Classes, Encapsulation, Methods and Constructors

- Class definitions
- Scope of Data
  - Instance data
  - Local data
- The `this` Reference
- Encapsulation and Java modifiers
- Reading for this Lecture: L&L, 4.1-4.5, & App E

Writing Classes

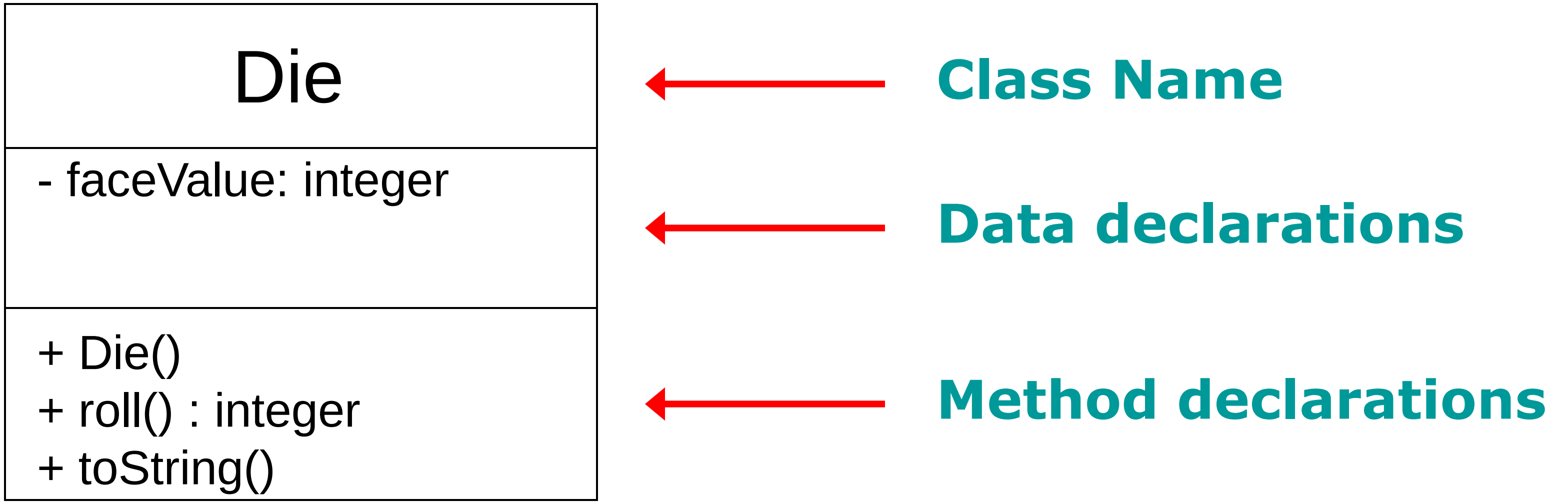
- True object-oriented programming is based on classes that represent objects with well-defined attributes and functionality
- The programs we’ve written in previous examples have used classes defined in the Java standard class library
- Now we will begin to design programs that rely on classes that we write ourselves

Classes and Objects

- An object has *state* and *behavior*
- Consider a six-sided die (singular of dice)
  - It’s state can be defined as the face showing
  - It’s primary behavior is that it can be rolled
- We can represent a die in software by designing a class called `Die` that models this state and behavior
  - The class serves as the blueprint for a die object
- We can then instantiate as many die objects as we need for any particular program

Classes

- A class has a name and can contain data declarations and/or method declarations
- A UML class diagram shows it as follows:



Die

- faceValue: integer

+ Die()
+ roll() : integer
+ toString()

Class Name

Data declarations

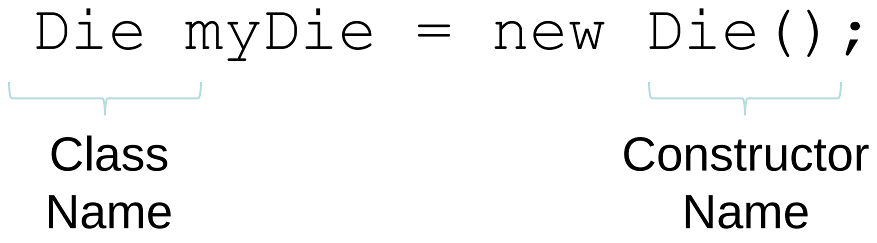
Method declarations

Classes

- The values of the attributes define the state of any object created from the class
- The functionality of the methods define the behaviors of any object created from the class
- For our `Die` class, an integer represents the current value showing on the face
- One of the methods would allow us to “roll” the die by setting its face value to a random value between one and six

Constructors

- A *constructor* is a special method that is used to set up an object when it is initially created
- A constructor has the same name as the class with no return type
- The `Die` constructor is used to set the initial face value of each new die object to one



```
Die myDie = new Die();
```

- We examine constructors in more detail later

The toString Method

- All classes that represent objects should define a `toString` method
- The `toString` method returns a string that represents the object in some way
- It is called automatically when a reference to an object is concatenated to a string or when it is passed to the `println` method

```
String s = “My die shows: ” + myDie;
System.out.println(myDie);
```

Data Scope

- The *scope* of data is the area in a program in which that data can be referenced (used)
- Data declared at the class level can be referenced by code in all methods defined in that class
- Instance data is declared at the class level and it exists for as long as the object exists
- Data declared within a method is called *local data*
- Data declared within a method can be used only within that method and exists only for as long as that method is executing

Data Scope

- Instance and local data

```
public class Die
{

  private int faceValue;


  public Die ()
  {
     int value = 1;
     faceValue = value;

  }
}
```

Local scope for value

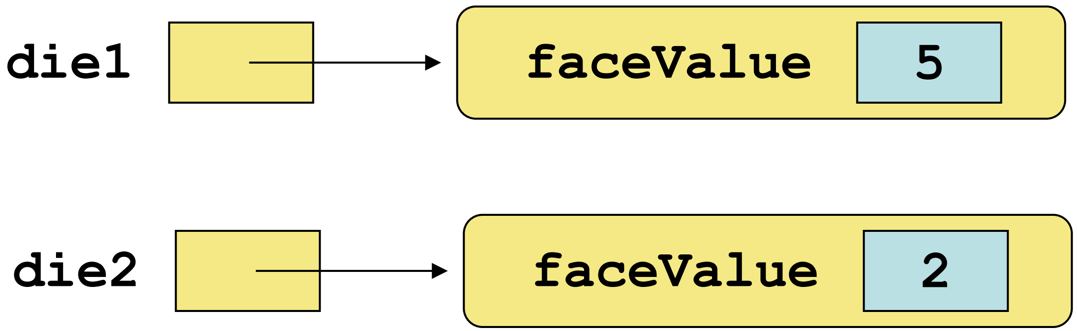
Class level scope for faceValue

Instance Data

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a new `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the code in the method definitions, but each object has its own data space in memory for instance data
- The instance data goes out of scope when the last reference to the object is set to null

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

Local Data

- Any variable defined inside the curly braces of a method (or inside any block statement, such as if/else clauses or bodies of loops):

```
public  String toString()
 {
    String result = “” + faceValue;
    return result;
 }
```

- The variable named `result` is accessible only inside this `toString()` method

The `this` Reference

- The `this` reference allows an object to refer to itself
- Inside the method, the object reference variable used to call it is not available (not in local scope)
- The `this` reference used inside a method refers to the object in which the method is being executed
- Suppose `this` is used in the Die class `toString()` method as follows:

```
return “” + this.faceValue;  // return string
```

- In these two invocations, `this` refers to and returns:

```
die1.toString()  → 5
die2.toString()  → 2
```

The `this` Reference

- The `this` reference can be used to distinguish the instance variable names of an object from local method parameters with the same names
- The `Account` class and its constructor can be written as shown on either of the next two slides
- Without the `this` reference, we need to invent and use two different names that are synonyms
- The `this` reference allows us to use the same name for instance data and a local variable or parameter in a method and resolves ambiguity

Without the `this` Reference

- A modified Die class could be written as follows:

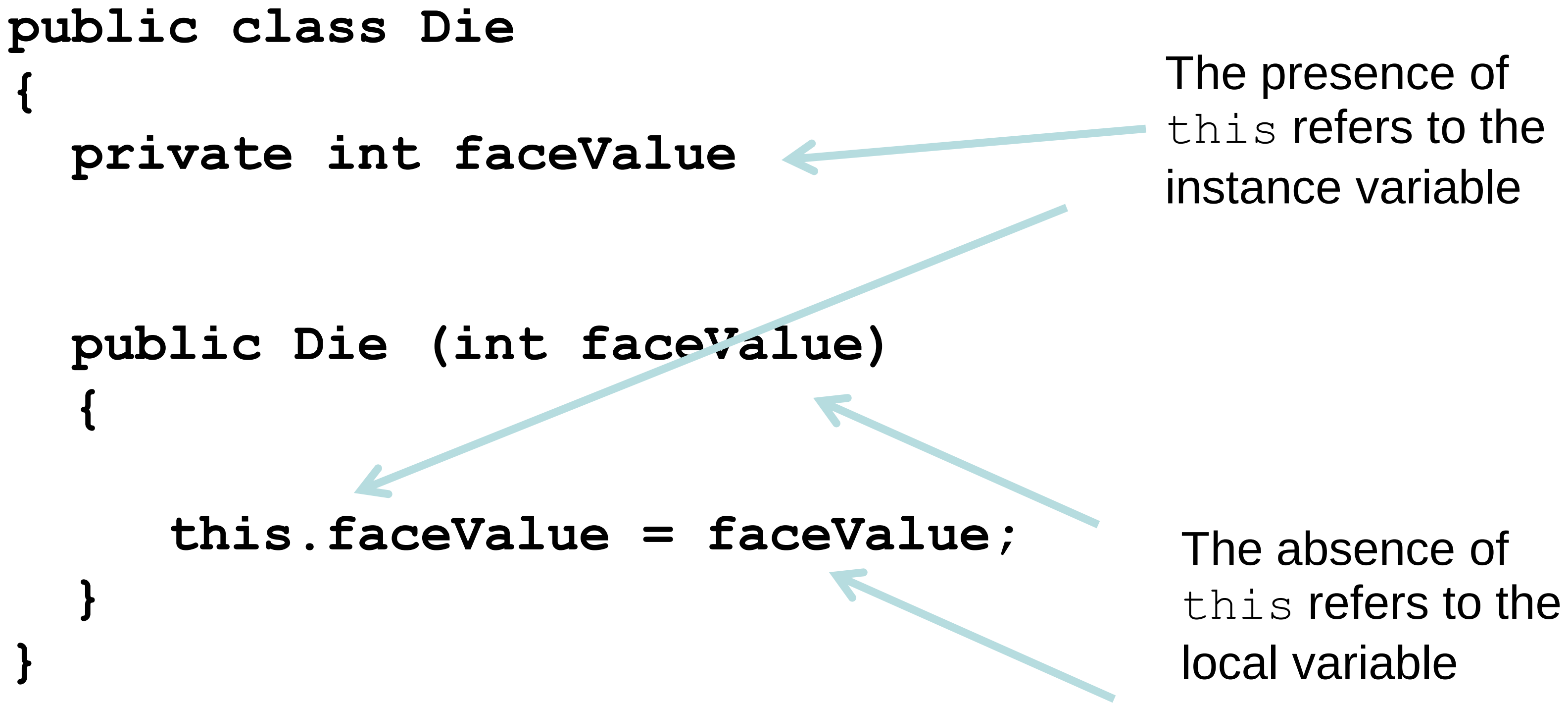
```
public class Die
{

  private int faceValue;


  public Die (int value)
  {

     faceValue = value;

  }
}
```

The instance variables have meaningful names

The local variables have similar but not identical names

With the `this` Reference

- The preferred method for writing it so we don't need to invent synonyms is as follows:



```
public class Die
{
  private int faceValue


  public Die (int faceValue)
  {

     this.faceValue = faceValue;
  }
}
```

The presence of `this` refers to the instance variable

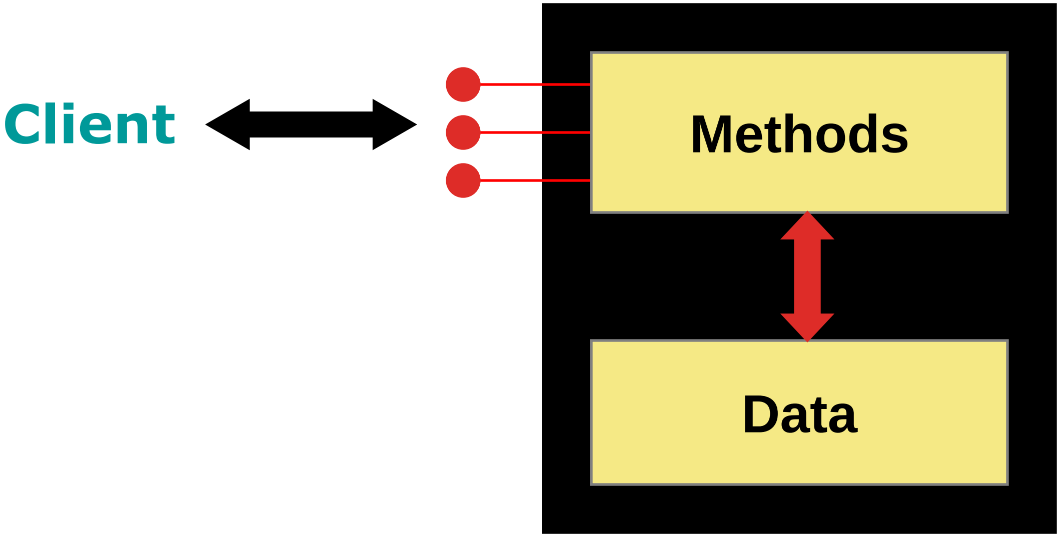
The absence of `this` refers to the local variable

Encapsulation

- We can take one of two views of an object:
  - internal - the details of the variables and methods of the class that defines it
  - external - the services that an object provides and how the object interacts with the rest of the system
- From the external viewpoint, an object is an *encapsulated* entity providing a set of specific services
- These services define the *interface* to the object

Encapsulation

- An object can be thought of as a *black box* -- its inner workings are encapsulated or hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- Members of a class that are declared with *public visibility* can be referenced anywhere
- Members of a class that are declared with *private visibility* can be referenced only within that class
- Members declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package

Visibility Modifiers

- Public variables violate the spirit of encapsulation because they allow the client to “reach in” and modify the object’s internal values directly
- Therefore, instance variables should not be declared with public 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 it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support or helper method*
- Since a support method is not intended to be called by a client, it should be declared with private - not with public visibility

Visibility Modifiers - Summary

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