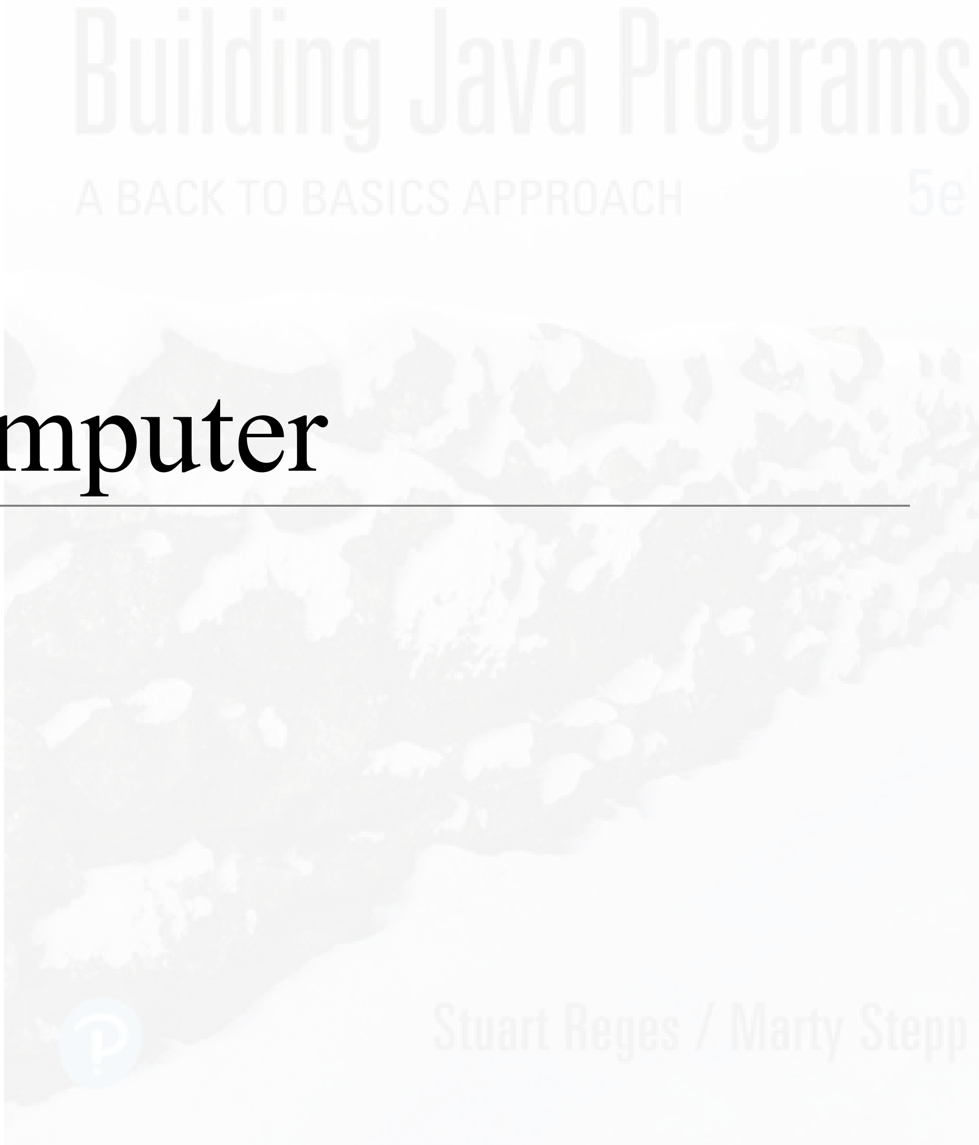
Introduction to Computer Programming

Chapter 8: Class

Instructor: Varik Hoang



Building Java Programs

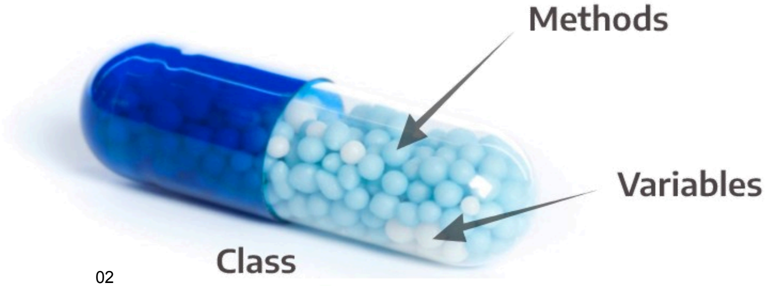
A BACK TO BASICS APPROACH

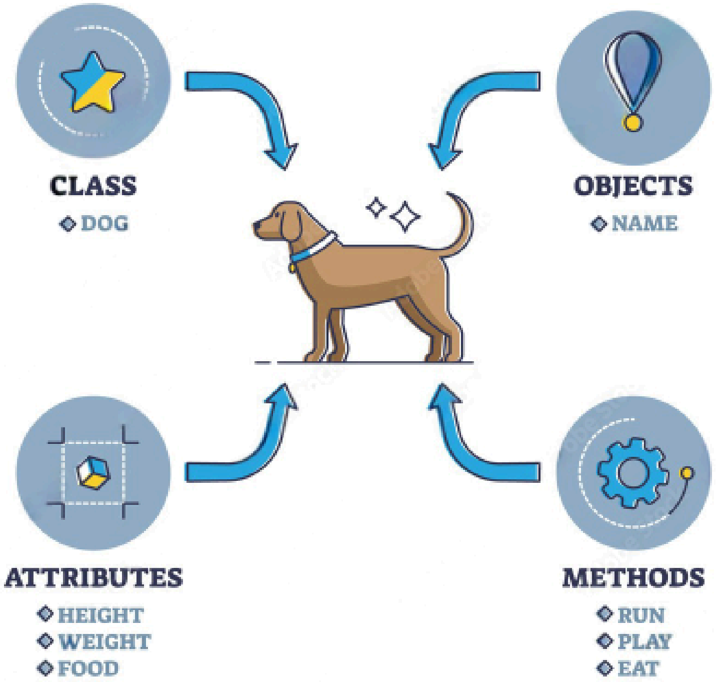
5e

Stuart Reges / Marty Stepp

Objectives

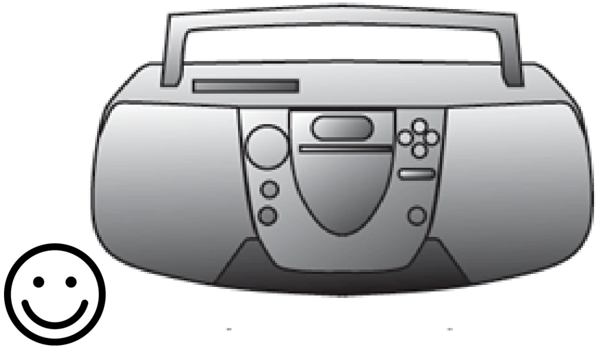
- Object-Oriented Programming
- Classes and Objects
- Object Class
  - Fields
  - Methods
  - Constructors
- Encapsulation

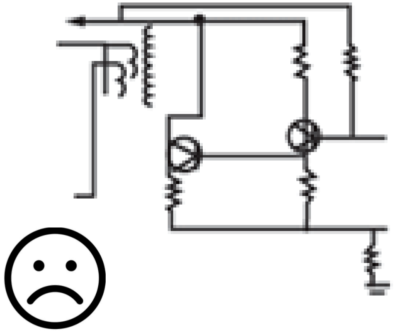




Encapsulation

- **Encapsulation**: to hide the implementation details of an object from clients





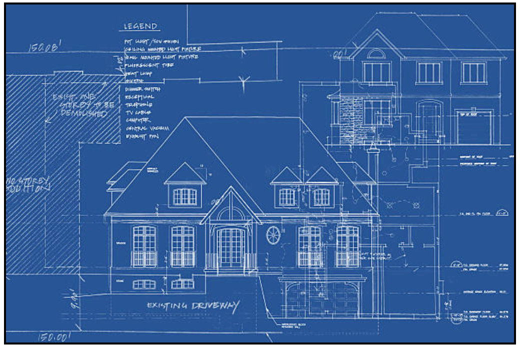
- The **Point** class *is not encapsulated*

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

```java
// in the main method of the PointMain class
Point p1 = new Point(7, 2);
p1.x = 4; // client can have direct access to x
p1.y = 3; // client can have direct access to y

System.out.println("(" + p1.x + ", " + p1.y + ")");
```

Abstraction

- **Abstraction** is the process of *eliminating details* so as to arise at a concise conception or model of some entity, or process.
  - **Abstraction** is applied in computer science when determining the problem to solve, the resources to use, and the algorithms to employ.
- **Concretization** is the process of *filling in details* of a model or concept so as to arrive at a real world product of the abstraction.
  - **Concretization** (implementation) is applied in computer science when we have to transform an abstract algorithm into a program.



Private Fields

- Declare fields to be **private** to encapsulate them:

```
private int x;
private int y;
```

- The syntax for declaring encapsulated fields:

```
private <type> <name>;
```

- Fields can also be declared with an initial value:

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

- Declaring fields **private** encapsulates the state of the object:
  - Fields are **visible to all of the code inside**
  - We can **no longer directly refer** to an object's fields

```
System.out.println(p1.x); // PointMain.java:11: x has private access in Point
```

Private Fields (cont.)

- Provide **accessor** methods to preserve the functionality of the client program:

```
public int getX() { return x; } // returns the x-coordinate of this point
public int getY() { return y; } // returns the y-coordinate of this point
```

- The client code to print a **Point** object's x and y values must be changed to:

```
System.out.println(p1.getX());
System.out.println(p1.getY());
```

- Add a new **mutator** method to set a **Point** to a new location

```
public void setLocation(int newX, int newY)
{
    x = newX;
    y = newY;
}
```

Private Fields (cont.)

- It is legal to call the instance methods from a constructor

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

- Eliminate the redundancy by calling the instance methods from another one:

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

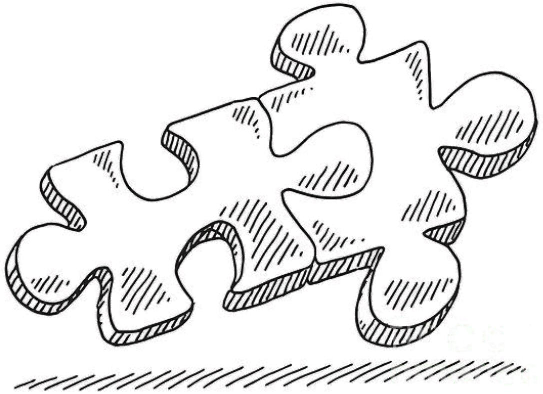
Class as Module

- **Module** is a reusable piece of software and stored as a class
  - Examples: **Math**, **Arrays**, **System**
- Module is **not** a complete program
  - It does not have a **main**. We *don't run it* directly
  - Modules are meant to be **utilized** by other client classes
- Syntax:

```
CLASS.METHOD(PARAMETERS);
```

- Examples:

```
Math.sqrt(25);
Math.pow(2, 4);
Arrays.toString(new int[] {7, 2});
```

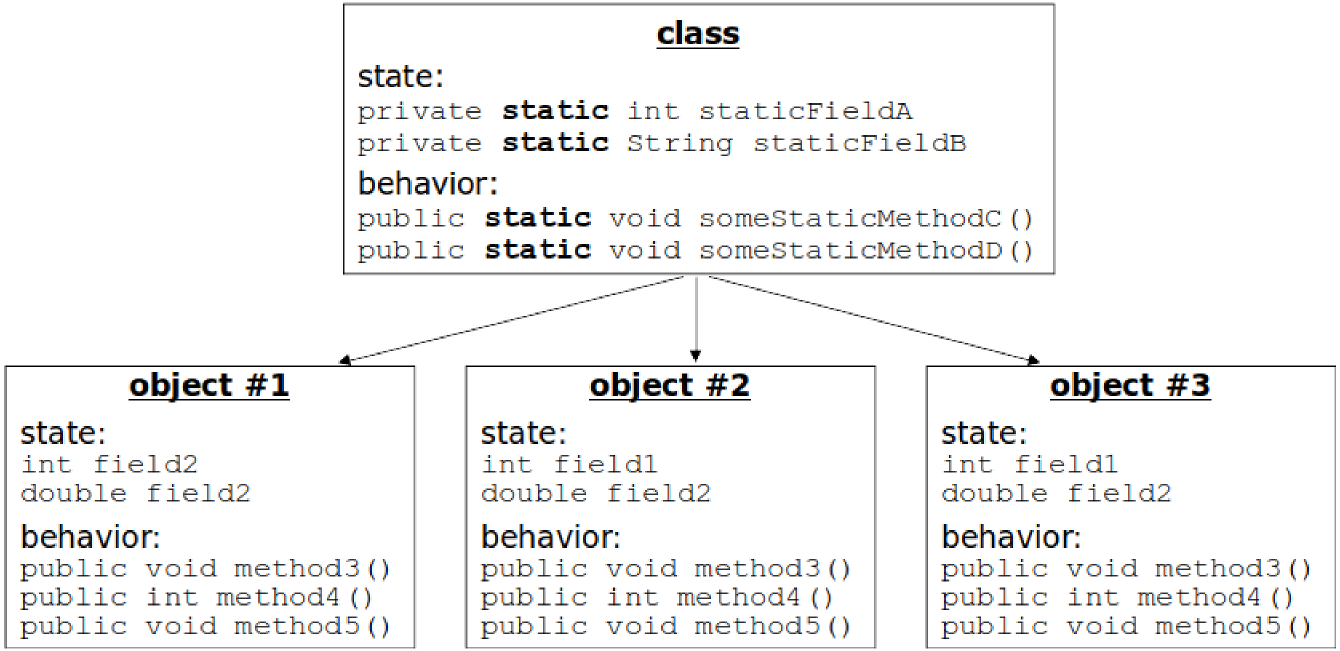


Math Module in Java Libraries

```
public class Math
{
    public static final double PI = 3.14159265358979323846;

    ...

    public static int abs(int a)
    {
        if (a >= 0)
        {
            return a;
        }
        else
        {
            return -a;
        }
    }


    public static double toDegrees(double radians)
    {
        return radians * 180 / PI;
    }
}
```

The **static** Members

- **static**: is part of a class (not part of an object)
  - Object classes can have static methods and fields.
  - Not copied into each object; shared by all objects of that class.



class

state:

```
private static int staticFieldA
private static String staticFieldB
```

behavior:

```
public static void someStaticMethodC()
public static void someStaticMethodD()
```

object #1

state:

```
int field2
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

object #2

state:

```
int field1
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

object #3

state:

```
int field1
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

The **static** Fields

- The **static** field is stored in the class instead of each object
  - A "shared" global field that all objects can access and modify
  - Like a class constant, except that its value can be changed
- Syntax:

```
private static type name;
private static type name = value;
```

- Examples:

```
private static boolean isPrime;
private static int theAnswer = 42;
```

The **static** Fields - Accessing & Modifying

- From inside the class where the field was declared

```
fieldName          // get the value
fieldName = value; // set the value
```

```
theAnswer
theAnswer = 36;
```

- From another class (if the field is **public**)

```
ClassName.fieldName          // get the value
ClassName.fieldName = value; // set the value
```

```
Point.counter
Point.counter = 8;
```

- Generally static fields are **not public** unless they are **final**
  - The **final** means the value **once assigned will not be changed**
  - The **final** fields **cannot** be **null**

The **static** Fields - Example

```
public class BankAccount
{
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount()
    {
        objectCount++;    // advance the id, and
        id = objectCount; // give number to account
    }

    ...

    public int getID()
    {
        return id;
    }
}
```

The **static** Methods

- The **static** field is stored in the class instead in an object
  - Shared by all objects of the class, not replicated
  - Does not have any implicit parameter **this**
    - **Cannot access** any particular object's fields
- Syntax:

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

- Examples:

```
public static boolean isNegative(int number)
{
    return number < 0;
}
```

The **static** Methods - Example

```
public class BankAccount
{
    private static int objectCount = 0;

    // clients can call this to find out # accounts created
    public static int getNumAccounts()
    {
        return objectCount;
    }

    private String name;
    private int id;

    public BankAccount()
    {
        objectCount++;    // advance the id, and
        id = objectCount; // give number to account
    }

    ...

    public int getID() { return id; }
}
```

Summary of Java Classes

- A class is used for any of the following in a large program
  - A **program** has a **main** and perhaps other **static** methods
    - Does not usually declare any **static** fields (except **final**)
    - Example: **PointMain**, **GradeBook**, etc.
  - An **object class** defines a new type of objects
    - Declares object fields, constructor(s), and methods
    - Might declare **static** fields or methods, but these are less of a focus
    - Should be encapsulated (all fields and **static** fields **private**)
    - Example: **Point**, **Record**, **Scanner**, etc.
  - A **module** is utility code that implemented as **static** methods
    - Example: **Math**, **Arrays**, etc.

Practice

- Design a class called **Point3D**, which represents **a point in three-dimensional space**, similar to the existing **Point** class for two-dimensional space.
  - **Point3D** should consist of instance fields x, y, and z coordinates (int)
  - **Point3D** should consist of **static** fields count (**int**)
  - **Point3D** should provide 3 different types of constructors
    - Parameterless constructor to initialize the point to the origin (**0**, **0**, **0**)
    - Parameterized constructor with x, y, and z to set point to specified coordinates
    - Copy constructor to create a new **Point3D** instance based on an existing **Point3D**
  - **Point3D** should provide 3 instance methods:
    - **distance**(): calculate and return the distance between the current **Point3D** instance and the origin (**0**, **0**, **0**)
    - **distance**(**Point3D** other): calculate and return distance between current **Point3D** instance and another given Point3D instance
    - **setLocation**(**int** x, **int** y, **int** z): set new x, y, and z coordinates for **Point3D** instance
  - **Point3D** should provide 1 **static** methods to return number of **Point3D** instances created

Questions & Answer

Thank You!