  - `println()`: a method of PrintStream class, prints the argument passed to the standard console and a newline
- Java is case-sensitive.
- The naming convention is camelCasing (ClassName, otherNames). Start with letters or _.

Class Fields, Constructors, and Methods

- Java is an OO language. It uses classes to describe the template of objects.
- A Java class usually has three part:
  - **Fields**: hold the data/attributes of an object (e.g., name, ID, and GPA of a student)
  - **Constructors**: called when new objects are created
  - **Methods**: the behavior of an object (e.g., you can let the student class have a getName method that can tell you the student's name when calling the method.)
- Now, let's take a look at a declaration of a class Die, and find out what are those three parts.

```
public class Die {

     // a field that specifies the number of surfaces on a die.
     private int faces;

     // a non-argument constructor that specifies the default number of surfaces
     public Die () {
          faces = 6;
     }

     // a getter (method) that returns the information of an object
     public int getFaces () {
          return faces;
     }

     // a setter (method) that changes the field's value
     public void setFaces (int numFaces) {
          faces = numFaces;
     }
}
```

- Each of them needs an accessibility specification.
  - Public: everything can access it.
  - Private: it can be only accessed with in the class.
  - For now, you should use private for fields and public for everything else by default.

- As Java is a statically typed language, every field needs one type.
  - Python variables can have values for any types.
  - Java variables can only have values of one type, and the type is specified when you declare the variables. Variables must be declared before you can use them.
  - This is because Java needs to decide the variable types when compiling, while Python can do that when running the script.
  - This means that if you accidentally assign a wrong type of data to a variable, you will get an error when you compile your Java code. You need to fix the error and recompile the code. If there is no more error, you will get a .class file. Then you can run your code.

- **Types**:
  - Primitive types: `boolean, char, byte, short, int, long, float, and double`
    - predefined by the language and is named by a reserved keyword.
  - Object types:

  - Classes in Java are also types. Classes can be built-in classes of Java or user defined classes (e.g. Die).
  - Java Application Programming Interface (API) lists all classes that are part of the Java Development Kit (JDK), including all Java packages, classes, and interfaces, along with their fields, and constructors. These built-in classes provide a tremendous amount of functionality to a programmer. [https://docs.oracle.com/javase/8/docs/api/]
  - Those prewritten classes include `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`, which wrap the values of primitive types in objects. These object types also provides several methods. E.g., Integer provides methods for converting an `int` to a `String` and a `String` to an `int`.

- **Constructors** have names that are the same as the class name. They have no return type and are mostly used to initialize the fields.
- **Methods** must include a return type in their declaration. In the above example, the method getFaces return faces which is an `int` value. So, the return type of the method is `int`. If a method returns nothing, the return type is `void` (e.g., the setFaces method).
- **Field** `faces`
  - 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 is 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 fo **information hiding**.
  - Therefore, any classes that wants to access the variable `faces` should access through these getters and setters. They are called **accessor functions**.

- Now let's compile and run the code [javac Die.java / java Die].
  - We can compile the code successfully, but cannot run it. Why? [Don't have a main method]
- Let's add a main method to the class. The main method creates an object of Die, and tests the getter and setter.

```
public class Die {

     // a field that specifies the number of surfaces on a die.
     private int faces;

     // a non-argument constructor that specifies the default number of surfaces
     public Die () {
          faces = 6;
     }

     // a getter (method) that returns the information of an object
     public int getFaces () {
          return faces;
     }

     // a setter (method) that changes the field's value
     public void setFaces (int numFaces) {
          faces = numFaces;
     }

     public static void main (String[] args) {
          // create an object of Die class using the non-argument constructor.
          Die d = new Die();
          // test the getFaces method and print out its return value
```

```
            System.out.println("value of faces: " + d.getFaces());
            // test the setFaces method
            d.setFaces(4);
            // check if the setFaces method worked
            System.out.println("value of faces after change: " + d.getFaces());
      }
}
```

Java Operations, Flow Control, Math class, “this”

- Let’s make a more advanced Die class.
  - The class can allow users to create objects with any number of faces.
  - The class can set the value of faces the same as a given Die object.
  - The class has a clone method, which can duplicate the current object.
  - The class has a roll method which can generate a random value between 1 and the faces.
  - A equals method that can compare the values of two objects.
  - The main method should test the rest methods.

```
public class Die {

      // a field that specifies the number of surfaces on a die.
      private int faces;

      // a non-argument constructor that specifies the default number of surfaces
      public Die () {
            faces = 6;
      }

      // a constructor that allows users to specify the number of surfaces
      public Die (int numFaces) {
            faces = numFaces;
      }

      // a getter (method) that returns the information of an object
      public int getFaces () {
            return faces;
      }

      // a setter (method) that changes the field's value
      public void setFaces (int numFaces) {
            faces = numFaces;
      }

      // set the faces to the value of a given object
      public void set (Die die) {
            faces = die.faces;
      }

      // clone method
      public Die clone () {
            Die d = new Die(this.faces);
            return d;
      }

      // roll the die using the random function from the math package
      public int roll () {
            int x = ((int)(Math.random() * faces)) + 1;
            return x;
      }

      // equals function that compare the values of objects
```

```
    public boolean equals (Die d) {
        return this.faces == d.faces;
    }

    public static void main (String[] args) {
        // create an object of Die class using the non-argument constructor.
        Die d = new Die();
        // test the getFaces method and print out its return value
        System.out.println("number of faces on d: " + d.getFaces());

        // create another object using non-default constructor
        Die b = new Die(4);
        System.out.println("number of faces on b: " + b.getFaces());
        // create a clone object of b
        Die c = b.clone();
        System.out.println("number of faces on c: " + c.getFaces());

        // test the setFaces method
        b.setFaces(d.getFaces());
        // check if the setFaces method worked
        System.out.println("number of faces on b: after setFaces: " + b.faces);

        // test the set method
        c.set(b);
        // check if the set method worked
        System.out.println("number of faces on c: after set: " + c.faces);


        System.out.println("*** Testing roll *** ");
        // roll 10 times and check the num of faces for each roll
        for (int i = 0; i < 10; i++) {
            int value = d.roll();

            if (value == d.getFaces()) {
                System.out.println("Max: " + value);
            }
            else if (value > 1) {
                System.out.println("Val: " + value);
            }
            else {
                System.out.println("Min: " + value);
            }
        }

        System.out.println("*** Testing equals ***");
        System.out.println("b == c: " + (b == d));
        System.out.println("d.equals(c): " + d.equals(c));
    }
}
```

- **Multiple constructors**: A Java class can have multiple constructors. Those constructors must have different argument list, as Java platform distinguishes them based on the number of arguments and their types. You must be very careful when do not give a constructor to a class. The compiler will provide a default constructor for any class without constructors. The default constructor will call the non-argument constructor of the superclass. If the superclass does not have a non-argument constructor, the compiler will complain. If the class has no explicit superclass, it will call the non-argument constructor from Object class.