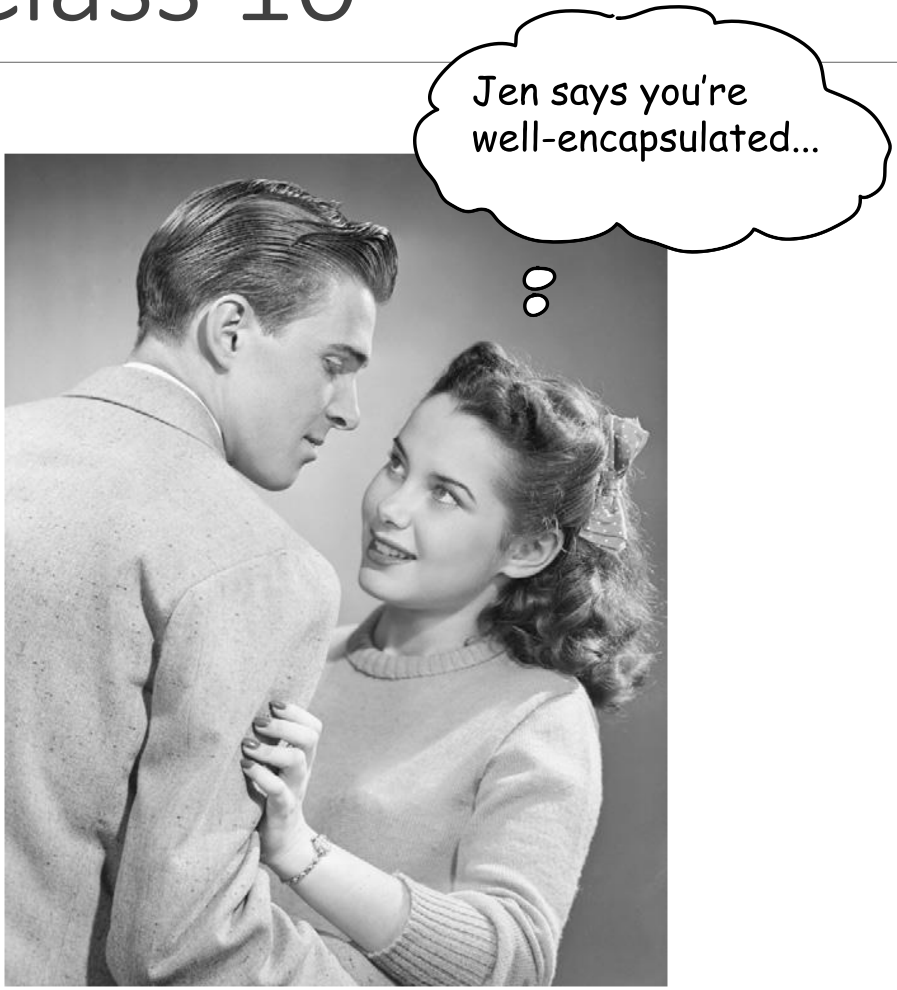
Java Class 10



Writing Classes and Using Objects

True ***object-oriented programming*** is based on defining classes that represent objects with well-defined characteristics and functionality

The programs we've written in previous examples have used classes defined in the Java Standard Class Library

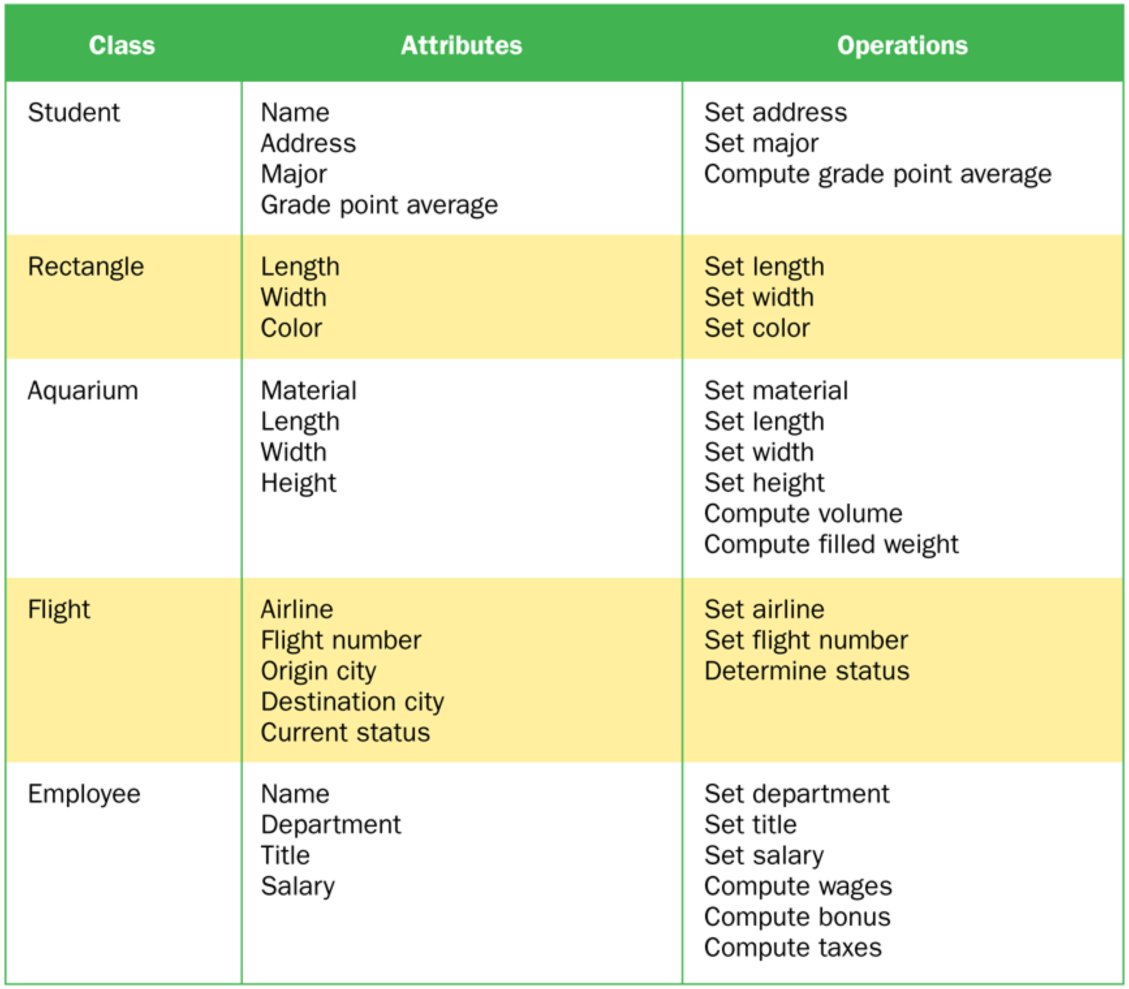
Now we will begin to design programs that rely on classes that *we will write ourselves*

The class that contains the `main` method is just the starting point of a program and is sometimes referred to as the ***driver***

The reserved word `class` means "I'm about to tell you what a new type of object looks like"

An object that we create from a class has attributes/state (data) and operations/behavior (methods)

Examples of Classes



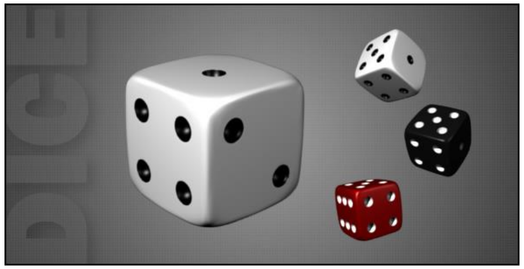
Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Classes and Objects

Consider a six-sided die (singular of dice)

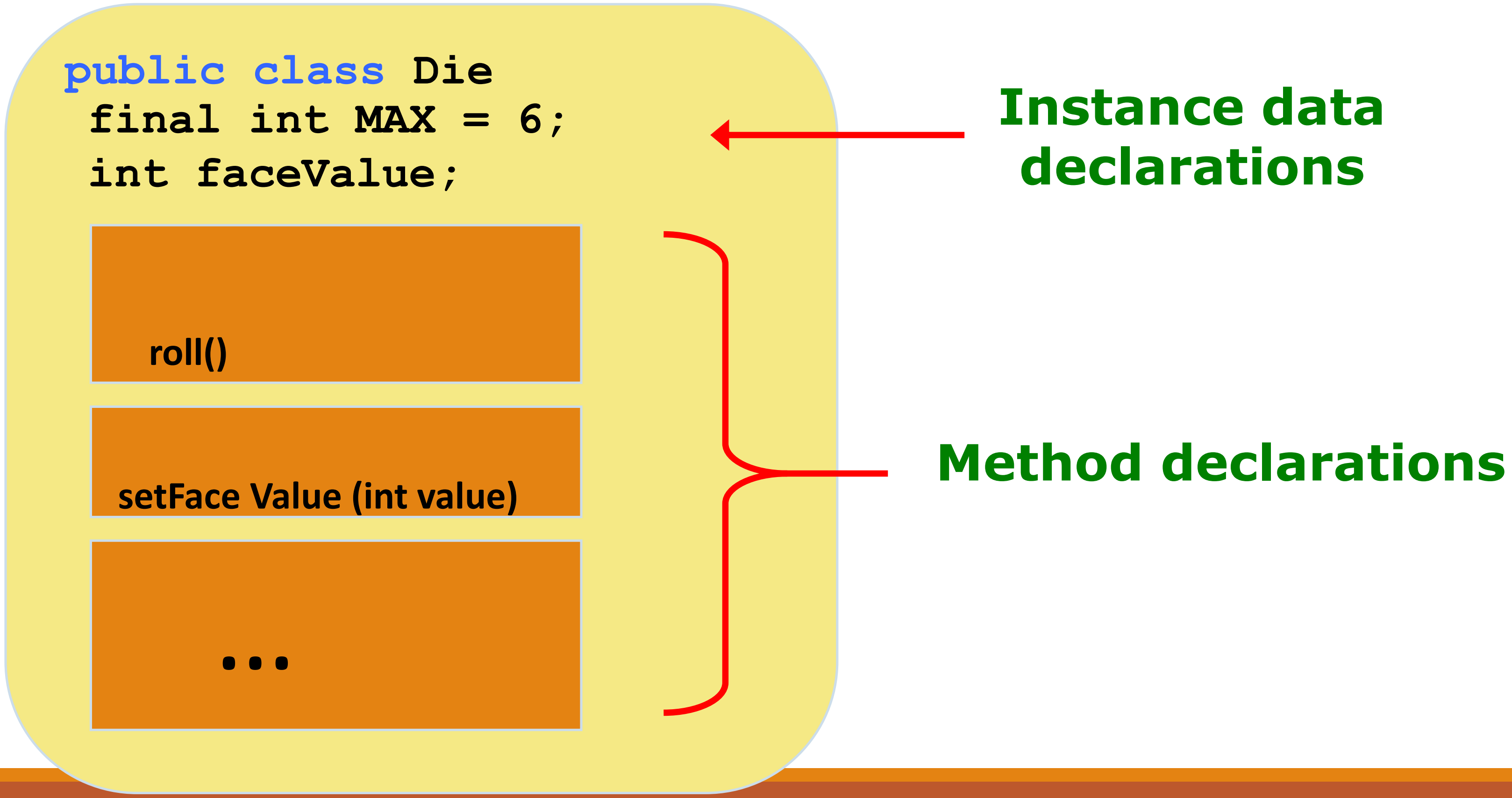
- We represent a die by designing a class called `Die` that models this state and behavior
- It's state can be defined as which face is showing
- It's primary behavior is that it can be rolled

Once we have created a Die class, we can instantiate as many die objects as we need for any particular program



Classes

A class can contain data declarations and method declarations



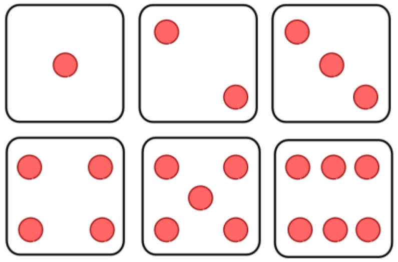
Instance Data

A variable declared at the class level (such as `faceValue`) is called ***instance data***

Each instance of an object has its own set of instance variables

That's the only way two objects can have different states

Each time a `Die` object is created, a new `faceValue` variable is created for this particular object, and it is given an initial value by a special method called the ***constructor***



Even though each object of a class has its own data space, the objects *share the method definitions*

Anatomy of a Class

```
public class Die
{

private final int MAX = 6;
private int faceValue

public Die()
    {
        faceValue = 1;
    }

...

}
```

Class definition

Instance data

Constructor

The ***constructor*** has the same name as the class and creates the initial state of an object

Other methods

```java
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Rolls the die and returns the result.
   //-----------------------------------------------------------------
   public int roll()
   {
      faceValue = (int)(Math.random() * MAX) + 1;
      return faceValue;
   }

   //-----------------------------------------------------------------
   //  Face value mutator.
   //-----------------------------------------------------------------
   public void setFaceValue (int value)
   {
      faceValue = value;
   }

   //-----------------------------------------------------------------
   //  Face value accessor.
   //-----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Returns a string representation of this die.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = Integer.toString(faceValue);

      return result;
   }
}
```

```java
//********************************************************************
//  RollingDice.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of a user-defined class.
//********************************************************************

public class RollingDice
{
   //-----------------------------------------------------------------
   //  Creates two Die objects and rolls them several times.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Die die1, die2;
      int sum;

      die1 = new Die();
      die2 = new Die();

      die1.roll();
      die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
```

continue

continue

```
      die1.roll();
      die2.setFaceValue(4);
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

      sum = die1.getFaceValue() + die2.getFaceValue();
      System.out.println ("Sum: " + sum);

      sum = die1.roll() + die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
      System.out.println ("New sum: " + sum);
   }
}
```

continue

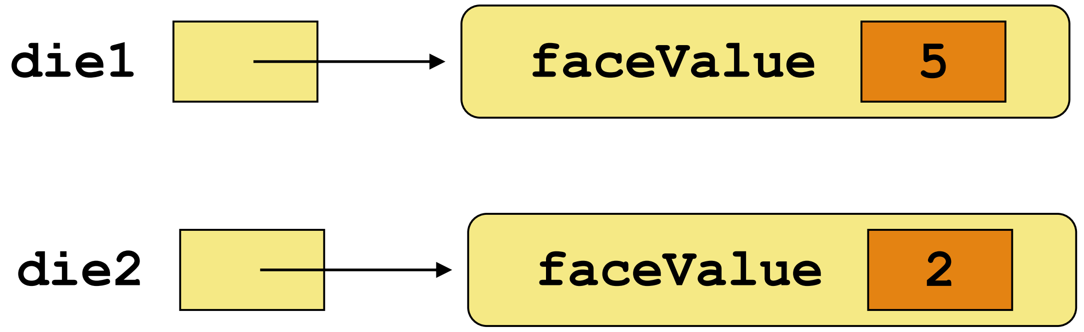
```
      die1.roll();
      die2.setFaceV
      System.out.pr                                    , Die Two: " + die2);

      sum = die1.ge                                    lue();
      System.out.pr

      sum = die1.roll() + die2.roll();
      System.out.println ("Die One: " + die1 + ", Die Two: " + die2);
      System.out.println ("New sum: " + sum);
   }
}
```

Sample Run

```
Die One: 5, Die Two: 2
Die One: 1, Die Two: 4
Sum: 5
Die One: 4, Die Two: 2
New sum: 6
```

Instance Data

We can depict the two `Die` objects from the `RollingDice` program as follows:



Each object maintains its own `faceValue` variable, and thus its own state

Dog Class

Each class declares:

- Instance variables: descriptive characteristics or states of the object (size and name of a dog)
- Methods: behaviors of the object or what the object can do (bark)

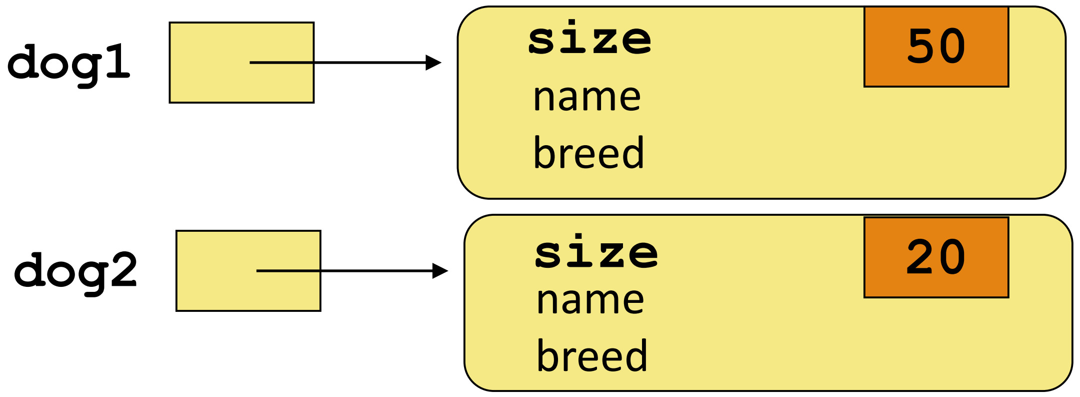
```
// Dog.java

public class Dog  {

   // instance variables
   int size;
   String name;
   String breed:

   // a method
   void bark() {
     System.out.println("Ruff! Ruff!");
   }
}
```

```
Dog dog1 = new Dog();

Dog dog2 = new Dog();
```

Instance Variables

A variable declared at the class level (such as `size`) is called an instance variable

Each time a `Dog` object is created, a new memory space is reserved for variable `size, name and breed` in this particular object



The objects of a class each have their own set of (different) values for their instance variables, and thus each has its own state

At the same time the objects of a class share the same method definitions

Instance Variables

The ***scope*** of a variable is the area in a program in which that variable can be referenced (used)

Variables declared at the class level can be referenced by all methods in that class

```
public class Dog  {
   // instance variables
   int size;
   String name;
   String breed:

   // a method
   void bark() {
     if (size > 40)
       System.out.println("Ruff! Ruff!");
     else
       System.out.println(”Yip! Yip!");
   }
}
```

```
dog1.size = 50;
dog1.bark();
```

```
Ruff! Ruff!
```

```
dog2.size = 20;
dog2.bark();
```

```
Yip! Yip!
```

Instance vs. Local Variables

1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

Lifetime: instance variables exist as long as the object exists

Scope: instance variables can be referenced by all methods in that class

Instance vs. Local Variables

❷ **Local** variables are declared within a method.

```
class AddThing {
    int a;
    int b = 12;

    public int add() {
        int total = a + b;
        return total;
    }
}
```

Lifetime: when the method finishes, all local variables are destroyed (including the parameters of the method)

Scope: local variables can only be used in that method

Instance vs. Local Variables

Instance variables always get a default value (but don't rely on it)

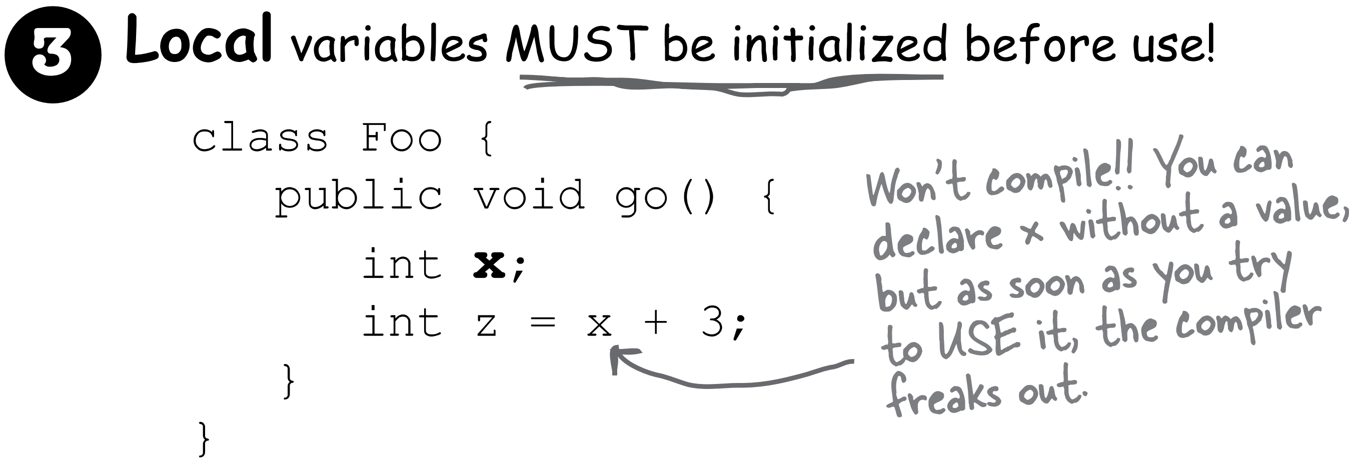
If you don't explicitly assign a value to an instance variable, the instance variable still has a value

Integers: 0

floating points: 0.0

booleans: false

references: null



3 **Local** variables MUST be initialized before use!

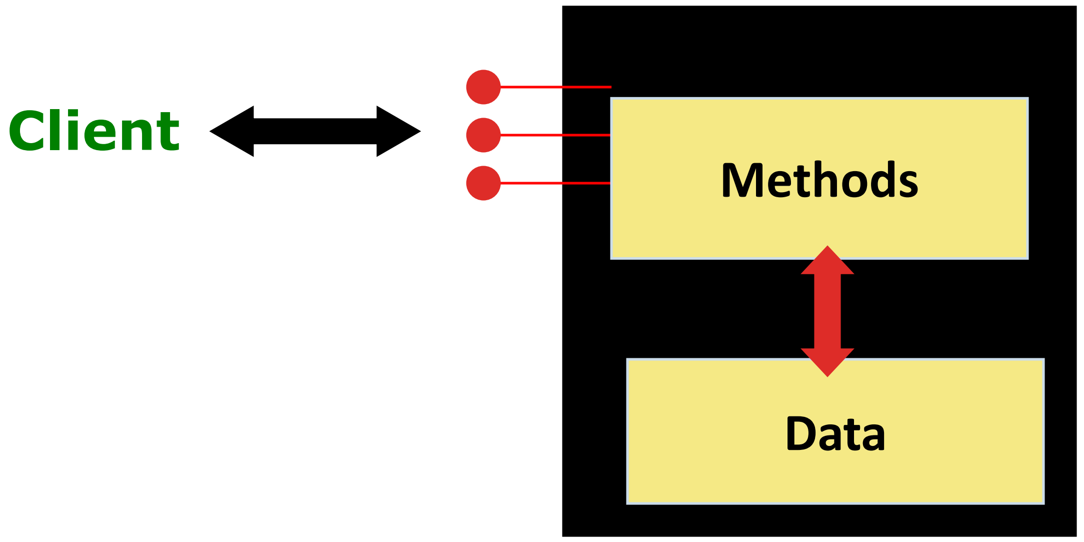
```
class Foo {
    public void go() {
        int x;
        int z = x + 3;
    }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

Encapsulation

An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client (class that uses the object)

The client object invokes the interface methods and they manage the instance data



Visibility Modifiers

In Java, we accomplish encapsulation through the appropriate use of ***visibility modifiers***

A *modifier* is a Java reserved word that specifies particular characteristics of a method or data

We've used the `final` modifier to define constants

Java has three visibility modifiers: `public`, `protected`, and `private`

Visibility Modifiers

Members of a class that are declared with ***public visibility*** can be referenced anywhere and anyone can use them

Members of a class that are declared with ***private visibility*** can be referenced only within that class and no one else can use them

Members declared without a visibility modifier have ***default visibility*** and can be referenced by any class in the same package but outside the package those same members appear as private

Visibility Modifiers

Public variables violate encapsulation because they allow the client to modify the values directly

Therefore instance variables should be declared with private visibility

It is acceptable to give a constant public visibility, which allows it to be used outside of the class

Public constants do not violate encapsulation because, although the client can access them, their values cannot be changed

Visibility Modifiers

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

Accessors and Mutators

Because instance data are private, a class usually provides services to access and modify data values

An ***accessor*** *method* returns the current value of a variable

A ***mutator*** *method* changes the value of a variable

The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value

They are sometimes called "getters" and "setters"

continue

```
   //-----------------------------------------------------------------
   //  Rolls the die and returns the result.
   //-----------------------------------------------------------------
   public int roll()
   {
      faceValue = (int)(Math.random() * MAX) + 1;
      return faceValue;
   }

   //-----------------------------------------------------------------
   //  Face value mutator.
   //-----------------------------------------------------------------
   public void setFaceValue (int value)
   {
      faceValue = value;
   }

   //-----------------------------------------------------------------
   //  Face value accessor.
   //-----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }
```

continue

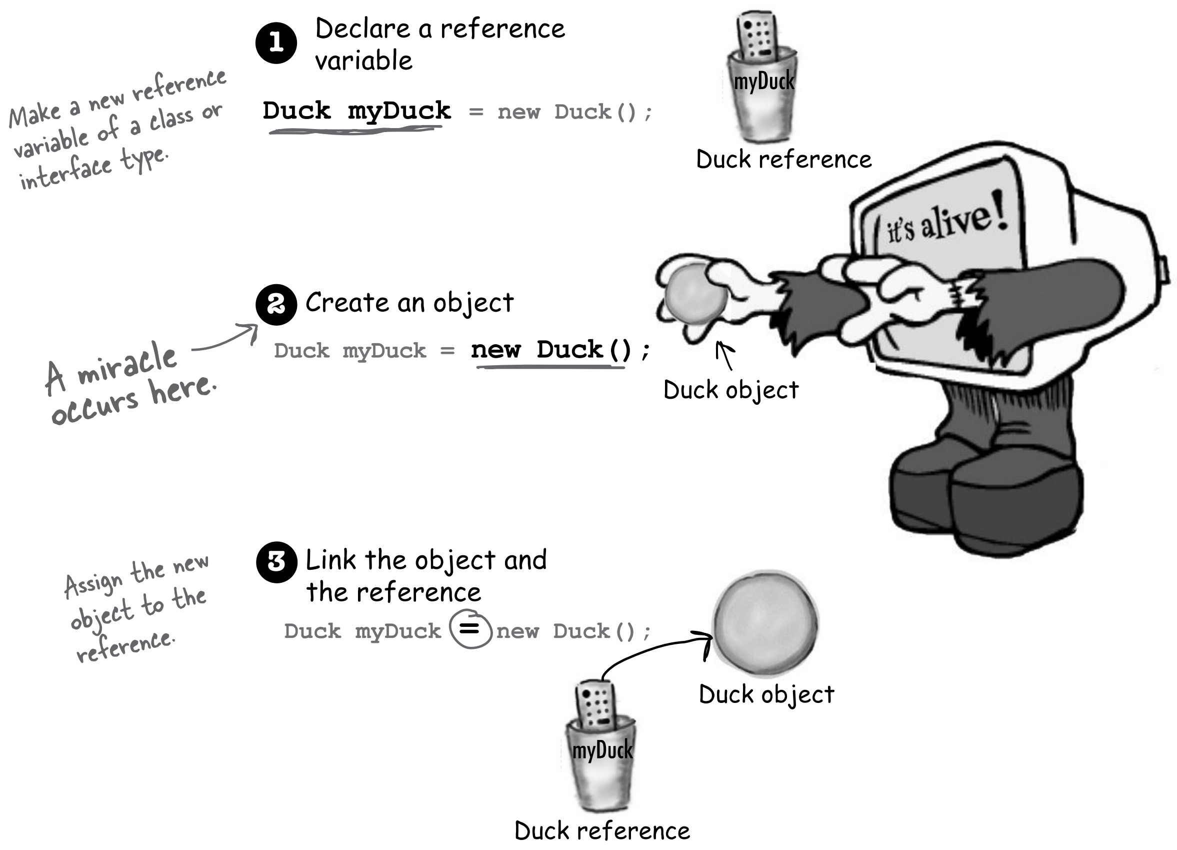
Mutator Restrictions

The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state

A mutator is often designed so that the values of variables can be set only within particular limits

For example, the `setFaceValue` mutator of the `Die` class should restrict the value to the valid range (1 to `MAX`)

Recap: Three Steps of Creating Objects


1 Declare a reference variable
Make a new reference variable of a class or interface type.
myDuck
Duck reference
2 Create an object
A miracle occurs here.
it's alive!
Duck object
3 Link the object and the reference
Assign the new object to the reference.
Duck object
myDuck
Duck reference

```
Duck myDuck = new Duck();
Duck myDuck = new Duck();
Duck myDuck = new Duck();
```

Group Exercises

Ex: 4.1

Ex: 4.3

Ex: 4.4

PP: 4.2

Assignment for Class 11

Review Rolling Dice, Die

Read Chapter 4.4, 4.5

```
public class Lights
{
   //------------------------------------------------------------------
   //  Creates and exercises some Bulb objects.
   //------------------------------------------------------------------
   public static void main (String[] args)
   {
      Bulb b1 = new Bulb();
      Bulb b2 = new Bulb();
      Bulb b3 = new Bulb();

      System.out.println ("Is b2 on? " + b2.isOn());

      b2.setOn (true);
      b1.setOn (true);

      System.out.println (b1);
      System.out.println (b2);
      System.out.println (b3);
   }
}
```