OBJECTS AND CLASSES

Natural-World Objects
In our natural world, an object is something that is an instance of a class, which is a larger group of like things.

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>You and I are instances of the class of objects known as people.</td>
</tr>
<tr>
<td>A pen is an instance of the class of writing implements.</td>
</tr>
</tbody>
</table>

Natural-world objects have both state and behavior. State is a condition of being. Behavior embodies actions that an object performs.

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A person has an age and height. Measure height is a possible action that a person object may perform.</td>
</tr>
<tr>
<td>A six-sided die (such as is used for a game of chance) has six surfaces, which can be rolled. The roll results in one of the sides facing up.</td>
</tr>
<tr>
<td>A calendar is set to a particular date and can be advanced from one day to the next.</td>
</tr>
</tbody>
</table>
**Java Objects**
Object-oriented computer programs have the capability of creating computer objects that represent natural-world objects.

**Example**
A computer program that simulates a dice game such as Craps can create computer objects that model the behavior of the dice.

A *Java object* combines data and operations within a single unit so that it can realistically model the state and behavior of the natural-world object it represents. Programmers borrow the natural world terminology and say that *an object is an instance of a class*. The object’s data is stored within variables that are internal to it, which are called *instance variables*. The computer object’s operations are called *methods*.

**Example**
In order to realistically model a six-sided die, the computer object the represents it must store the face up value in an instance variable and have a method that simulates the roll of the die. We might picture such an object as:

```
Die object

Instance variable ➔ faceUp [6]
Method ➔ roll( )
```

A computer object simulating a calendar may have instance variables for the month, day and year and a method that advances the current date by one day.

```
Calendar object

month [7]
day [4]
year [2015]
advanceDate( )
```
**Java Classes**

In the natural world, a class is like a collection – a large group of objects. A *Java class* is a plan or a blueprint that tells the JVM how to build objects of the class.

**Example**

This plan describes a house. Many houses can be built according to this plan and each house built from it contains the same rooms. Each house can have different color and decorating schemes.

The Java class contains two kinds of components – field declarations and method declarations. **Field declarations** specify the variables that are to be included in each object of the class. **Method declarations** specify the methods of each object.

**Example**

This Java class describes a Java object that models a six-sided die. According to the plan laid out by the class, each `Die` object is to include an instance variable named `faceUp` to hold the face up value of the die and a method named `roll` to simulate the rolling action.

```java
// A Die object models a 6-sided die used for dice games.
public class Die {
    public int faceUp;   // number shown facing up
    public void roll()  // roll the die
    {
        // randomly select a number from 1 to 6
        faceUp = (int)(Math.random() * 6 + 1);
    }
}
```
Like the house plan, all objects that are built from a Java class have the same instance variables and methods. But just as each house can be a different color, each object can have different data stored in the instance variables.

**Example**
This picture shows two different **Die** objects built from the Java class given in the previous example. Both have the same structure consisting of the instance variable `faceUp` and the method `roll`. Each has different data stored within its instance variable.

<table>
<thead>
<tr>
<th>Die object</th>
<th>Die object</th>
</tr>
</thead>
<tbody>
<tr>
<td>faceUp 3</td>
<td>faceUp 6</td>
</tr>
<tr>
<td>roll()</td>
<td>roll()</td>
</tr>
</tbody>
</table>
**Constructors**
A *constructor* is a special method used to initialize an object’s fields when it is built. The identifier of a constructor is exactly the same as the identifier of the class, right down to its upper and lower-case spelling. A class can have an *explicit constructor*, which is explicitly written out in the class. If there is not explicit constructor then the Java compiler automatically gives the class an *implicit constructor*.

*Example*
The Java class **Die** shown on page 3 has an implicit constructor named **Die( )**.

*Example*
Below, we give the **Die** class an explicit constructor that initializes the die object by rolling it. This way every **Die** object, when built, has a face up value in the range 1 to 6. This seems a more realistic model of a real die which never has a face up value of 0.

```java
1  // A Die object models a 6-sided die used for dice games.
2  public class Die
3  {
4      public int faceUp;   // number shown facing up
5
6      public Die( )       // explicit constructor
7      {
8          roll( );
9      }
10
11     public void roll( ) // roll the die
12     {
13         // randomly select a number from 1 to 6
14         faceUp = (int)(Math.random( ) * 6 + 1);
15     }
16 }
```
**The Static Modifier**

In our natural world, objects interact in ways requiring that they have access to their own internal data as well as data shared by other objects. Behaviors can also specific to a single object or shared among a group of objects.

**Example**

Imagine a group of people gambling at poker. Everybody antes the same amount into a single pot, but each player has his or her own bankroll.\(^1\)

The act of betting is specific to each player because each determines how much to bet. The ante before the deal, however, must be agreed upon by all players.

Java models this natural world concept by allowing the programmer to create fields and methods within a class that are shared by all of its objects. This is done by using the keyword `static` in the field or method declaration.

**Example**

Here’s a class modeling the poker player of the previous example.

```java
// A class that models a person that plays poker.
public class PokerPlayer {
    public static int ante; // initial wager
    public static int pot;   // pot of all wagers
    public int bankroll;     // amount available to player

    public PokerPlayer( int bank )
        // Build a player object with the given bankroll.
        
    public static void setAnte( int amount )
        // Set the ante to the given amount.
    
    public void bet( int amount )
        // Bet the given amount.
}
```

---

\(^1\) For those of you who aren’t poker players, *ante* is the amount each player bets before the cards are dealt, *pot* is the money all players collectively wager during a hand and *bankroll* is each player’s personal stash of money.
When the computer builds objects from a Java class, a static field becomes a *class variable* that all objects of the class share. A non-static field becomes an *instance variable* for which each object has its own copy. Likewise, a static method becomes a *class method* that all objects share whereas a non-static method becomes an *instance method* within each object.

<table>
<thead>
<tr>
<th>Meaning and Association of Static and Non-Static Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile-Time Construct</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>static field</td>
</tr>
<tr>
<td>non-static field</td>
</tr>
<tr>
<td>static method</td>
</tr>
<tr>
<td>non-static method</td>
</tr>
</tbody>
</table>

**Example**

This picture shows two different *PokerPlayer* objects built from the Java class given in the previous example. Each object has its own instance variable *bankroll*, its own constructor and its own instance method *bet*. Both objects share a single class variable *ante* and a single class variable *pot*. Also, both objects can call the single class method *setAnte*.

![PokerPlayer object 1](bankroll 1000)  
*PokerPlayer* object 1  
*bankroll* 1000  
*constructor*( int bank )  
*bet* ( int amount )  

![PokerPlayer object 2](bankroll 5000)  
*PokerPlayer* object 2  
*bankroll* 5000  
*constructor* ( int bank )  
*bet* ( int amount )  

**Class Variables and Methods**

- *PokerPlayer*.ante 1
- *PokerPlayer*.pot 7
- *PokerPlayer*.setAnte( int amount )
Exercises

This incomplete Java class describes a computer object modeling a kitchen timer.

Answer the questions that follow it.

```java
/* A Timer object models a kitchen timer that counts down seconds from some initial value. */

public class Timer
{
    public int secStart;   // initial number of seconds
    public int secElapsed; // how many have gone by
    public int secLeft;    // how many are left

    public Timer( )       // constructor
    { . . . }

    public void reset( )  // reset timer to initial seconds
    { . . . }

    public void start( )  // start timer from seconds left
    { . . . }

    public void pause( )  // pause timer
    { . . . }
}
```

1. Circle and label its field declarations.
2. Circle and label its method declarations.
3. Does it have an implicit constructor?
4. If `Timer` has an explicit constructor, circle and label it.
A system that monitors radiation around a nuclear power uses Geiger counters that record the ambient radiation and displays the level on a LCD screen in different colors – black for normal, yellow for alert, red for high alert.

Each Geiger counter stores its own recorded radiation level but the threshold levels for determining yellow and red alerts are shared by all objects.

The Java class outlined below appropriately models a class for building Geiger counter objects.

Answer the questions that follow it.

```java
// To model a Geiger counter.
public class GeigerCounter {
    private static int yellow; // lowest level for yellow alert
    private static int red;    // lowest level for red alert
    public  int level;         // level detected by this device

public GeigerCounter( )
    // Construct new GeigerCounter object.
{ . . . }

    public static void setYellowAlert( int yellow )
    // Set level of radiation for yellow alert.
{ . . . }

    public static void setRedAlert( int red )
    // set level of radiation for red alert.
{ . . . }

    public void display( )
    // Display radiation level in proper color.
{ . . . }
}
```

5. Circle and label its static field declarations.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6.</td>
<td>Circle and label its non-static field declarations.</td>
</tr>
<tr>
<td>7.</td>
<td>Circle and label its static method declarations.</td>
</tr>
<tr>
<td>8.</td>
<td>Circle and label its non-static method declarations.</td>
</tr>
<tr>
<td>9.</td>
<td>Does it have an implicit constructor?</td>
</tr>
<tr>
<td>10.</td>
<td>If <strong>GeigerCounter</strong> has an explicit constructor, circle and label it.</td>
</tr>
</tbody>
</table>