Chapter 5: Writing Classes and Enums

CS 121

Department of Computer Science
College of Engineering
Boise State University

November 2, 2015

Chapter 5 Topics

Classes and Objects

- A class is a blueprint of an object. The class represents the *concept* of an object, and any object created from that class is a *realization* (or instantiation) of that concept.

```
Random rand = new Random();
Color springGreen = new Color(107, 235, 128);
Color skyBlue = new Color(135, 206, 235);
```

- An object has state, which is defined by the values of the attributes associated with an object.
  - Example: Attributes for a song: *title*, *artist*, *play time*, *file name*
  - In Java, the attributes are defined by the instance variables declared within the class.
- An object also has behaviors, which are defined by the operations associated with that object.
  - Example: Operations for a song: Set the *title*, find its *play time*, play the song
  - In Java, the operations are defined by the methods declared within the class.

Attributes and Operations

Class	Attributes	Operations
Student	Name Address Major GPA	Set address Set major Compute GPA
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
Account	Name Email Alias Last login date List of session start/end times	Set email Set alias Determine alias Compute time since last login Compute average session length

- **In-class Exercise**: Suggest attributes and operations for classes representing a Sphere, Dog, Calendar Appointment, *Snap*, Smartphone, Computer.

Anatomy of a Class (1)

- A class consists of data declarations and method declarations. These are known as members of the class.

```
public class MyClass
{
    // instance variables
    private int myInstanceVariable;
    private double anotherInstanceVariable;
    private final int MY_CONSTANT = 6;

    // constructor
    public MyClass()
    {
    ...
    }
    // other methods
    public int doYourThing()
    {
    ...
    }
    public String toString()
    {
    ...
    }
}
```

Anatomy of a Class (2)

- Consider a class representing a single dice (a *die*).
  - *State*: Which face is showing on top.
  - *Primary Behavior*: It can be rolled.
- We define it in a class named `Die` with the following methods.

`Die()` Constructor: sets the initial face value to 1
`int roll()` Rolls the die by setting face value to appropriate random number
`void setFaceValue(int value)` Sets the face value to the specified value
`int getFaceValue()` Returns the current face value
`String toString()` Returns a string representation of the Die

- Example: Die.java, SnakeEyes.java

Anatomy of a Class (3)

- Constructor: A constructor has the same name as the class and is used to setup an object when it is created.
- `toString()` method: This method provides a meaningful textual representation of the object.
  - It is called automatically when an object is printed with the `print` or `println` method or concatenated to a string.
  - All classes should provide an appropriate `toString()` method. Useful for debugging and logging.
  - *Recommended style*: Class name followed by attribute name, value pairs. Separate multiple attributes by commas. For example:

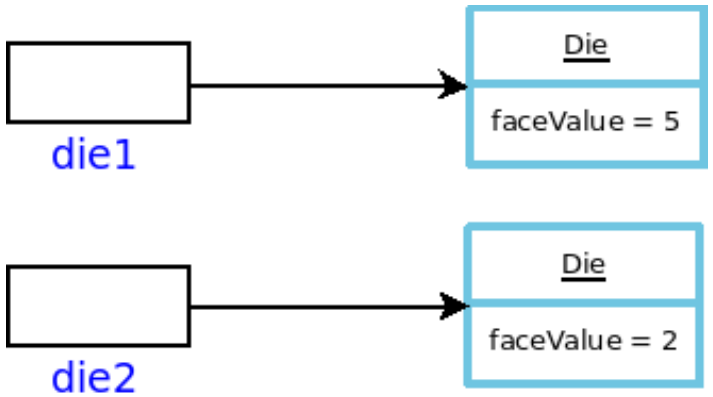
    ```
    Die [faceValue = 1]
    ```

Anatomy of a Class: Data Scope (4)

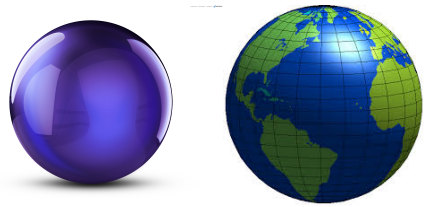
- The scope of the data is the area in the program where it can be referenced and used.
- Instance variables: These are variables declared at the class level, that is, outside of any methods. Typically, they are declared at the top of the class declaration. These can be referenced by all the methods in the class.
- Local variables: Local variables are declared inside a method (within its defining curly braces) and can only be used in that method.

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
- The objects of a class share the method definitions but each object has its own data space for its instance variables. That's how different objects of the same class can have different states.
- The two `Die` objects from the SnakeEyes.java program. Each object has its own `faceValue` variable.



In-Class Exercise



- **PP 5.4**. Implement a class named `Sphere` that contains the instance data that represents the sphere's radius.
  - Include get/set methods for the radius.
  - Include methods to calculate the surface area and volume.
  - Include a `toString` method that returns a one-line description of the sphere.
  - Write a driver class called `MultiSphere` that instantiates and updates several `Sphere` objects.

Encapsulation (1)

- Two views of an object:
  - *internal*: the details of the variables and methods that make up a class
  - *external*: the services that an object provides and how the objects interacts with the rest of the system
- From the external view, an object is an encapsulated entity, providing a set of specific services These services define the interface to the object.
- One object (called the client) may use another object for the services it provides.
- Clients should not be able to access an object's variables directly. Any changes to the object's state should be made by that object's methods. That is, an object should be self-governing.

Visibility Modifiers (1)

- Encapsulation is provided via the use of visibility modifiers. A modifier is a Java reserved keyword that specifies particular characteristics of a variable or method. Example: `final`
- Java provides three visibility modifiers: `public`, `protected`, `private`.
- Instance variables and methods of a class declared `public` can be referenced anywhere.
- Instance variables and methods of a class declared `private` can be referenced only within that class.
- Instance variables and methods declared without a visibility modifier have default visibility and can be referenced by any class in the same package.
- Instance variables and methods of a class declared `protected` can be referenced by any class in the same package as well as any subclass in any package. We will study this in CS 221.

Visibility Modifiers (3)

- To enforce encapsulation, instance variables should always be declared with `private` visibility.
  - Then we can allow controlled access using special methods know as *accessors* and *mutators*.
  - An accessor method returns the current value of an instance variable.
  - A mutator method changes the value of an instance variable.
  - They are named as `getX` and `setX`, where `X` is the name of the instance variable. Hence they are also referred to as getters and setters.
- It is acceptable to give a constant instance variable `public` visibility. (Why?)
- Methods that provide service to clients should be declared `public`.
- A method created simply to assist a service method is called a support method. Support methods are declared `private`.

Visibility Modifiers (4)

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

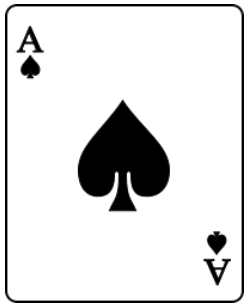
In-Class Exercise

- What is the relationship between a class and an object?
- Where are instance variables declared?
- What is the scope of instance variables?
- What is a local variable?

Examples

- Create a class to represent a coin that can be flipped for heads or tail.
  - Coin.java, CountFlips.java
  - FlipRace.java
- Create a n-sided die version of the `Die` class.
  - NSidedDie.java, RollingNSidedDie.java

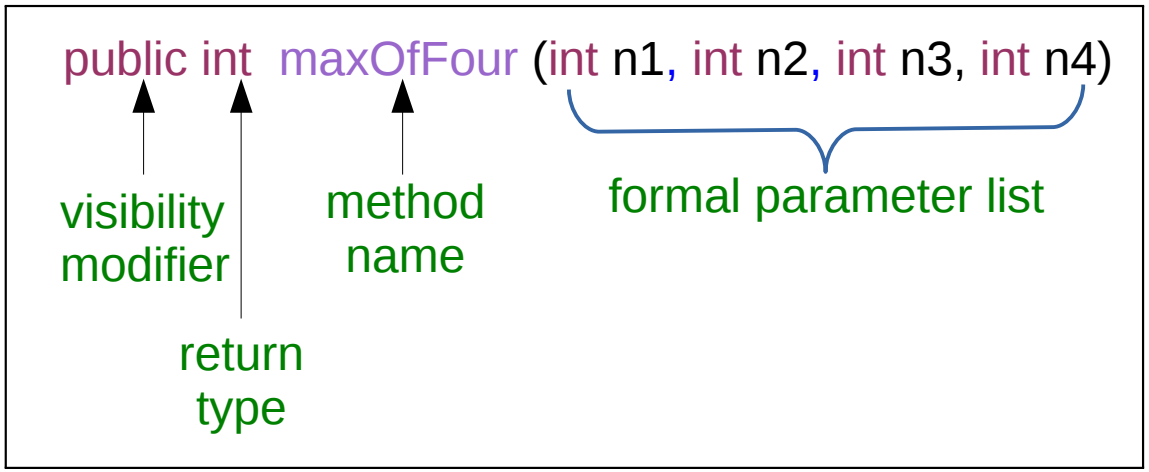
In-Class Exercise



- Design a class to represent a single playing card.
  - What instance data do you need?
  - What would your constructor look like?
  - What card operations (methods) do you want to provide?
- Let's write the class.

Anatomy of a Method (1)

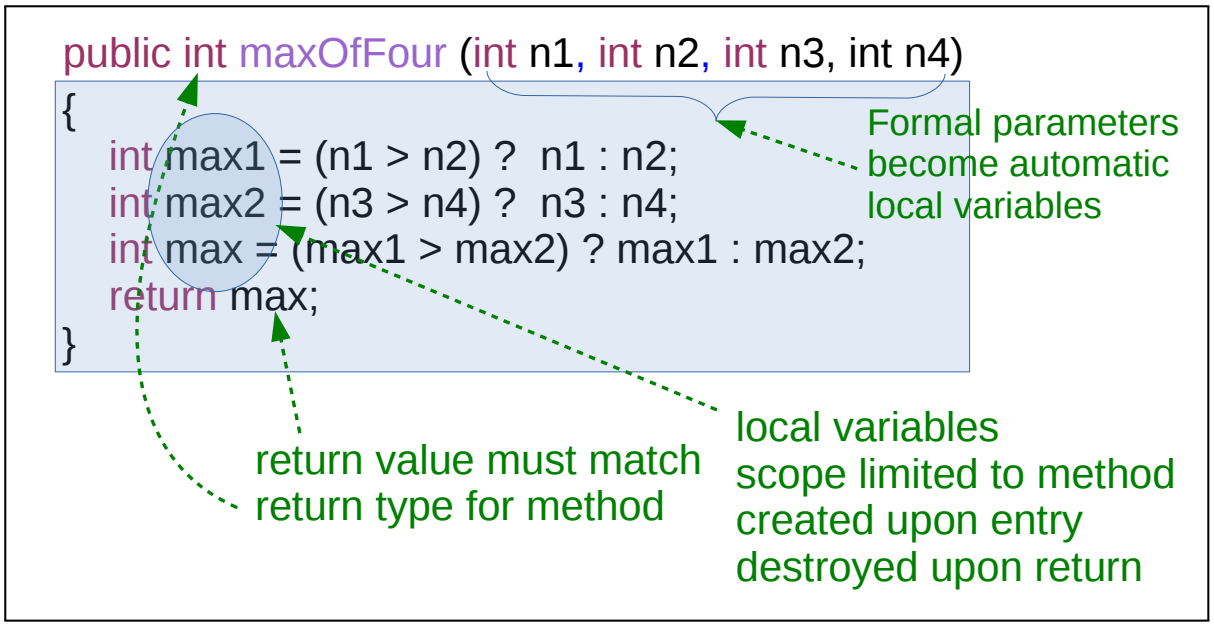
- A method consists of a header and a body
- The format of a method header:



- The parameter list specifies the type and name of each parameter.
- The name of a parameter in the method header is called a formal parameter

Anatomy of a Method (2)

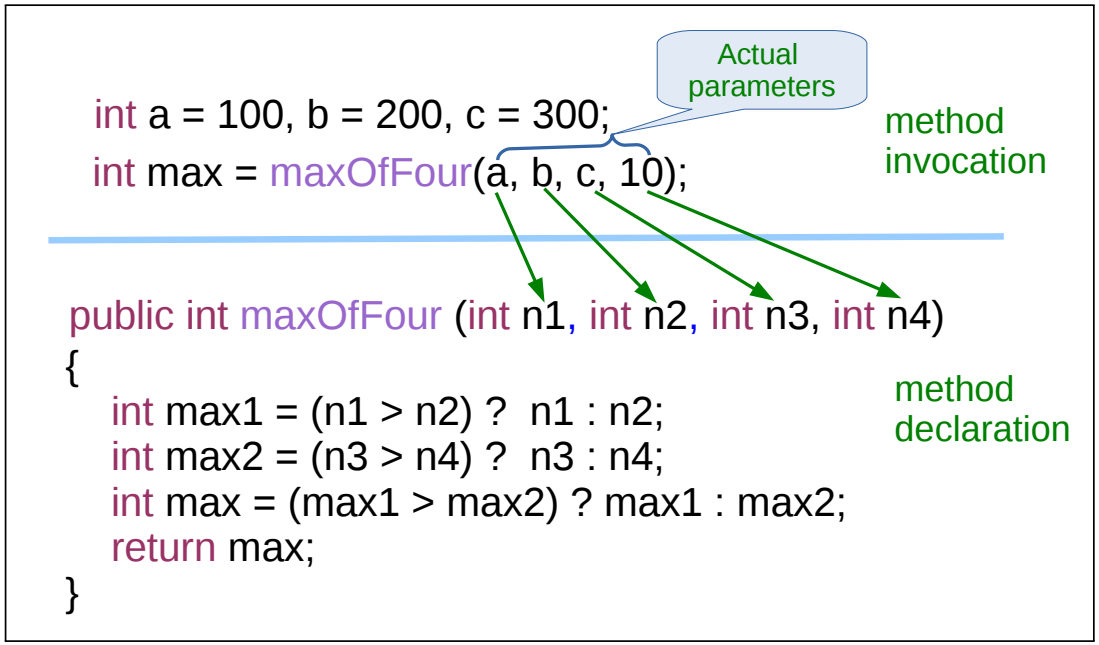
- The method body follows the header and is enclosed within *curly brackets*



- A `return` statement specifies the value that will be returned. A method can have multiple `return` statements.
- A method that does not return a value has `void` return type in the header. Such methods modify the state of an object.
  - For example: A *setter* (*mutator*) method is typically void since it modifies an instance variables but need not return any value back.

Method Invocation (1)

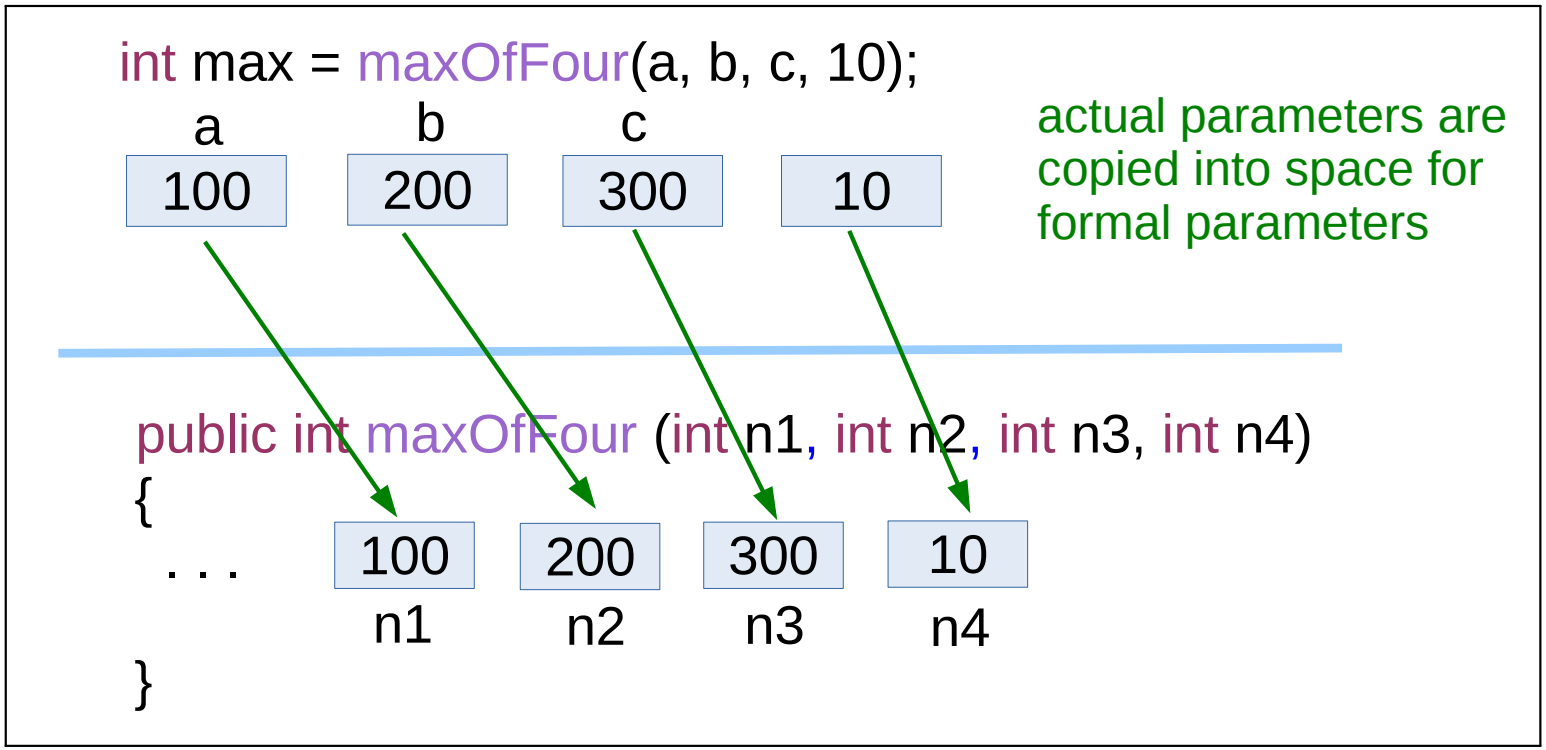
- When a method is called, actual parameters are copied into the formal parameters.



- Note that methods are contained inside classes so we would have to include reference to an object variable above to invoke the `maxOfFour` method, but we skip that for simplicity in this section.
- When a method finishes, the local variables (including the formal parameters) are destroyed and the memory is reclaimed.
- However, the instance variables, declared at the class level, remain in existence as long as the object exists.

Method Invocation (2)

- Note that method calls pass parameters by copying. This is known as call-by-value.
- See below for the memory model for method invocation.



Method Examples (1)

- **Ex 5.7**. Write a method called `cube` that accepts one integer parameter and returns that value raised to the third power.

```
public int cube(int n)
{
    return n * n * n;
}
```

- **Ex 5.9** Write a method named `randomInRange` that accepts two integer parameters representing a range and returns a random integer in the range (inclusive). Assume that the range is valid.

```
public int randomInRange(int low, int high)
{
    Random rand = new Random();
    int value = rand.nextInt(high - low + 1) + low;
    return value;
}
```

Method Examples (2)

- **Ex 5.18**. Write a method name `isAlpha` that accepts a character parameter and returns true if the character is either an uppercase or lowercase alphabetic letter.

```
public boolean isAlpha(char ch)
{
    if ('a' <= ch && ch <= 'z')
        return true;
    else if ('A' < = ch && ch <= 'Z')
        return true;

    return false;
}
```

Method Examples (3)

- Write a `public` method named `poemOfTheMomemt` that takes no arguments, returns nothing but prints a poem of your choosing on the console.

```
public void poemOfTheMoment() {
  System.out.println("========================================")
  System.out.println("\"Like a piece of ice on a hot stove,");
  System.out.println("the poem must ride on its own melting.\"");
  System.out.println("\t--Robert Frost");
  System.out.println("========================================")
}
```

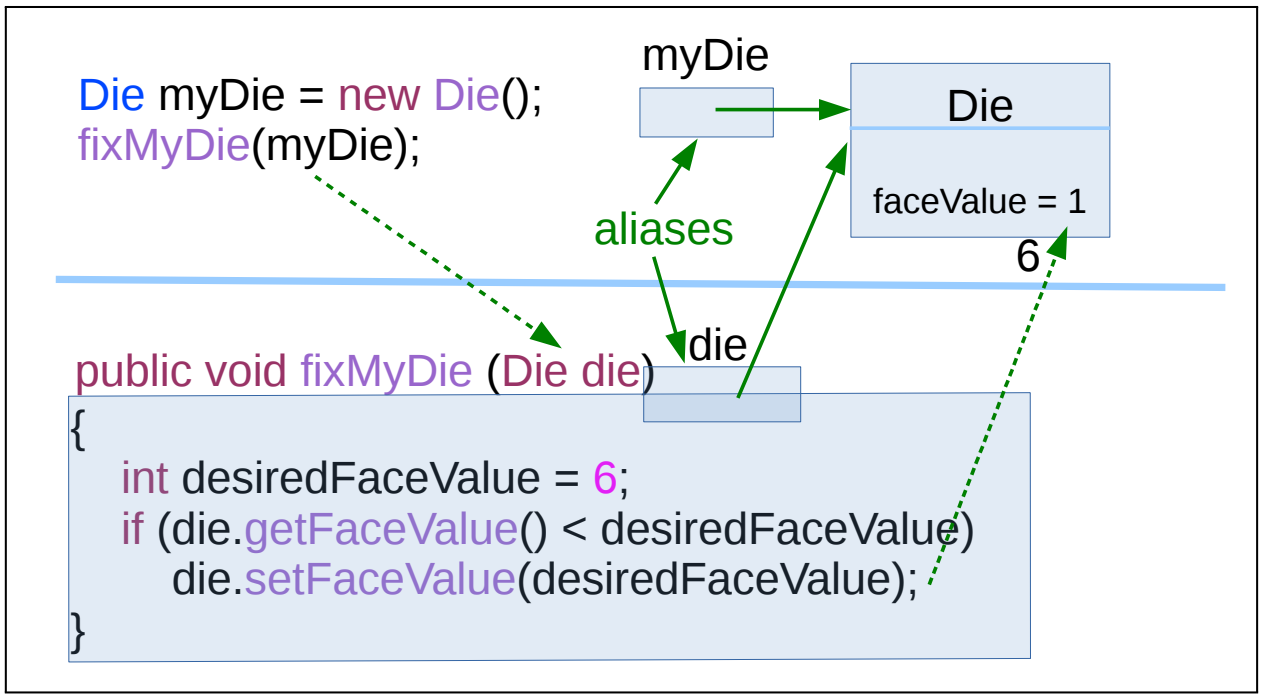
- Write a `public` method that displays a line of "=" symbols *n* times, where n is specified by the caller.

```
public void printSeparator(int n)
{
    for (int i = 0; i < n; i++)
        System.out.print("=");
    System.out.println();
}
```

- **In-class Exercise**. Rewrite the method `poemOfTheMoment` using the `printSeparator` method.
- **In-class Exercise**. Rewrite the `printSeparator` so that it takes another `char` argument, which is the character to use in the separator line.

Passing Objects as Parameters

- Passing objects as parameters copies the reference/address making the corresponding formal parameter an *alias*.
- This implies that changes made to an object inside an method change the original object.
- See below for an example:



- Example: ParameterTester.java, ParameterModifier.java, Num.java

In-class Exercise

- Write a method that adds two `Color` objects together and returns a new `Color` object.
- Let's figure out the method header first.
- Now we will complete the method body.
- What if adding the colors makes the values be outside the range $[0, 255]$?
- **Additional Exercise**. Write a method that creates and returns a random `Color` object.

Example - Account

- Consider an `Account` class:
  - Attributes: *name*, *account number* and *balance*
  - Operations (services): *withdrawals*, *deposits* and *adding interest*.
- Example: Account.java. Transactions.java
- Draw the three objects and show their state at the end of the Transactions.java driver program.

The this Reference

- The this reference allows an object to refer to itself. That is, the this reference, used inside a method, refers to the object through which the method is being executed.
- The this reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names.
- Thus we can reuse the instance variable names as formal parameter names, avoiding having to create unnecessary new names.

```
public Account (String name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Static Methods and Static Variables

- Static methods and variables are declared using the `static` modifier. It associates the method or variable with the class rather than with an object of that class.
- Hence, static method and static variables are also known as class methods and class variables.
- Static methods are invoked using the name of the class. For example:

```
double x = Math.sqrt(2.0);
```

Static Variables

- If a variable is declared as static, only one copy of the variable exists.
- Memory space for a static variable is created when the class is first referenced.
- All objects instantiated from the class share its static variables.
- Changing the value of a static variable in one object changes it for all others.

Static Methods

- Static methods cannot refer to instance variables because instance variables don't exist until an object exists. They cannot call non-static methods as well as no object exists to call them on.
- Static methods can refer to static variables or local variables. Static methods often work together with static variables.
- Example: Slogan.java, SloganCounter.java
- Determining if a variable or method should be static is a design decision.

Method Overloading (1)

- Method overloading is the process of giving a single method name multiple definitions.
- The signature of a method includes the number, type, and order of the parameters. The return type isn't part of the signature.
- The signature of each overloaded method must be unique.

```
double tryMe(int x)
{
    return x + 2.0;
}
double tryMe(int x, double y)
{
    return x * y;
}
```

Method Overloading (2)

- Example: The `println` method is overloaded.

```
println(String s)
println(int i)
println(double x)
...
```

- The following code snippet uses multiple versions of println.

```
System.out.println("A String!");
System.out.println(1000);
```

- Constructors can also be overloaded. Overloaded constructors provide multiple ways to initialize a new object. For example:

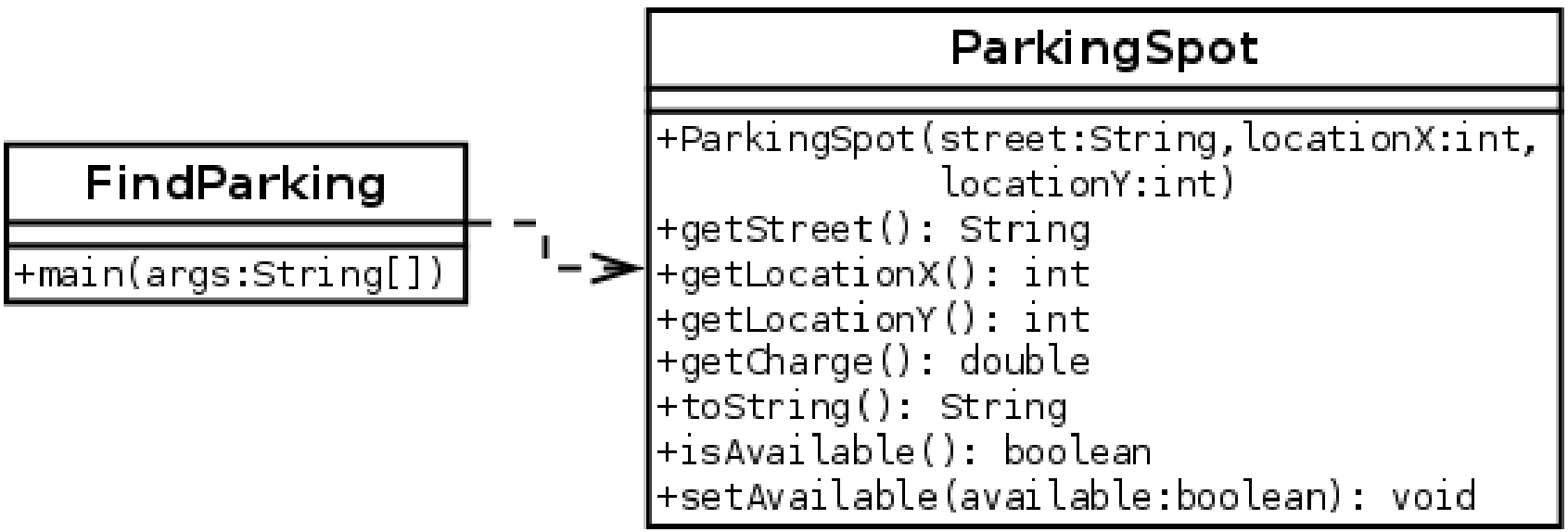
```
NSidedDie die1 = new NSidedDie();
NSidedDie die2 = new NSidedDie(20);
```

Class Relationships

- Classes can be related in three common ways:
  - Dependency – `A` *uses* `B`.
  - Aggregation – `A` *has-a* `B`.
  - Inheritance – `A` *is-a* `B`. (This is covered in CS 221)

Dependency (*uses*) Relationship

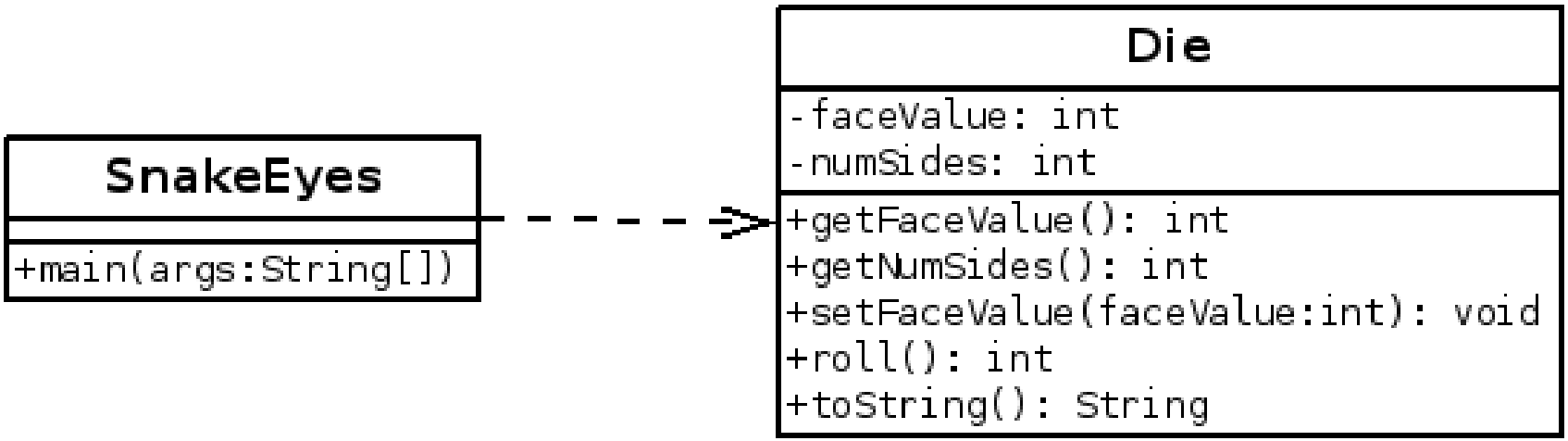
- A dependency exists when one class relies on the other in some way. Usually by invoking the methods of the other.
- For example:
  - `FindParking` *uses* `ParkingSpot`.
  - `SnakeEyes` *uses* `Die`.



UML Diagrams

- UML stands for the Unified Modeling Language
- UML diagrams show relationships among classes and objects.
- A UML class diagram consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent associations.
- A solid arrow shows that one class uses the other (calls its methods)

UML Diagrams



UML Class Diagrams Classes are drawn as rectangles, which may be divided into 1, 2 or 3 partitions. The top partition is for the class name, the second one for the instance variables, and the third one for the methods or operations. Each variable/method is preceded by a *visibility indicator*.

- \+ indicates public
- \- indicates private
- # indicates protected

Self Dependencies

- Dependencies can also exist between objects of the same class.
- For example, a method of a class may accept an object of the same class as a parameter.

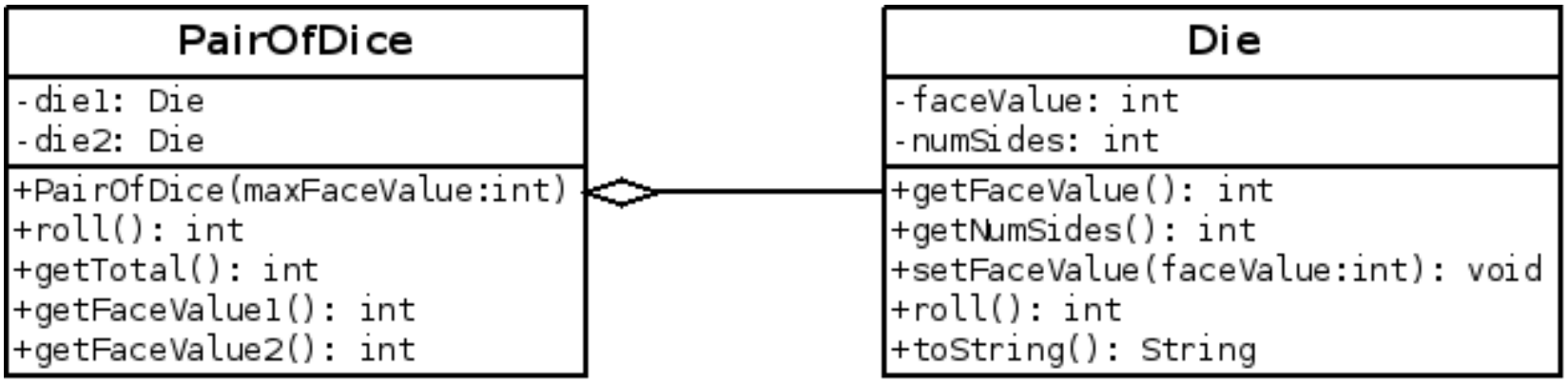
```
String fullName = firstName.concat(lastName);
```

In-Class Exercise

- Recall: A Rational Number is a real number that can be written as a simple fraction (i.e. as a ratio).
  - $\frac{3}{2}$, $\frac{5}{8}$, etc.
- Think for a minute...how can we represent a rational number in Java?
- What types of operations do we want to be able to do?
- Example: RationalNumber.java
- Several of the methods in `RationalNumber` depend on another `RationalNumber` object.

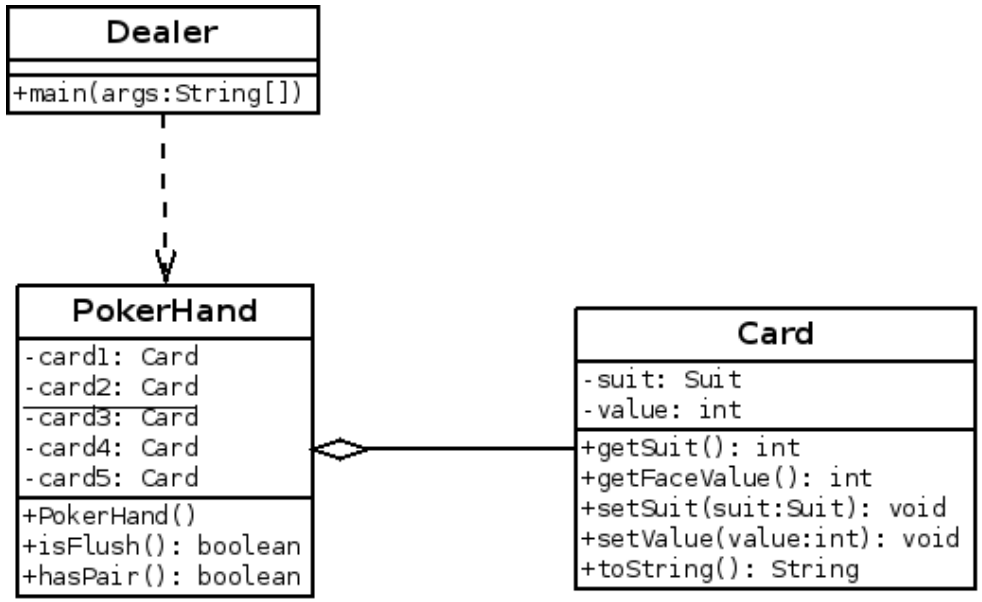
Aggregation (*has-a*) Relationship

- An aggregate object is an object that is made up of other objects.
- For example:
  - A `PairOfDice` *has-a* `Die`.
- An aggregate object contains references to other objects as *instance data*.



Aggregation (*has-a*) Relationship

- A `PokerHand` *has-a* `Card`.
- A `Dealer` class *depends on* the `PokerHand`.



Method Design

- Program design involves two levels:
  - *High-level*: Identify primary classes and objects. Assign primary responsibilities.
  - *Low-level*: Decompose classes into methods.
- An algorithm is a step-by-step process for solving a problem.
  - An algorithm may be expressed in pseudocode, a mixture of code statements and English that communicate the steps required.
  - A method implements an algorithm that determines how the method accomplishes its goals.

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity.
- A potentially large method should be decomposed into several smaller methods as needed for clarity.
- A public service method of an object may call one or more private support methods to help it accomplish its goal.
- Support methods might call other support methods if appropriate.

Extended Example: PigLatin (1)

- Objective: Write a program that translates English into Pig Latin.
- Pig Latin is a language in which each word is modified as follows:
  - Words that begin with a vowel simply have the "yay" sound added to the end.
  - Words that begin with a consonant blends such as "ch" "st" are moved together to the end before adding the "ay" sound to the end.
  - Words that begin with a single consonant have the initial sound moved to the end before adding the "ay" sound at the end.
- Examples:

  `book` $\rightarrow$ `ookbay`

  `item` $\rightarrow$ `itemyay`

  `chair` $\rightarrow$ `airchay`

Extended Example: PigLatin (2)

- The primary objective (translating a sentence) is too complicated for one method to accomplish. Therefore we look for natural ways to decompose the solution into pieces
- Translating a sentence can be decomposed into the process of translating each word.
- The process of translating a word can be separated into translating words that
  - begin with vowels
  - begin with consonant blends (sh, cr, th, etc.)
  - begin with single consonants
- Example: PigLatin.java, PigLatinTranslator.java

Enumerated Types (1)

- An Enumerated type is a set of values or elements that behave as constants. For example:

```
public enum Season {WINTER, SPRING, SUMMER, FALL};
public enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- The enumeration lists all possible values for the specified type.
- Now we can declare a variable of the enumerated type and assign it values:

```
Suit cardSuit = Suit.DIAMONDS;
```

- Enumerated values are type-safe, that is, only valid assignments are the listed values.
- **In-Class Exercise: EX 3.12**. Write a declaration for an enumerated type that represents the days of the week. Declare a variable of this type and assign it a day of the week.

Enumerated Types(2)

- Internally, each value of an enumerated type is stored as an integer, called its ordinal value. We can access the ordinal value with the `ordinal()` method.

```
System.out.println(cardSuit.ordinal());
```

- The first value in an enumerated type has an ordinal value of zero, the second an ordinal value of one, and so on.
- However, we cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value.
- Example: SuitTest.java, IceCream.java, Navigator.java
- Example: FavoriteColorsByOrdinal.java

Exercises

- Read Chapter 5.
- **Recommended Homework**:
  - Exercises: EX 5.8, 5.10, 5.11, 5.12, 5.14, 5.21, 5.25, 5.26, 5.32.
  - Projects: PP 5.1, 5.7, 5.11 (more difficult), 5.12.
- Browse: Sections 7.1–7.3.