially forcing a class relationship.
* Declaring methods that one or more classes are expected to implement.
* Revealing an object's programming interface without revealing its class.

**Task 8**

*1 Look through the text and pick out key words for further retelling.*

*2 Pick out 10-15 sentences which convey the basic information.*

*3 Make a selective retelling of the text.*