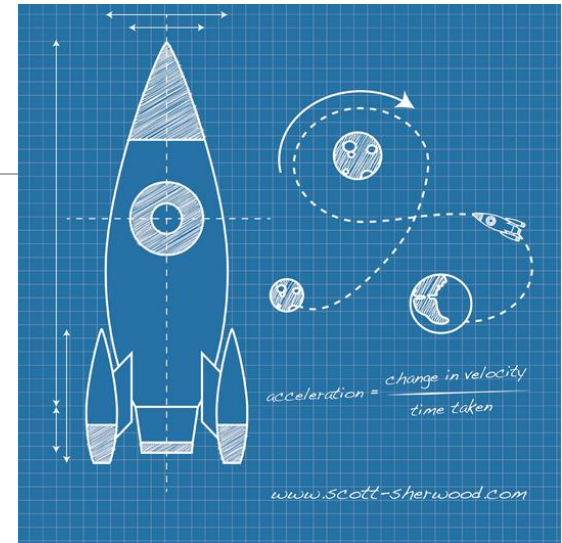


Chapter 1000: Classes

CHAPTER 8: CLASSES

BLUEPRINTS FOR OBJECTS AND
SPECIFICATIONS FOR THEIR ACTIONS



Object-Oriented Programming

Some Definitions:

- **Object-Oriented Programming:** Reasoning about a program as a set of objects rather than as a set of actions
- **Object:** A programming entity that contains state (data) and behavior (methods)
- **State:** A set of values (internal data) stored in an object
- **Behavior:** A set of actions an object can perform, often reporting or modifying its internal state
- **Client:** code that interacts with a class or objects of that class

TO RECAP (It's important!): Objects themselves are **NOT** complete programs. They are **COMPONENTS** that can be used as parts of larger programs. Pieces of code that create and use objects are known as **CLIENTS**. You will be asked to write client code that uses the classes you write. This is also sometimes called “driver” code.

Classes & Objects

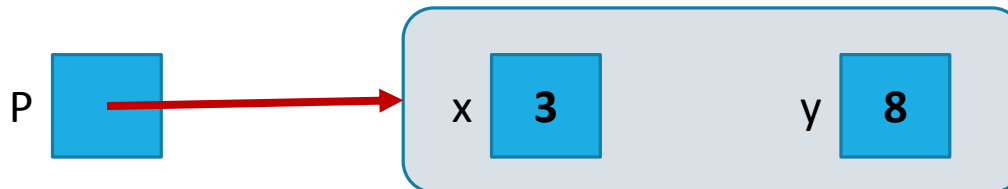
- Classes serve as **BLUEPRINTS** for new **OBJECTS**
- To create a new type of object in Java, you must create a new class and add code to it specifying:
 - The state stored in each object (its internal data)
 - The behavior each object can perform (its methods)
 - How to construct objects of that type (using constructors – covered soon!)
- Once this code is complete, the new class can be used to create objects of its type (**INSTANCES** of the class)
- These objects can then be used in client programs

Example: Point Objects

- We'll look at the Java Point class to see how classes work – this class is part of the java.awt package
- A Point object stores information on its x and y coordinates (expressed as integers)
- To create a specific Point object, you call a constructor (namely, code that specifies what values should be stored in x and y)
- For example, the code:

`Point p = new Point(3, 8);`

creates a new INSTANCE of the point class – a new Point object with an x-coordinate of 3 and a y-coordinate of 8.



Point Methods

- Now that you've created a new Point object, you probably want to DO stuff with it
- The table below gives some methods of Point Objects:

METHOD	DESCRIPTION
<code>translate(dx, dy)</code>	Translates the coordinates by the given amount
<code>setLocation(x,y)</code>	Sets the coordinates to the given values
<code>distance(p2)</code>	Returns the distance from this point to p2

- To use one of these methods, use DOT NOTATION:

`p.translate(-1, -2);` // subtracts 1 from current x value and 2 from current y value

- Another thing you can do with a Point object is refer to its x and y values using the dot notation:

`int sum = p.x + p.y;` // stores the sum of the current x value and the current y value

Client Code for Point Class

```
import java.awt.*;

public class PointExample {

    public static void main(String[] args) {
        // demonstrate how the translate method shifts the Point object's location
        // (i.e. alters the x and y coordinates)
        Point p = new Point(3, 8);
        System.out.println("Initially, p = " + p);
        p.translate(-1, -2);
        System.out.println("After translating, p = " + p);
    }
}
```

Problems @ Javadoc Declaration Console

<terminated> PointExample [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Apr 4, 2015, 6:26:43 PM)

Initially, p = java.awt.Point[x=3,y=8]

After translating, p = java.awt.Point[x=2,y=6]

NOTE: Although this Client code is created using the public class syntax, we are really only using it to create and manipulate instances of other classes (in this case, the Point class).

Object State: Fields

- We don't have to import the Java Point class – we can create our own Point class
- Our class will need fields to specify each object's state
- **Field:** A variable inside an object that makes up part of its internal state
- One of the main benefits of using classes in your programs is that it allows us to “hide” and protect the details of the inner working of the class (**encapsulation**)
- The focus is then on the external behavior of the object (**abstraction**)
- **Encapsulation:** Hiding the implementation details of an object from the clients of the object
- **Abstraction:** Focusing on essential properties rather than inner details

Object State: Private Fields

- In order to achieve this abstraction and encapsulation, we need to actually specify that we want the fields of our objects to be hidden
- We accomplish this by declaring them private with the Java keyword **private**, using the following syntax:

private <type> <name>;

- Fields can also be declared with an initial value:

private <type> <name> = <value>;

- For our Point class, we can declare the x and y fields as shown:

```
public class Point {  
    // encapsulated fields of Point objects  
    private int x;  
    private int y;  
  
    // . . .
```


Object Initialization: Constructors

- Constructors make it possible for new objects to be created
- A constructor's header begins with the keyword `public`, followed by the class's name and any parameters
- Since our `Point` objects should have initial `x` and `y` values, our constructor could be written as follows:

```
// constructs a new point with the given (x, y) location
public Point(int initialX, int initialY){
    x = initialX;
    y = initialY;
}
```

- Like instance methods, constructors execute on a particular object (the one that's being created with the `new` keyword):

`Point p1 = new Point(7, 2);`

Multiple Constructors

- A class can have multiple constructors to provide multiple ways for clients to construct objects of that class
- Each constructor must have a different SIGNATURE (i.e. number and type of parameters)
- If a class does not include a constructor, Java provides a parameterless DEFAULT CONSTRUCTOR that initialized all the new object's fields to a zero-equivalent value
- When we created our two-parameter constructor, we “lost” the default constructor provided by Java
- If we still want a second, parameterless constructor for our Point class, we can write one:

```
// constructs a Point object with location (0, 0)
public Point(){
    x = 0;
    y = 0;
}
```

Multiple Constructors and the keyword 'this'

- A common programming practice when writing classes with multiple constructors is to have one constructor contain the true initialization code and to have all the other constructors call it
- The syntax used to accomplish this is to have one constructor call another using the keyword 'this' followed by the parameters to pass to the other constructor in parentheses:

this(<expression>, <expression>, ... , <expression>)

- Thus, another way to write a second, parameterless constructor for our Point would be:

```
// constructs a Point object with location (0, 0)
public Point(){
    this(0,0); // calls Point(int, int) constructor
}
```


Common Programming Errors Involving Constructors

- **Using “void”** with a constructor will create a BUG – constructors are NOT supposed to have return types, as they are not normal instance methods
- **Redeclaring fields** in a constructor (by writing their types) will also create a BUG
- Both of these errors can be difficult to catch because the code will still compile correctly

Accessing Private Fields

- **HOWEVER**...Now that our fields are private, they are no longer accessible to code **OUTSIDE** the Point class – such as our **CLIENT CODE**.
- We need a way to allow client code to get information from class objects without having direct access to the object's field variables – we do this with **ACCESSOR METHODS**
- **Accessor** : An instance method that provides information about the state of an object without modifying it
- For our Point class, useful accessor methods will be:

These **ACCESSOR** methods return the values stored in the Point object's fields so that this data can be used externally in client code.



```
// accessor methods
public int getX(){
    return x;
}

public int getY(){
    return y;
}
```

Object Behavior: Methods

- Our Point class will also need instance methods that specify behavior
- **Instance Method:** A method inside an object that operates on that object
- You've already seen an example of instance methods in the accessor methods just discussed
- More generally, instance methods of an object describe the messages to which that object can respond
- In the client code we looked at earlier, we used the Java Point Class's translate method. Now we can write our own:

```
// shifts the point's location by the given amount
public void translate(int dx, int dy){
    x += dx;
    y += dy;
}
```

Object Behavior: Mutators

- The translate method is an example of a **MUTATOR**
- **Mutator:** An instance method that modifies the object's internal state
- In general, a mutator assigns a new value to one of the object's fields
- By convention, mutators often have names that begin with “set,” as in **setX** or **setY** or **setLocation**
- Usually a mutator has a **void** return type
- Mutators often accept parameters that specify the new state of the object or that specify the amount by which to modify the object's current state

The Implicit Parameter

- Instance method code has an **IMPLIED KNOWLEDGE** of the object on which it operates. This object is called the implicit parameter.

- **Implicit Parameter:** The object that is referenced during an instance method call

- Example of a call of an instance method on an object:

p1.translate(11, 6);

- During this call, p1's translate method is passed the parameters 11 and 6
- The **IMPLICIT PARAMETER** here is **p1's object**, thus the statements in the translate() method body:

x += dx; and y += dy;

affect **p1.x** and **p1.y**, respectively

The Keyword 'this'

- The implicit parameter is actually a special reference that is set each time an instance method is called
- The keyword '**this**' allows us to refer directly to the implicit parameter
- **this**: A Java keyword that allows you to refer to the implicit parameter inside a class
- The general syntax for using the keyword **this** to refer to fields is:

this.<field name>

Shadowed Variables

- Shadowing occurs when a field is obscured by another variable with the same name
- This can happen when a field has the same name as a parameter or local variable in a class
- For example, our 2-parameter Point constructor could have been written as:

```
// constructs a new point with the given (x, y) location
public Point(int x, int y){
    this.x = x;
    this.y = y;
}
```

- Using the keyword '**this**' clarifies that the variables on the left hand side are the **field variables** of the object, and prevents them from being confused with the parameter variables with the same names

The toString Method

- When a Java program is printing an object or concatenating it with a String, the program calls a special method, **toString**, on the object to convert it into a String
- The **toString** method is an instance method that returns a String representation of the object
- This method accepts **NO PARAMETERS** and has a String **return type**
- If you don't write a **toString** method for your class, Java provides a default version that returns the class name followed by a @ sign and some letters and numbers related to the object's address in memory
- If you write your own **toString** method (such as this one for the Point class), it replaces Java's default method:

```
//returns a String representation of this point
public String toString(){
    return "(" + x + ", " + y + ")";
}
```

Class Invariants

- **Class Invariant:** An assertion about an object's state that is true for the lifetime of that object
- Class invariants are related to preconditions, postconditions, and assertions
- Mutator methods are not allowed to break invariants – a class invariant should be treated as an implicit postcondition of every method in the class
- Invariants demonstrate the importance of proper encapsulation. Without proper encapsulation, invariants could not be properly enforced, since buggy or malicious clients could introduce invalid states by setting object field values directly
- Encapsulation gives the class much better control over how clients can use its objects, making it impossible for a client program to violate the class invariant

Changing Internal Implementations

- Another benefit of encapsulation is that it allows us to make internal design changes to a class without affecting clients of that class
- Because the class is encapsulated, the internal representation of a class does not necessarily have to match the external view seen by the client
- The implementation of the object's internal state can be changed, provided the methods of the class still produce the expected results from the client's external point of view
- This allows us to construct the internal state of the client in the way that is most efficient for us as programmers

In Conclusion...

- **Object-oriented programming** (OOP) is a different approach to writing programs that focuses on nouns or entities in a program, rather than on verbs or actions of a program
- In OOP, state and behavior are grouped in to **objects** that interact with each other
- A **class** serves as the **blueprint** for a new type of object, specifying that object's **state** and **behavior**
- A class can be asked to **construct** many objects (called “**instances**”) of its type
- The **data** for each object are specified using special variables called **fields**
- The **behavior** of each object is specified by writing **instance methods** in the class
- Instance methods exist **inside** an object and can access that object's **internal state**

In Conclusion...continued

- The **toString** method returns the **text representation** of an object
- **Constructors** are used to initialize the state of new objects as they are created
- The constructor is called when **external client code** creates a new object using the '**new**' keyword
- The keyword '**this**' can be used to have an object refer to **itself**
- The keyword '**this**' can also be used when a class has multiple constructors and one constructor wishes to call another
- Objects can protect their internal data from unwanted external modification by declaring them to be **private** (**encapsulation**)
- Encapsulation provides **abstraction**, so clients can use objects without being aware of their internal implementation