Variable scope

- The **scope** of a variable is where the variable is accessible
- The **scope** depends on where the variable is declared
- There are three (3) levels of scope
  - **Class level scope**: the variable is accessible in the entire class. *Instance variables*
  - **Method level scope**: the variable is accessible inside the method. *Local variables and parameters*
  - **Block level scope**: the variable is accessible inside the body of a loop. *Loop control variables*
Variable scope

```java
public class Person {
    // instance variables
    private String name;
    private String email;

    public void verticalPrint(int length) {
        for (int i = 0; i < length; i++) {
            System.out.println(name.charAt(i));
        }
    }
}
```
Static variables and methods

- Instance variables and methods define the attributes and behavior of the objects
- Instance variables and methods are called with the object name:
  ObjectName.methodName();
  ObjectName.varIableName;

- Static variables and methods belong to the class
- Static variables and methods are called with the class (and object) name
  ClassName.staticMethodName();
  ClassName.staticVariableName;
Static variables and methods

- The `static` keyword is placed right after the public/private modifier when defining static variables and methods.
- A `static` method can be public or private.
- To declare a `static variable`, you write:

```java
public static VariableType VariableName;
```

```java
public static int numberOfStudents;
```
Static variables and methods

- To define a static method, you write

```java
public static MethodType MethodName(/* parameters */);

public static int getNumberOfStudents()
{
    return numberOfStudents;
}
```
Static variables and methods

- All objects share the same copy of a static variable

```java
public class Student {
    private String name;
    private int age;
    public static int numberOfStudents;

    public Student(String newName, int newAge) {
        name = newName;
        age = newAge;
        numberOfStudents++;
    }

    public static int getNumberOfStudents() {
        return numberOfStudents;
    }
}
```

```java
Student s1 = new Student("Alice", 12);
System.out.print(Student.getNumberOfStudents()); // 1
Student s2 = new Student("Dwayne", 13);
System.out.print(Student.getNumberOfStudents()); // 2
Student s3 = new Student("Rachel", 10);
System.out.print(s3.getNumberOfStudents()); // 3
```

The same copy of `numberOfStudents` is incremented for all objects.
this keyword

- this is used inside a non-static method to refer to the current calling object
- Can be used to refer to instance variables (with the ‘.’ operator)
- It can be used to differentiate between instance variables and parameters

```java
public class Student{
    private String name;
    private int age;

    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }

    public String getName(){
        return this.name;
    }
}
```
Class Design
Overview

- We know how to write classes in general now
  - Pick the entity you’re trying to model
  - Decide on its attributes and behaviors
    - implement them as instance variables and methods!
- Next, we want to think about writing well-designed classes
  - Classes that do just what we need them to do
  - Classes that aren’t vulnerable to being misused
Who Designs Objects?

- Basically every programmer!
- Example:
  - We use String objects, which were designed by Java’s programmers.
    - We use String objects to write our own classes (Student objects have a name attribute, e.g.)
    - String objects themselves are designed to use constructs written at a lower level
How do we decide what our new objects should do?

- **Abstraction**: set of information properties relevant to a stakeholder about an entity
  - Information Property (or property): a named, objective and quantifiable aspect of an entity
  - Stakeholder: a real or imagined person (or a class of people) who is seen as the audience for, or user of the abstraction being defined
Class Design

- Entity: Movie
- Properties:
  - Title
  - Year
  - Length
  - Genre
  - Format
  - Price
Class Design

<table>
<thead>
<tr>
<th>Movie</th>
<th>On-Line Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong> (string)</td>
<td><strong>Year</strong> (int)</td>
</tr>
<tr>
<td>“Moneyball”</td>
<td>2011</td>
</tr>
<tr>
<td>“Gone With the Wind”</td>
<td>1939</td>
</tr>
<tr>
<td>“Jurassic Park”</td>
<td>1993</td>
</tr>
<tr>
<td>“Pirates of the Caribbean”</td>
<td>2003</td>
</tr>
<tr>
<td>“Sicko”</td>
<td>2007</td>
</tr>
</tbody>
</table>

Representing the Movie Abstraction using a Table
Who Uses Objects?

● …Basically every programmer!
● It would be troublesome to have to learn in great detail how every object we encounter works.
  ○ Do you care whether a String is really a char[] or something else?
  ○ Do you want to have to explain to someone how your NBody.java works, or would you rather show them what it does?
How do we decide how our objects are used?

- **Encapsulation**: the hiding of unimportant details in the implementation of an object.
  - For a user of an object, it doesn’t really matter what the types of the instance variables are, or what order the constructor works in.
  - What matters is **what the object can be asked to do!**
The Public Interface

- **Abstraction and Encapsulation**: applied together, these allow a programmer to design an object that…
  - Has methods for everything a user might need the object for
  - Hides the implementation to reduce confusion and misuse
The Public Interface

- **The public interface**: the collection of public methods in an object
- Make every desired behavior a **public method**.
- Keep every instance variable **private**.
- If some instance attribute might be relevant to a user, write an accessor method for that attribute.
- If some instance attribute might be changed by a user, write a mutator method for that attribute.