The basic building block on an object-oriented language is an **object**, simulating a real-life object.

- A **class** is like a blueprint from which you can create many objects that may have different characteristics.

- An object has **state**, defined by the values of its attributes:
  - The **attributes** are defined by the data associated with the object’s class.

- An object has **behaviors**, defined by the operations associated with it:
  - Behaviors (operations) are implemented by the **methods** of the class.
A class contains **data** declarations and **method** declarations.

An **object** is an **instantiation** of a class.

The **values** of the data are the **object’s state**.

The **functionality** of the methods define the **object’s behavior**.

---

Generally, classes that represent tangible things are called using names that are **singular nouns**:

- Examples: Coin, Student, Classroom

Generally, the methods that encapsulate behaviors are called using names that are **verbs**:

- Examples: flip, register, assign, get, set

What are the data and methods you would define for class **Coin**?
Consider a six-sided die (singular of dice)

- What should its state be?
- What should its primary behavior be?

We represent a die in Java by designing a class called Die that models its state and behavior.

We want to design the Die class with other data and methods to make it a versatile and reusable resource.
Encapsulation

- Enforces access to an object’s data only through specific methods – PROTECTS the class implementation
- A well **encapsulated** object can be thought of as a non-transparent box - the inner workings are hidden from whomever is using it (the **client**)
- The client invokes the interface methods of the object, which manages the instance data

Visibility Modifiers

- A **modifier** specifies particular characteristics of a method or data
- Java has three visibility modifiers: **public**, **protected**, and **private**

<table>
<thead>
<tr>
<th>Variables</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>violates encapsulation</td>
<td>enforces encapsulation</td>
</tr>
<tr>
<td>Methods</td>
<td>provides services to clients</td>
<td>supports other methods in class</td>
</tr>
</tbody>
</table>
Anatomy of a Class

- Consider a six-sided die (singular of dice)
  - Its state can be defined as which face is showing
  - Its primary behavior is that it can be rolled
- We can represent a die in Java by designing a class called Die that models this state and behavior
- We want to design the Die class with other data and methods to make it a versatile and reusable resource

Let’s see how we would use Die to play snakeEyes, that is, write a client for Die

```java
public class SnakeEyes { public static void main(String[] args) { final int ROLLS = 500; int num1, num2, count = 0; // Instantiate two new Die objects

die1 ─── faceValue 1
die2 ─── faceValue 1

for(int roll = 1; roll <=ROLLS; roll++) {
    // Roll die, save each faceValue into num1 and num2
    // Check for snake eyes
}

System.out.println("Number of rolls: " + ROLLS);
System.out.println("Number of snake eyes " + count);
System.out.println("Ratio: " + (float)count/ROLLS);
```
Constructors

- A **constructor** is a special method which builds a new instance of the class.
- Note that a constructor has no **return type** in the method header, **not even void**.
- A **common error** is to put a return type on a constructor, which makes it a "regular" method that happens to have the same name as the class.
- The programmer does not have to define a constructor for a class:
  - Each class has a **default constructor** that accepts no parameters.

```java
import java.util.Random;
/**
 * Represents one die with faces between 1 and 6
 * @author Java Foundations
 */
public class Die {
    private final int MAX = 6; // max face value
    private int faceValue; // current value showing

    public Die(){ // Constructor! Sets initial value.
    }/**
     * Computes a new face value for this die
     * @return the new face value between 1 and MAX
     */
    public int roll(){
    }
}
/**
 * Face value mutator. Only modified if value is valid
 * @param value die is set to this integer, 1 to MAX
 */
 public void setFaceValue(int value){
}
/**
 * Face value accessor.
 * @return the current face value of this die
 */
 public int getFaceValue() {
}
/**
 * @return string representation of this die
 */
 public String toString() {
}
}

---

**UML Diagrams**

- A UML class diagram showing the classes involved in the SnakeEyes program:
Wrapper Classes in Java

- Not all data types in Java are objects
  - Some are **primitive data types** (but have related objects)
  - All primitive data types have a corresponding Wrapper Class

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Char</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Control Flow

- Understanding the control flow is essential to debugging!

```java
class SnakeEyes
    main
        die = new Die();
        die.roll()

class Die
    Die()
    roll
```

More on Control Flow

- If the called method is in the same class, only the method name is needed
- If the called method is part of another class, use the dot notation
- Understanding the control flow is essential to debugging!

```java
main
    obj.doThis();
    helpMe();
```

```java
doThis
    helpMe();
```
Suppose we execute the following `main()` method:

```java
// Main method... the Bronte sisters’ grades in CS230
public static void main(String[] args) {
    Grade charlotte = new Grade("B-", 82.1);
    Grade emily = new Grade("A", 94.5);
    Grade anne = new Grade("C+", 79.0);

    System.out.println(charlotte.isHigherThan(emily));
    System.out.println(Grade.max(charlotte, emily));
}
```
We need to write a `Grade` class that contains (at least) a constructor, and a few of methods. You may think that `isHigherThan` and `max` do essentially the same thing (a comparison of scores) but they are defined differently:

```java
public Grade(String letterGrade, double numericalGrade)

public boolean isHigherThan(Grade g)

public static Grade max(Grade g1, Grade g2)
```

**Instance vs Static**

Reusing Classes
/**
 * Represents a coin with two sides that can be flipped.
 * @author Java Foundations
 */
public class Coin {
    private final int HEADS = 0; // tails is 1
    private int face; // current side showing
    
    /**
     * Constructor: Sets up this coin by flipping it initially.
     */
    public Coin () { ... }
    
    /**
     * Flips this coin by randomly choosing a face value.
     */
    public void flip () { ... }
    
    /**
     * @return true if the current face of this coin is heads, false otherwise
     */
    public boolean isHeads () { ... }
    
    /**
     * @return string representation of this coin
     */
    public String toString() { ... }
}

// CountFlips.java uses Coin.java

/*
 * Demonstrates the use of a programmer-defined class.
 * @author Java Foundations
 */
public class CountFlips {
    /**
     * Driver: Flips a coin multiple times and counts the number of heads
     * and tails that result.
     */
    public static void main (String[] args) {
        final int FLIPS = 10000;
        int heads = 0, tails = 0;
        Coin myCoin = new Coin();
        for (int count=1; count <= FLIPS; count++) {
            myCoin.flip();
            if (myCoin.isHeads())
                heads++;
            else
                tails++;
        }
        System.out.println("Number of flips: " + FLIPS);
        System.out.println("Number of heads: " + heads);
        System.out.println("Number of tails: " + tails);
    }
}
FlipRace.java also uses Coin.java

```java
public class FlipRace {
    /**
     * Demonstrates the reuse of a programmer-defined class.
     * @author Java Foundations
     */
    public static void main(String[] args) {
        final int GOAL = 3;
        int count1 = 0, count2 = 0;
        Coin coin1 = new Coin(), coin2 = new Coin();
        while (count1 < GOAL && count2 < GOAL) {
            coin1.flip();
            coin2.flip();
            System.out.println("Coin 1: " + coin1 + "\tCoin 2: " + coin2);
            count1 = (coin1.isHeads()) ? count1 + 1 : 0; // Increment or reset the counters
            count2 = (coin2.isHeads()) ? count2 + 1 : 0;
        }
        if (count1 < GOAL) System.out.println("Coin 2 Wins!");
        else if (count2 < GOAL) System.out.println("Coin 1 Wins!");
        else System.out.println("It's a TIE!");
    }
}
```