CS231: Lecture 2 (More Java)
The following example shows a declaration of a public class Die with a single private int field numFaces and two constructor methods. numFaces indicates which face of the Die is on top.

```java
public class Die {
    private int numFaces;  // instance variables

    //Constructor for objects of class Die
    public Die() {
        numFaces = 6;
    }

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

- Compile and run
  - Compile but doesn't do anything
- 3 parts: field declaration, constructor declaration, method declaration.
  - **Fields** are for the data held by an object
  - **Constructors** are called when new objects are being created
  - **Methods** are the behavior of the objects, that is, what they can do for you if you ask them nicely.
- Each of them needs an accessibility specification. "Private" means that it can only be accessed within the class. "Public" means everything can access it. Your default should be to use private for fields and public for everything else.
- Every **field** needs one type. Python variables can have values of any type (dynamically typed), but Java variables can only have values of one type and you specify what that type is when you declare the variable (statically typed). One way to think about the difference is that in Python, data has a type but in Java, data and variables both have types and if you want to assign data to a variable, the types better match. In Java, a variable can only have one type of data and you have to declare that type before you use the variable. Why the restriction? [HW] [It allows Java to detect errors where you accidentally assign the wrong type of data to a variable.]
- **Primitive data types**: byte, short, int, long, float, double, Boolean, char [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html]
- **Constructors** have names that are the same as the class name. They are mostly used to initialize values of the fields.
- **Methods** must include a return type in their declaration. In our example here, the first method returns an integer and the second returns nothing and so we declare it to be of type "void".
- **Variable** numFaces
  - The field is not declared static. This means that it can have a different value for each instance of Die. A field like this is called an instance field or instance
variable. Since this is the most common kind of field, it is often simply called a field.

- The field is declared private. Instance fields are normally declared private. This means that they cannot be accessed by methods in other classes. When other classes do things with Die instances, code in those classes can't access private fields directly. This is an example of **information hiding**.
- Therefore, any class that wants to access the variable numFaces should access them through these getters and setters. They are called **accessor functions**.

Let's add a main and create a couple of objects and print the top face value. We add the roll() method. The implementation involves the static random() method from the built-in Math class. This method returns a random double which is at least 0 and less than 1. If we multiply the result by 6, throw away any fractional part by casting it to an int, and add 1, we get a random int between 1 and 6 inclusive.

```java
/**
  * This class models a standard 6-sided die.
 */
public class Die {
    private int numFaces; // instance variables

    // Constructor for objects of class Die
    public Die() {
        this.numFaces = 6;
    }

    public Die(int numFaces) {
        this.numFaces = numFaces;
    }

    // accessor
    public int getNumFaces() {
        return this.numFaces;
    }

    // Roll the die and return the number
    // that ends up on top.
    public int roll() {
        double r; // declare
        r = Math.random();
        int topFace;
        topFace = (int) (r*this.numFaces) + 1;
        return topFace;
    }

    public static void main(String Args[]) {
        Die die1 = new Die(); // create a Die instance called die1 (we are using class Die to create objects)

        System.out.println("Die 1 has " + die1.numFaces + " faces"); // value of die before rolling
        System.out.println("Rolling die 1");
        // for (int i = 0; i < 20; i = i + 1)
```
for (int i = 0; i < 20; i++) {
    System.out.println("Roll output: " + die1.roll() );
}

Die die2 = new Die(4); //another instance called die2
System.out.println("Die 2 has " + die2.numFaces + " faces"); // value of die before rolling
System.out.println( "Rolling die 2 ");
for (int i = 0; i < 20; i++) {
    System.out.println("Roll output: " + die2.roll() );
}
}

• this. Keyword
  • this is a reference variable that refers to the current object.
  • Problem (without this keyword)

    class Student{
        int id;
        String name;

        Student(int id,String name){
            id = id;
            name = name;
        }
        void display(){
            System.out.println(id+" "+name); } } } public static void main(String args[]){
    Student s1 = new Student(111,"Jane");
    Student s2 = new Student(321,"David");
    s1.display();
    s2.display();
    }

  • The parameter/local variable and instance variables are the same - that is why we are using this keyword to distinguish between local variable and instance variable.
  • Solution (with this keyword)

    class Student{
        int id;
        String name;

        Student(int id,String name){

this.id = id;
this.name = name;

} 
void display(){
    System.out.println(id+" "+name);
}

public static void main(String args[]){
    Student s1 = new Student(111,"Jane");
    Student s2 = new Student(222,"David");
    s1.display();
    s2.display();
}

  o  Note: If local variables (formal arguments) and instance variables are different, there is no need to use this keyword