Java classes

Outline

- Objects, classes, and object-oriented programming
  - relationship between classes and objects
  - abstraction
- Anatomy of a class
  - instance variables
  - instance methods
  - constructors

Objects and classes

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Writing programs that perform most of their behavior as interactions between objects.
- **class**: 1. A program. or,
  2. A blueprint of an object.
  - classes you may have used so far:

    `String`, `Scanner`, `File`
- We can write classes to define new types of objects.

Abstraction

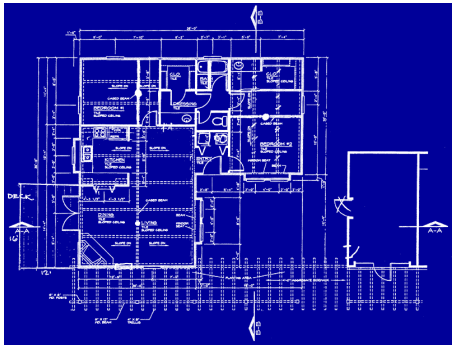
- **abstraction**: A distancing between ideas and details.
  - Objects in Java provide abstraction:
    We can use them without knowing how they work.
- You use abstraction every day.

  Example: Your portable music player.
  - You understand its external behavior (buttons, screen, etc.)
  - You don't understand its inner details (and you don't need to).

Blueprint analogy

Music player blueprint

state:
current song
volume
battery life
behavior:
power on/off
change station/song
change volume
choose random song



creates

Music player #1

state:
song = "Thriller"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Music player #2

state:
song = "Sandstorm"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Music player #3

state:
song = "Code Monkey"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

How often would you expect to get snake eyes?

If you're unsure on how to compute the probability then you write a program that simulates the process

Snake Eyes

```
public class SnakeEyes {
   public static void main(String [] args){
      int ROLLS = 10000;
      int count = 0;
      Die die1 = new Die();
      Die die2 = new Die();
      for (int i = 0; i < ROLLS; i++){
          if (die1.roll() == 1 && die2.roll() == 1){
             count++;
          }
      }
      System.out.println("snake eyes count: " + count);
    }
}
```

Need to write the Die class!

Die object

- State (data) of a `Die` object:

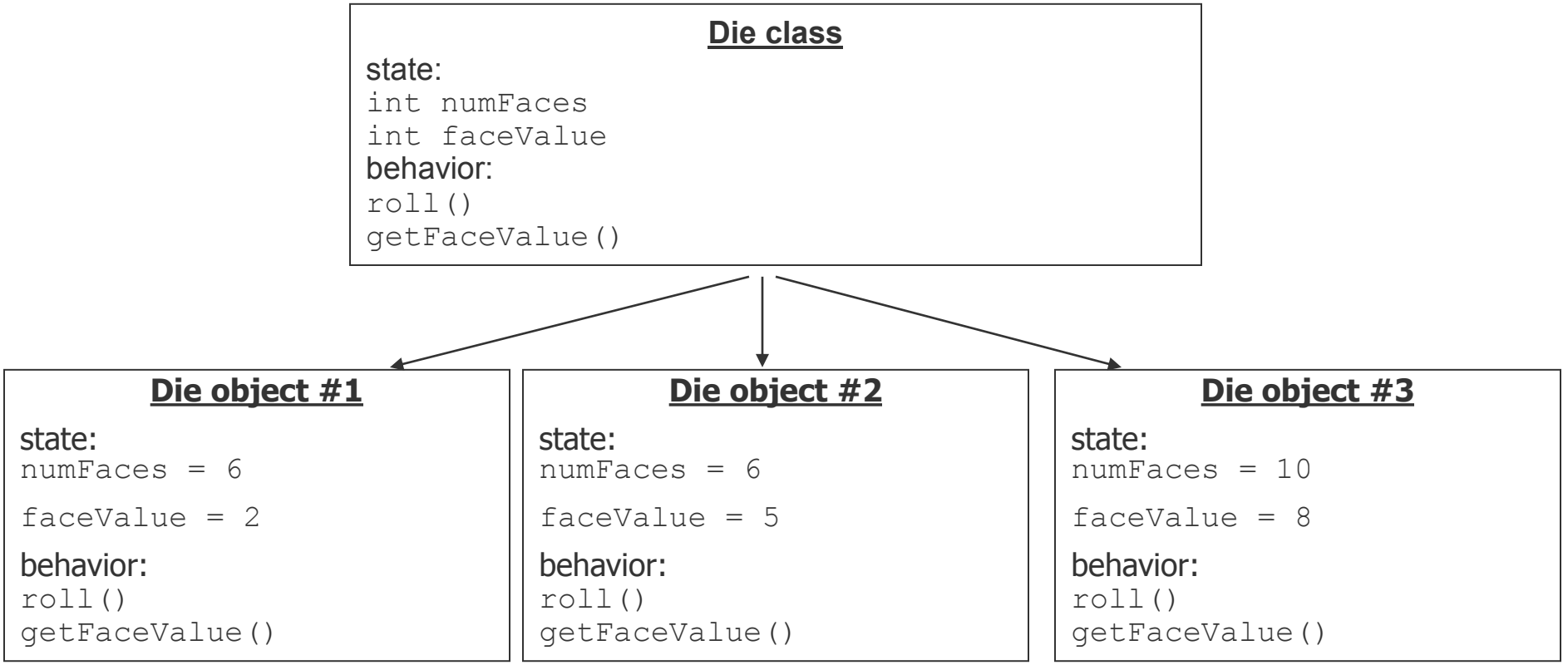
Instance variable	Description
`numFaces`	the number of faces for a die
`faceValue`	the current value produced by rolling the die

- Behavior (methods) of a `Die` object:

Method name	Description
`roll()`	roll the die
`getFaceValue()`	retrieve the value of the last roll

The Die class

- The class (blueprint) knows how to create objects.




```
Die die1 = new Die();
```

Object state: instance variables

Die class, version 1

- The following code creates a new class named `Die`.

```
public class Die {
    int numFaces;
    int faceValue;
}
```

declared outside of any method

  - Save this code into a file named `Die.java`.
- Each `Die` object contains two pieces of data:
  - an `int` named `numFaces`,
  - an `int` named `faceValue`
- No behavior (yet).

Instance variables

- **instance variable**: A variable inside an object that holds part of its state.
  - Each object has ***its own copy*.**
- Declaring an instance variable:

  <type> <name> ;

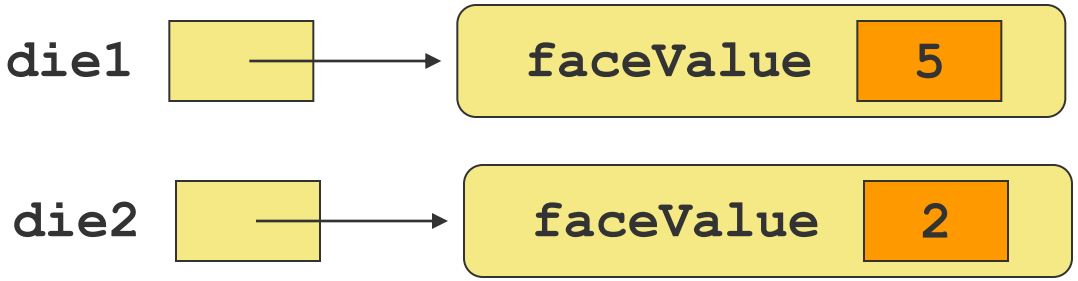
  - Examples:

```
public class Student {
  String name; //Student object has a name
  double gpa;  //and a gpa
}
```

Instance variables

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

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



Accessing instance variables

- Code in other classes can access your object's instance variables.
  - Accessing an instance variable:

    <variable name> **.** ***<instance variable>***

  - Modifying an instance variable:

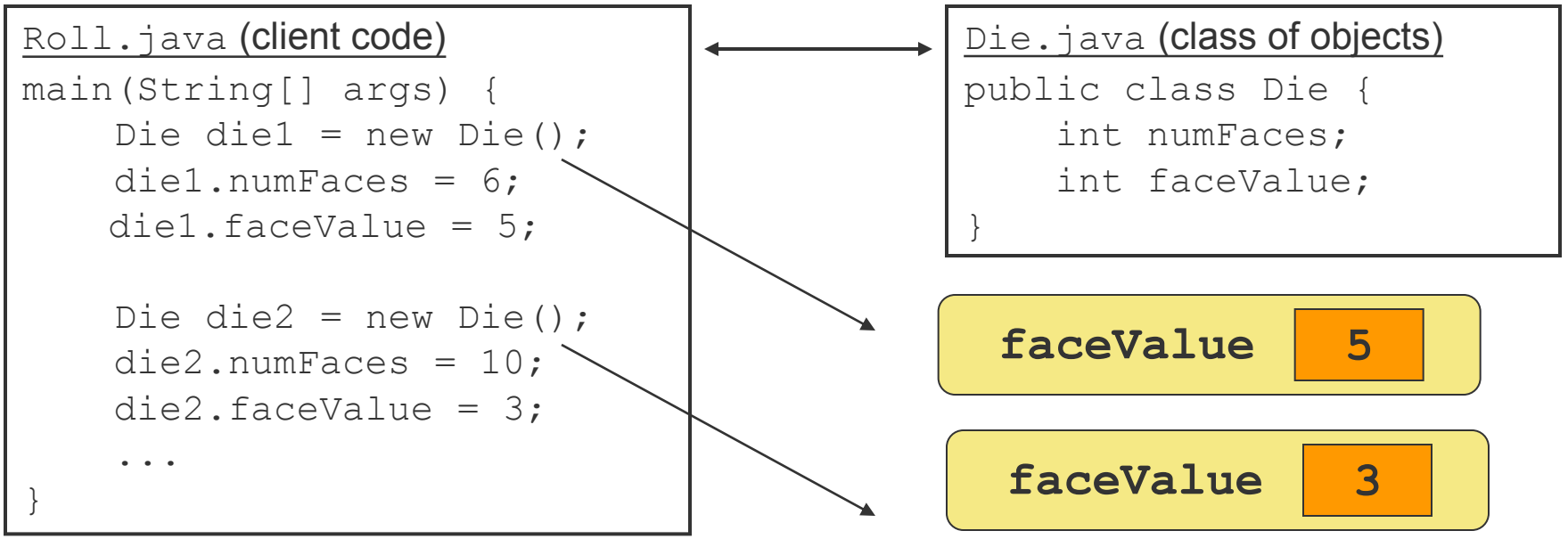
    <variable name> **.** ***<instance variable>*** = ***<value>*** ***;***

- Examples:

```
System.out.println("you rolled " + die.faceValue);
die.faceValue = 20;
```

Client code

- `Die.java` is not, by itself, a runnable program.
  - Can be used by other programs stored in separate `.java` files.
- **client code**: Code that uses a class.
  - Driver program – used for testing a class (type of client)



Object behavior: methods

Procedural vs OO methods

- Procedural emphasizes action (static)
  - *When is your birthday, Chris?*

    birthday(Chris)
  - *Stand up, Chris*

    stand(Chris)
- OO emphasizes object (non static)
  - *Chris, when is your birthday?*

    Chris.birthday()
  - *Chris, stand up*

    Chris.stand()

}

Getting the dice rolling – procedural

```
public class SnakeEyes {
   public static void main(String [] args){
      int ROLLS = 10000;
      int count = 0;
      Die die1 = new Die();
      Die die2 = new Die();
      for (int i = 0; i < ROLLS; i++){
          if (roll(die1) == 1 && roll(die2) == 1){
             count++;
      }
   …
   public static int roll(Die die) {
       return (int) (Math.random() * die.numFaces) + 1;
   }
}
```

Problems with the procedural solution

- The procedural method solution isn't fitting the Object Oriented nature of Java
  - The syntax doesn't match the way we're used to using objects.

```
int value = roll(die);
```

Roll is in SnakeEyes even though it is a Die operation. In an Object Oriented program roll belongs in Die.

- The point of classes is to combine state and behavior.
  - **`roll` belongs in the `Die` object.**

```
int value = die.roll();
```

OO Instance methods

- **instance method**:
  One that defines behavior for each object of a class.

- instance method declaration, general syntax:

```
public <type> <name> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

Getting the dice rolling – using OO instance methods

```
public class Die {
    int numFaces;
    int faceValue;

    public int roll (){
        faceValue = (int)(Math.random() * numFaces) + 1;
        return faceValue;
    }
}
```

```
Die die1 = new Die();
die1.numFaces = 6;
int value1 = die1.roll();
Die die2 = new Die();
die2.numFaces = 10;
int value2 = die.roll();
```

Think of each `Die` object as having its own copy of the `roll` method, which operates on that object's state

Object initialization: constructors

Initializing objects

- It is tedious to construct an object and assign values to all of its instance variables one by one.

```
Die die = new Die();
die.numFaces = 6;   //tedious
```

- We'd rather pass the instance variables' initial values as parameters:

```
Die die = new Die(6);  // better!
```

Constructors

- **constructor**: creates and initializes a new object
  - Constructor syntax:

```
public <type> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

  - The <type> is the name of the class
  - A constructor runs when the client uses the `new` keyword.
  - A constructor implicitly returns the newly created and initialized object.
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all the object's fields to 0 or null.

Die constructor

```
public class Die {
    int numFaces;
    int faceValue;

    public Die (int faces) {
        numFaces = faces;
        faceValue = 1;
    }

    public int roll (){
        faceValue = (int)(Math.random() * numFaces) + 1;
        return faceValue;
    }
}
```

```
Die die1 = new Die(6);
```

Multiple constructors are possible

```
public class Die {
    int numFaces;
    int faceValue;

    public Die () {
        numFaces = 6;
        faceValue = 1;
    }

    public Die (int faces) {
        numFaces = faces;
        faceValue = 1;
    }
}
```

```
Die die1 = new Die(6);
Die die2 = new Die();
```

Encapsulation

Encapsulation

- **encapsulation**:
  Hiding implementation details of an object from clients.
- Encapsulation provides *abstraction*;
  we can use objects without knowing how they work.

  The object has:
  - an **external view** (its behavior)
  - an **internal view** (the state that accomplishes the behavior)

Implementing encapsulation

- Instance variables can be declared ***private*** to indicate that no code outside their own class can access or change them.
  - Declaring a private instance variable:

    `private` ***<type> <name>*** `;`
  - Examples:

```
private int faceValue;
private String name;
```

- Once instance variables are private, client code cannot access them:

```
Roll.java:11: faceValue has private access in Die
System.out.println("faceValue is " + die.faceValue);
                                          ^
```

Instance variables encapsulation and access

- In our initial implementation of the Die class we didn't use access modifiers. This is the same as using the public access modifier:

```
public class Die {
   public int numFaces;
   public int faceValue;
}
```

- We can encapsulate the instance variables using private:

```
public class Die {
   private int numFaces;
   private int faceValue;
}
```

But how does a client class now get to these?

Accessors and mutators

- We provide accessor methods to examine their values:

```
public int getFaceValue() {
    return faceValue;
}
```

  - This gives clients read-only access to the object's fields.
- **If so desired**, we can also provide mutator methods:

```
public void setFaceValue(int value) {
    faceValue = value;
}
```

Mostly not needed, a roll method in Die should do this

  - Client code will look like this:

```
System.out.println("faceValue is " + die.getFaceValue());
```

Benefits of encapsulation

- Protects an object from unwanted access by clients.
  - Example: If we write a program to manage users' bank accounts, we don't want a malicious client program to be able to arbitrarily change a `BankAccount` object's balance.

- Allows you to change the class implementation later.

- As a general rule, all instance data should be modified only by the object, i.e. **instance variables should be declared private**

Printing Objects

- Java's default method of printing objects:

```
Account acct = new Account(…);
System.out.println("acct: " + acct);
// result: acct: Account@9e8c34
```

- We ***could*** give Account a print method that gives a more informative result:

```
acct.print();
```

- ***But***, Java gives us a better mechanism using the toString() method

The `toString()` method

- tells Java how to convert an object into a `String`
- called when an object is printed/concatenated to a `String`:

```
Point p = new Point(7, 2);
System.out.println("p: " + p);
```

  - Same as:

```
System.out.println("p: " + p.toString());
```

- Every class has a `toString()`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:

```
Point@9e8c34
```

toString() syntax

```
public String toString() {
        code that returns a suitable String;
}
```

- The method name, return, parameters must match exactly.
- Example:

```
public String toString(){
   return "Account number " + accountNumber + "\n"
     + "Name " + name + "\n"
     + "Balance " + balance + "\n"
     + "interest rate " + interestRate;
}
```

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - During the call `die1.roll();`,
    the object referred to by `die1` is the implicit parameter.
  - The instance method can refer to that object's instance variables.
  - The implicit parameter has the name **`this`**
  - The method `int roll()` is really `int roll(Die this)`
  - The call `die1.roll()` is translated to `roll(die1)`

Use of `this`

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method is called
- Syntax for using `this`:
  - To refer to an instance variable (optional):
    `this.`**`variable`**
  - To call a method (optional):
    `this.`**`method`**`(`**`parameters`**`);`
  - To call a constructor from another constructor:
    `this(`**`parameters`**`);`

Variable shadowing

- An instance method parameter can have the same name as one of the object's instance variables:

```
public class Loc {
  private int x;
  private int y;
     …
  // this is legal
  public void setLocation(int x, int y) {
    // when using x and y you get the parameters
  }
```

  - Instance variables `x` and `y` are *shadowed* by parameters with same names.

Avoiding shadowing using `this`

```
public class Loc{
    private int x;
    private int y;
    ...
    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- Inside the `setLocation` method,
  - When `this.x` is seen, the *instance variable* `x` is used.
  - When `x` is seen, the *parameter* `x` is used.

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```
public class Loc {
    private int x;
    private int y;

    public Loc () {
        x = 0;
        y = 0;
    }

    public Loc(int x, int y) {
        this.x = y;
        this.y = y;
    }

    ...
}
```

Constructors and `this`

- One constructor can call another using `this`:

```
public class Loc {
    private int x;
    private int y;

    public Loc() {
        this(0, 0);  //calls the (x, y) constructor
    }

    public Loc(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

Method overloading

- Can you write different methods that have the same name?
- Yes! We have already done it:

```
System.out.println("I can handle strings");
System.out.println(2 + 2);
System.out.println(3.14);
System.out.println(object);
Math.max(10, 15);          // returns integer
Math.max(10.0, 15.0);      // returns double
```

Useful when you need to perform the same operation on different kinds of data.

Method overloading

```
public int sum(int num1, int num2){
    return num1 + num2;
}
public int sum(int num1, int num2, int num3){
    return num1 + num2 + num3;
}
```

- A method's name + number, type, and order of its parameters: **method signature**
- The compiler uses a method's signature to bind a method invocation to the appropriate definition

The return value is not part of the signature

- You ***cannot*** overload on the basis of the return type (because it can be ignored)

Example:

```
public int convert(int value) {
  return 2 * value;
}

public double convert(int value) {
  return 2.54 * value;
}
```

Example

- Consider the class Pet

```
class Pet {
   private String name;
   private int age;
   private double weight;

…
}
```

Example (cont)

```
public Pet()
public Pet(String name, int age, double weight)
public Pet(int age)
public Pet(double weight)
```

Suppose you have a horse that weights 750 pounds then you use:

```
Pet myHorse = new Pet(750.0);
```

but what happens if you do:

```
Pet myHorse = new Pet(750);  ?
```

Primitive Equality

- Suppose we have two integers `i` and `j`
- How does the statement `i==j` behave?
- `i==j` if `i` and `j` contain the same value

Object Equality

- Suppose we have two pet instances `pet1` and `pet2`
- How does the statement `pet1==pet2` behave?
- Just like for primitives!
- `pet1==pet2` is true if ***both*** refer to the ***same*** object

Object Equality - extended

- The == operator checks if the ***addresses*** of the two objects are equal
- If you want a different notion of equality define your own `.equals()` method.
- Do `pet1.equals(pet2)` instead of `pet1==pet2`
- The default definition of `.equals()` is the value of ==

.equals for the Pet class

```
public boolean equals (Pet other) {
   return ((this.age == other.age)
     &&(Math.abs(this.weight - other.weight) < 1e-8)
     &&(this.name.equals(other.name)));
}
```

== vs .equals() - again

```
String helloWorld = “HelloWorld”
String hello = “Hello”;
String world = “World”;
String hw = hello + world;

is helloWorld == hw ?

is helloWorld.equals(hw) true/false?

Let's play with the Equals.java program...
```

Object Equality – are they .equal() ?

Summary: Access Protection

Access protection has three main benefits:

- It allows you to enforce constraints on an object's state.
- It provides a simpler client interface. Client programmers don't need to know everything that’s in the class, only the public parts.
- It separates interface from implementation, allowing them to vary independently.

General guidelines

As a rule of thumb:

- Classes are public.
- Instance variables are private.
- Constructors are public.
- Getter and setter methods are public (unless…)
- Other methods must be decided on a case-by-case basis.

Naming things

- Computer programs are written to be read by humans and only incidentally by computers.
- Use names that convey meaning
- Loop indices are often a single character (i, j, k), but others should be more informative.
- Importance of a name depends on its scope.
- Names with a “short life” need not be as informative as those with a “long life”
- Read code and see how others do it