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

Music player #1

- **State**
  - Song: "Thriller"
  - Volume: 15
  - Battery: 2.5 hrs
- **Behavior**
  - Power on/off
  - Change station/song
  - Change volume
  - Choose random song

Music player #2

- **State**
  - Song: "Sandstorm"
  - Volume: 15
  - Battery: 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: 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

```java
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
    - Name: `numFaces`
      - Description: the number of faces for a die
    - Name: `faceValue`
      - Description: the current value produced by rolling the die
  
- **Behavior (methods) of a Die object:**
  - Method name
    - `roll()`
      - Description: roll the die
    - `getFaceValue()`
      - Description: retrieve the value of the last roll

## The Die class

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

  ```java
  public class Die {
      int numFaces;
      int faceValue;
      // Other behaviors...
  }
  ```

- Save this code into a file named `Die.java`.

## Object state:

### 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:
    ```java
    <type> <name>;
    ```
  - Examples:
    ```java
    public class Student {
        String name;  // Student object has a
        double gpa;  // name and gpa
    }
    ```
**Instance variables**

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

```java
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:**

```java
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)

```java
public class Roll {
    public static void main(String[] args) {
        Die die1 = new Die();
        die1.numFaces = 6;
        die2.faceValue = 5;
        Die die2 = new Die();
        die2.numFaces = 10;
        die2.faceValue = 3;
        ...
    }
}
```

**Object behavior: methods**

---

**Getting the dice rolling – solution 1**

```java
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 static solution**

- The static method solution isn’t a good idea.
  - The syntax doesn’t match the way we’re used to using objects.
    ```java
    int value = roll(die);  
    ```
- The point of classes is to combine state and behavior.
  - `roll` belongs in the `Die` object.
    ```java
    int value = die.roll();
    ```
Instance methods

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

- instance method declaration, general syntax:
  ```java
  public <type> <name> ( <parameter(s)> ) { <statement(s)> ; }
  ```

  (same as with static methods, but without the `static` keyword)

Getting the dice rolling – using instance methods

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

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

```java
Die die1 = new Die();
die1.numFaces = 6;
int value1 = die1.roll();

Die die2 = new Die();
die2.numFaces = 10;
int value2 = die2.roll();
```

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.

Object initialization: constructors

Initializing objects

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

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

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

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

Constructors

- **constructor**: Initializes the state of new objects.
  - Constructor syntax:
    ```java
    public <type> ( <parameter(s)> ) { <statement(s)> ; }
    ```

  - A constructor runs when the client uses the `new` keyword.
  - A constructor does not specify a return type; it implicitly returns the new object being created.
  - If a class has no constructor, Java gives it a default constructor with no parameters that sets all the object's fields to 0.
Die constructor

```java
import java.util.Random;

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;
    }
}
```

Multiple constructors are possible

```java
import java.util.Random;

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**: 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

- Fields can be declared private to indicate that no code outside their own class can access or change them.
  - Declaring a private instance variable:
    ```java
    private <type> <name>;
    ```
  - Examples:
    ```java
    private int faceValue;
    private String name;
    ```
- Once instance variables are private, client code cannot access them:
  ```java
  Roll.java:11: faceValue has private access in Die
  System.out.println("faceValue is " + die.faceValue);
  ^
  ```

Comment

- In our initial implementation of the Die class we didn’t use access modifiers:
  ```java
  public class Die {
      int numFaces;
      int faceValue;
  }
  ```
- This is the same as using the public access modifier:
  ```java
  public class Die {
      public int numFaces;
      public int faceValue;
  }
  ```
Accessors and mutators

- We provide accessor methods to examine their values:
  ```java
  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:
  ```java
  public void setFaceValue(int value) {
    faceValue = value;
  }
  ```
  - Client code will look more like this: Not recommended in this case!
    ```java
    System.out.println("faceValue is " + die.getFaceValue());
    ```

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- 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.