Objects

- **object**: An entity that contains data and behavior.
  - **data**: variables inside the object
  - **behavior**: methods inside the object
    - You interact with the methods; the data is hidden in the object.
    - A **class** is a type of objects.

- Constructing (creating) an object:
  ```
  Type objectName = new Type(parameters);
  ```

- Calling an object's method:
  ```
  objectName . methodName (parameters);
  ```
THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Blueprint analogy

**iPhone blueprint**

**state:**
- current song
- volume
- battery life

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

**iPhone #1**

**state:**
- **song** = "Watch Me (Whip/Nae Nae)"
- volume = 17
- battery life = 2.5 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

**iPhone #2**

**state:**
- **song** = "Don't Think Twice, It's All Right"
- volume = 9
- battery life = 3.41 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

**iPhone #3**

**state:**
- **song** = "Heart-Shaped Box"
- volume = 24
- battery life = 1.8 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

Copyright 2010 by Pearson Education
Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.

- **abstraction in an iPhone:**
  - You understand its external behavior (buttons, screen).
  - You don't understand its inner details, and you don't need to.
Classes and objects

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

- The **DrawingPanel** class is a template for creating **DrawingPanel** objects.

- **object**: An instance of a class. An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
Point class as blueprint

**Point class**

**state:**
int x, y

**behavior:**
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)

**Point object #1**

**state:**
x = 5, y = -2

**behavior:**
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)

**Point object #2**

**state:**
x = -245, y = 1897

**behavior:**
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)

**Point object #3**

**state:**
x = 18, y = 42

**behavior:**
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)

- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.
Clients of objects

- **client program**: A program that uses objects.
- **Example**: *Shapes* is a client of *DrawingPanel* and *Graphics*.

```java
Shapes.java (client program)
public class Shapes {
    public static void main(String[] args) {
        new DrawingPanel(...)
        new DrawingPanel(...)
        ...
    }
}
```

```java
DrawingPanel.java (class)
public class DrawingPanel {
    ...
}
```
The Object Concept

- **procedural programming**: Programs that perform their behavior as a series of steps to be carried out

- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects
  - Takes practice to understand the object concept
Our task

- In the following slides, we will implement a `Point` class as a way of learning about defining classes.
  - We will define a type of objects named `Point`.
  - Each `Point` object will contain x/y data called `fields`.
  - Each `Point` object will contain behavior called `methods`.
  - **Client programs** will use the `Point` objects.
Point objects (desired)

```java
Point p1 = new Point(5, -2);
Point p2 = new Point(); // origin, (0, 0)
```

- **Data in each Point object:**

<table>
<thead>
<tr>
<th>name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point's x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point's y-coordinate</td>
</tr>
</tbody>
</table>

- **Methods in each Point object:**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setLocation(x, y)</td>
<td>sets the point's x and y to the given values</td>
</tr>
<tr>
<td>translate(dx, dy)</td>
<td>adjusts the point's x and y by the given amounts</td>
</tr>
<tr>
<td>distance(p)</td>
<td>how far away the point is from point p</td>
</tr>
<tr>
<td>draw(g)</td>
<td>displays the point on a drawing panel</td>
</tr>
</tbody>
</table>
Object state: Fields

reading: 8.2
Point class, version 1

public class Point {
    int x;
    int y;
}

- Save this code into a file named Point.java.

- The above code creates a new type named Point.
  - Each Point object contains two pieces of data:
    - an int named x, and
    - an int named y.

- Point objects do not contain any behavior (yet).
Fields

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.

- Declaration syntax:
  
  ```
  type name;
  ```

- Example:
  
  ```java
  public class Student {
      String name;   // each Student object has a
      double gpa;    // name and gpa field
  }
  ```
Accessing fields

- Other classes can access/modify an object's fields.
  - access: `variable.field`
  - modify: `variable.field = value;`

- Example:

  ```java
  Point p1 = new Point();
  Point p2 = new Point();
  System.out.println("the x-coord is " + p1.x);  // access
  p2.y = 13;                                   // modify
  ```
A class and its client

- `Point.java` is not, by itself, a runnable program.
- A class can be used by `client` programs.

```java
public class Point {
    int x;
    int y;
}
```

```java
public class PointMain {
    public static void main(String args) {
        Point p1 = new Point();
        p1.x = 7;
        p1.y = 2;

        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;

        ...}
    }
}
```
public class PointMain {

    public static void main(String[] args) {

        // create two Point objects
        Point p1 = new Point();
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;

        System.out.println(p1.x + ", " + p1.y);  // 0, 2

        // move p2 and then print it
        p2.x += 2;
        p2.y++;
        System.out.println(p2.x + ", " + p2.y);  // 6, 1
    }
}

PointMain client example
Object behavior: Methods

reading: 8.3
Client code redundancy

- Suppose our client program wants to draw `Point` objects:

```java
// draw each city
Point p1 = new Point();
p1.x = 15;
p1.y = 37;
g.fillOval(p1.x, p1.y, 3, 3);
g.drawString("(" + p1.x + ", " + p1.y + ")", p1.x, p1.y);
```

- To draw other points, the same code must be repeated.
  - We can remove this redundancy using a method.
Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

```java
// Draws the given point on the DrawingPanel.
public static void draw(Point p, Graphics g) {
    g.fillOval(p.x, p.y, 3, 3);
    g.drawString("(" + p.x + ", " + p.y + ")", p.x, p.y);
}
```

- `main` would call the method as follows:

```java
draw(p1, g);
```
Problems with static solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws Points would need a draw method.

- The syntax doesn't match how we're used to using objects.
  
  ```java
  draw(p1, g);    // static (bad)
  ```

- The point of classes is to combine state and behavior.
  - The draw behavior is closely related to a Point's data.
  - The method belongs inside each Point object.

  ```java
  p1.draw(g);     // inside the object (better)
  ```
Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```java
public type name(parameters) {
    statements;
}
```

- same syntax as static methods, but without `static` keyword

Example:

```java
public void shout() {
    System.out.println("HELLO THERE!");
}
```
public class Point {
    int x;
    int y;

    // Draws this Point object with the given pen.
    public void draw(Graphics g) {
        ...
    }
}

- The draw method no longer has a Point p parameter.
- How will the method know which point to draw?
  - How will the method access that point's x/y data?
Point objects w/ method

- Each Point object has its own copy of the draw method, which operates on that object's state:

```java
Point p1 = new Point();
p1.x = 7;
p1.y = 2;

Point p2 = new Point();
p2.x = 4;
p2.y = 3;

p1.draw(g);
p2.draw(g);
```

```java
public void draw(Graphics g) {
    // this code can see p1's x and y
}
```

```java
public void draw(Graphics g) {
    // this code can see p2's x and y
}
```
The implicit parameter

- **implicit parameter:**
  The object on which an instance method is called.

- During the call `p1.draw(g);`
  the object referred to by `p1` is the implicit parameter.

- During the call `p2.draw(g);`
  the object referred to by `p2` is the implicit parameter.

- The instance method can refer to that object's fields.
  - We say that it executes in the *context* of a particular object.
  - `draw` can refer to the `x` and `y` of the object it was called on.
Each Point object contains a draw method that draws that point at its current x/y position.
Class method questions

- Write a method `translate` that changes a Point's location by a given $dx, dy$ amount.

- Write a method `distanceFromOrigin` that returns the distance between a Point and the origin, (0, 0).

Use the formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- Modify the Point and client code to use these methods.
public class Point {
    int x;
    int y;

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
}