البرمجة الترجمة الموحدة

عام 1426 - 1427 هـ - الفصل الدراسي الأول
إعداد الطالبة: حمدان عنزي
أَسْمَاحَ المَعَاَجَةَ عَنْ حَمْدِينَةَ
مَعَاَجَةَ (314) مَجَابَ
جمعية (996)
If you've never used an object-oriented language before, you need to understand the underlying concepts before beginning to write code. You need to know what an object is, what a class is, how objects and classes are related, and how objects communicate by using messages. The first few sections of this chapter describe the concepts behind object-oriented programming. The last section shows how these concepts translate into code.

**What Is an Object?**

An object is a software bundle of related variables and methods. Software objects are often used to model real-world objects you find in everyday life.

**What Is a Message?**

Software objects interact and communicate with each other using messages.

**What Is a Class?**

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

**What Is Inheritance?**

A class inherits state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

**What Is an Interface?**

An interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.

**How Do These Concepts Translate into Code?**

This section looks at a small applet, and shows you the code that creates objects, implements classes, sends messages, establishes a superclass, and implements an interface.
What Is an Object?

Objects are key 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.

Real-world objects share two characteristics: They all have state and behavior. For example, dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, 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 a 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 window systems to represent the action of a user pressing a mouse button or a key on the keyboard. The following illustration is a common visual representation of a software object.
Everything the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. A software object that models your real-world bicycle would have variables that indicate the bicycle's current state: Its speed is 18 mph, its pedal cadence is 90 rpm, and its current gear is 5th. These variables are formally known as instance variables because they contain the state for a particular bicycle object; in object-oriented terminology, a particular object is called an instance. The following figure illustrates a bicycle modeled as a software object.

In addition to its variables, the software bicycle would also have methods to brake, change the pedal cadence, and change gears. (It would not have a method for changing its speed because the bike's speed is just a side effect of which gear it's in and how fast the rider is pedaling.) These methods are known formally as instance methods because they inspect or change the state of a particular bicycle instance.
Object diagrams show that an 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 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. Refer to the Controlling Access to Members of a Class section for details.

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 other objects that depend on it. You don't need to understand a bike's gear mechanism to use it.
**What Is a Message?**

A single object alone generally is 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 metal and rubber; by itself, it is incapable of any activity; the bicycle is useful only when another object (you) interacts with it (by pedaling).

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 (see the following figure).

![Message diagram](image)

Objects interact by sending each other messages.

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*.

The next figure shows the three parts of a message:

- **The object to which the message is addressed** (*YourBicycle*)
- **The name of the method to perform** (*changeGears*)
- **Any parameters needed by the method** (*lowerGear*)

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

Messages use parameters to pass along extra information that the object needs — in this case, which gear the bicycle should be in.

**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 messages back and forth and receive messages from each other.
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 that defines the variables and the methods common to all objects of a certain kind.

A class can be visually represented like this:

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, as shown in the next figure. 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 (as shown in the next figure).
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.
The section **Understanding Instance and Class Members** discusses instance variables and methods and class variables and methods in detail.

Objects provide the benefit of modularity and information hiding. Classes provide the benefit of reusability. Bicycle manufacturers reuse the same blueprint over and over again to build lots of bicycles. Software programmers use the same class, and thus the same code, over and over again to create many objects.

**Objects versus 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.

In the figures, the class is not shaded, because it represents a blueprint of an object rather than an object itself. In comparison, an object is shaded, indicating that the object exists and that you can use it.
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, handlebars, 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, road bikes, and tandems are all types of bicycles. In object-oriented terminology, mountain bikes, road bikes, and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, road bikes, and tandems. This relationship is shown in the following figure.

```

Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, road bikes, and tandems share some states: cadence, speed, and the like. Also, each subclass inherits methods from the superclass. Mountain bikes, road bikes, and tandems share some behaviors — braking and changing pedaling speed, for example.
However, subclasses are not limited to the states 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 handlebars; some mountain bikes have an additional chain ring, giving them 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 additional chain ring, you could override the "change gears" method so that the rider could shift into those lower 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 specialized its behavior.

Note: Class hierarchies should reflect what the classes are, not how they're implemented. When implementing a tricycle class, it might be convenient to make it a subclass of the bicycle class — after all, both tricycles and bicycles have a current speed and cadence. However, because a tricycle is not a bicycle, it's unwise to publicly tie the two classes together. It could confuse users, make the tricycle class have methods (for example, to change gears) that it doesn't need, and make updating or improving the tricycle class difficult.

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 shared by all objects running in the Java Virtual Machine. For example, all classes inherit Object's toString method, which returns a string representation of the object. The Managing Inheritance section covers the Object class in detail.

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 common 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 general, 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 individuals of different ranks.

Within the Java programming language, an interface is a type, just as a class is a type. Like a class, an interface defines methods. Unlike a class, an interface never implements methods; instead, classes that implement the interface implement the methods defined by the interface. A class can implement multiple interfaces.

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 communication protocol. This protocol comes in the form of a set of 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
• Modeling multiple inheritance, a feature of some object-oriented languages that allows a class to have more than one superclass
How Do These Concepts Translate into Code?

Now that you have a conceptual understanding of object-oriented programming let's look at how these concepts get translated into code. The following figure shows a snapshot of the graphical user interface (GUI) presented by an example named ClickMeApp, which features a custom GUI component named ClickMe. A spot appears when you click the mouse inside the ClickMe component's area.

The GUI of the ClickMeApp application.

The ClickMeApp example is not large, but if you don't have much experience with programming, you might find the code daunting. We don't expect you to understand everything in this example right away, and this section won't explain every detail. The intent is to expose you to some source code and to associate it with the concepts and terminology you've just learned. You will find more details in later chapters.

Compiling and Running the Example

There are three source files for the ClickMeApp example: ClickMeApp.java, ClickMe.java, and Spot.java. These source files contain the code for three classes named ClickMeApp, ClickMe, and Spot. In addition to these classes, the example uses some classes provided by the Java platform.

You can compile the ClickMeApp.java, ClickMe.java, and Spot.java files by invoking the Java compiler (javac) on ClickMeApp.java.

You can run the ClickMeApp example as an application using either Java Web Start or the java command. To run a precompiled version with Java Web Start, direct your browser to this URL:
If you prefer to run your own copy, you can use the command `java ClickMeApp`.

Note: `ClickMeApp` depends on API introduced in J2SE 5.0. It will not run or compile under earlier versions of the JDK and runtime.

If you're having problems compiling or running this example, see [Common Problems (and Their Solutions)](http://java.sun.com/docs/books/tutorialJWS/java/concepts/ex5/ClickMeApp.jnlp).

Objects in the Example

Many objects play a part in the `ClickMeApp` example. The most obvious ones are those with an onscreen representation: the window containing the GUI, the label describing what the user should do, the custom component that paints the initially empty rectangle, and the spot that's eventually painted by that custom component.

These objects are created when the user launches the application. The application's `main` method creates an object to represent the entire application, and that object creates others to represent the window, label, and custom component.

The object representing the custom component in turn creates an object to represent the spot on the screen. Every time you click the mouse in the custom component, the component moves the spot by changing the spot object's x,y location and repainting itself. The spot object does not display itself; the custom component paints a circle based on information contained within the spot object.

Other, nonvisible objects also play a part in the application. Three objects represent the three colors used in the custom component (black, white, and olive green), an event object represents the user action of clicking the mouse, and so on.

Classes in the Example
As you may have guessed, the object that represents the application is an instance of the `ClickMeApp` class, the object that represents the custom component is an instance of `ClickMe`, and the spot is represented by an instance of `Spot`.

Because the object that represents the spot on the screen is very simple, let's look at its code. The `Spot` class declares three instance variables: `size` contains the spot's radius, `x` contains the spot's current horizontal location, and `y` contains the spot's current vertical location. It also declares two methods and a constructor — a subroutine used to initialize new objects created from the class.

```java
public class Spot {
    // instance variables
    public int x, y;
    private int size;

    // constructor
    public Spot() {
        x = -1;
        y = -1;
        size = 1;
    }

    // methods for access to the size instance variable
    public void setSize(int newSize) {
        if (newSize >= 0) {
            size = newSize;
        }
    }
    public int getSize() {
        return size;
    }
}
```

You can recognize a constructor because it has the same name as the class. The `Spot` constructor initializes all three of the object's variables. The `x` and `y` variables are set to -1, indicating that the spot is not onscreen when the application starts up. The `size` variable is set to a reasonable value, in this case 1.

The `setSize` and `getSize` methods provide a way for other objects to read and change the value of the size instance variable without giving other objects access to the actual variable.

The application's `ClickMe` object uses a `Spot` object to track whether to paint a spot, and where. Here's the `ClickMe` class code that declares and creates the `Spot` object.
The first line shown declares a variable named spot with a data type of Spot; it initializes the variable's value to null. The keyword null is used to specify an undefined value for variables that refer to objects. The next line shown creates the object (see the next figure). The new keyword allocates memory space for the object. Spot() calls the constructor you saw previously.

Messages in the Example

As you know, object A can use a message to request that object B do something, and a message has three components:

- The object to which the message is addressed
- The name of the method to perform
- Any parameters the method needs

Here are two lines of code from the ClickMe class.

```java
g2d.setColor(Color.WHITE);
g2d.fillRect(0, 0, getWidth() - 1, getHeight() - 1);
```

Both are messages from the ClickMe object to an object named g2d — a Graphics2D object that is essential for
painting onscreen. This object is provided to the component when the painting system instructs the component to paint itself. The first line sets the color to white and the second fills a rectangle the size of the component, thus painting the extent of the component’s area white.

The following figure highlights each part of the first message to g2d.

```
g2d.setColor(Color.white);
```

The components of a message sent by the ClickMe object.

Inheritance in the Example

To paint itself onscreen, an object must be a component. This means that the object must be an instance of a class that derives from the Component class provided by the Java platform.

The ClickMe object is an instance of the ClickMe class, which is declared this way:

```java
public class ClickMe extends JComponent implements MouseListener {
    ...
}
```

The `extends JComponent` clause makes ClickMe a subclass of JComponent — a class that derives from the Component class. ClickMe inherits a lot of functionality from Component and JComponent, including the ability to paint itself, to have a standard border around its edges, and to register listeners to be notified of mouse events. Along with these benefits, the ClickMe class has certain obligations: Its painting code must be in a standard place, such as the paintComponent method; it must specify its preferred and minimum sizes; and so on.

```java
public ClickMe() {
    ...
    setPreferredSize(aDimensionObject);
}
```
setMinimumSize(anotherDimensionObject);
...
}

public void paintComponent(Graphics g) {
... //ClickMe's painting code is here.
}

Interfaces in the Example

The **ClickMe** component responds to mouse clicks by displaying a spot at the click location. Any object can detect mouse clicks on a component. It just needs to implement the **MouseListener** interface and register with the component as a mouse listener.

The **MouseListener** interface declares five methods, each of which is called for a different kind of mouse event: when the mouse is clicked, when the mouse moves outside of the component, and so on. Even though the **ClickMe** component responds only to mouse clicks, its mouse listener implementation must have all five methods. The methods for the events that the object isn't interested in are empty.

The complete code for the **ClickMe** component follows. The code that participates in mouse event handling is in **colored boldface**.

```java
import javax.swing.BorderFactory;
import javax.swing.JComponent;
import java.awt.*;
import java.awt.event.*;

public class ClickMe extends JComponent implements MouseListener {
    private Spot spot = null;
    private static final int RADIUS = 7;
    private Color spotColor = new Color(107, 116, 2); //olive

    /** Creates and initializes the ClickMe component. */
    public ClickMe() {
        addMouseListener(this);
        //Hint at good sizes for this component.
        setPreferredSize(new Dimension(RADIUS*30, RADIUS*15));
        setMinimumSize(new Dimension(RADIUS*4, RADIUS*4));

        //Request a black line around this component.
        setBorder(
            BorderFactory.createLineBorder(Color.BLACK));
    }

    /**
     * Paints the ClickMe component. This method is
     * invoked by the Swing component-painting system.
     */
```
public void paintComponent(Graphics g) {
/**
 * Copy the graphics context so we can change it.
 * Cast it to Graphics2D so we can use antialiasing.
 */
Graphics2D g2d = (Graphics2D)g.create();

//Turn on antialiasing so that painting is smooth.
g2d.setRenderingHint(
    RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);

//Paint the background.
g2d.setColor(Color.WHITE);
g2d.fillRect(0, 0, getWidth() - 1, getHeight() - 1);

//Paint the spot.
if (spot != null) {
    int radius = spot.getSize();
    g2d.setColor(spotColor);
    g2d.fillOval(spot.x - radius, spot.y - radius,
                radius * 2, radius * 2);
}
}

//Methods required by the MouseListener interface.
public void mousePressed(MouseEvent event) {
    if (spot == null) {
        spot = new Spot();
        spot.setSize(RADIUS);
    }
    spot.x = event.getX();
    spot.y = event.getY();
    repaint();
}
public void mouseClicked(MouseEvent event) {}
public void mouseReleased(MouseEvent event) {}
public void mouseEntered(MouseEvent event) {}
public void mouseExited(MouseEvent event) {}
}

API Documentation

To learn more about how ClickMe works, you need to learn about its superclasses — JComponent and Component. How do you find that information? The API documentation, which constitutes the specification for the classes that make up the Java platform, contains detailed descriptions of every class.

The API documentation for the Java 2 Platform is online at java.sun.com. It's helpful to have the API documentation for all releases you use bookmarked in your browser. You can find the 5.0 API documentation here:
To learn more about all the classes and interfaces from the Java platform used by the ClickMe class, you can look at the API documentation for the following classes.

- `javax.swing.JComponent`
- `java.awt.Graphics`
- `java.awt.Graphics2D`
- `java.awt.Dimension`
- `java.awt.Color`
- `javax.swing.BorderFactory`
- `java.awt.event.MouseListener`
- `java.awt.event.MouseEvent`
A complete discussion of using these classes them to create GUIs is in *Creating a GUI with JFC/Swing*.

**Summary**

The discussion in this chapter glossed over many details and left some things unexplained, but now you should have some understanding of what object-oriented concepts look like in code. Specifically, you should have a general understanding of the following:

- That a class is a blueprint for objects
- That objects are created from classes
- How to create an object from a class
- What constructors are
- How to initialize objects
- What the code for a class looks like
- What class variables and methods are
- What instance variables and methods are
- How to find out what a class's superclass is
- That an interface is a protocol of behavior
- What it means to implement an interface