Lecture 4: Java and Objects

Sam McCauley
Data Structures, Spring 2026

Classes and Objects

Java is an *object-oriented language*. Classes and objects are fundamental to how Java operates. In the next part of the course we'll learn about what classes and objects are, and the basics of how to use them in Java.

Throughout the course, we will keep returning to classes and objects, learning further functionality and how to use it to write effective and scalable code.

What do Classes and Objects Accomplish? Objects and classes are used to *organize* code. That is to say: you could write a perfectly correct Java program without ever taking advantage of objects and classes.

You can think of objects and classes as generalizing functions. We use functions to organize code, and allow us to easily call pieces of code repeatedly. Objects and classes have much the same purpose: we want to group parts of our program together to allow us to organize it better, and to allow us to reuse it effectively.

What is a Class? What is an Object? A *class* is a way to group code and data together in one place. This is why I say classes "generalize" functions: functions group together code for a single purpose, while classes group code *and data* together for a single purpose.

With this in mind, a class will have two parts. First, it will have some data; perhaps a few variables that store related values. Then, it will have some methods that operate on these values. Either of these parts is optional: a class may have only data, or may have only methods.

An *object* is a specific instance of the class. This distinction is important when we're dealing with data. You can think of an object like a variable. The Java code `int x = 10;` means that x stores the value `10`. A class is much closer to a variable *type*: much like the keyword `int` in the example `int x = 10;`

Classes and objects are likely best learned through examples. Let's look at a longer use case that motivates what they are, and why they are useful for coding.

Object-Oriented Example: List of Students

Let's say that I want my code to keep track of a classroom of 20 students. For each student, I want to write down their name, which courses they are taking, and their graduation year.

Using Java as we've learned so far (without objects), we have to keep track of each type of information in a separate array. I think we'd do something like this.

```java
String[] names = new String[20];  //array of 20 names
String[][] courses = new String[20][4];  //array of length 20, where
    each entry is itself an array of 4 Strings
int[] graduationYear = new int[20]; //array of 20 graduation years
```

This is technically workable. If I want to get the name of the 8th student, I can access `names[7]`. If I want to get their graduation year, I can access `graduationYear[7]`.

Problems with This Approach. The first problem of this approach is that it's difficult to work with. I need to set up all of these arrays (including an array of arrays). Then, data is hard to access: the data for a single student is spread across all of these arrays (or even an array of arrays). If I want to make a new student, I need to make 3 separate variables for the new student, and again keep track of all 3 explicitly.

This problem gets worse with scaling: imagine we had to keep track of 10 or 20 variables for each student. Or imagine each student had an "assignment partner"; how would we keep track of that? We could keep track of the partner as an `int` index in the array, but this is unintuitive. We could keep track of the partner's `name`, but then to find (say) the partner's graduation year we would have to search through the entire array. It would be much better if we could keep track of an entire student explicitly.

And there is an even bigger problem. This code is highly prone to *mistakes*. I am depending on indices to separate one student from another. If I look up the name of student 7, but accidentally change the grade of student 6, I'm suddenly storing incorrect data! And this problem could easily evade detection. This problem becomes more worrisome if there are, say, 10 or 15 programmers all working on this dataset.

An Object-Oriented Approach. Let's create a *class* that holds all of the information for a single student together. We'll call this the `Student` class. Note that in Java, it is convention that class names are always capitalized. Ignore the keyword `public` in the following for now; we'll come back to this later.

```java
public class Student {
    public String name;
    public String[] courses = new String[4];
    public int graduationYear;
}
```

This is called a *class declaration*, because it is creating a new class, called `Student`. It must be in a file with the same name: so this class must be in `Student.java`. Each variable in the class is called an *instance variable*.

Now, all of the information for a given student is stored in one place.

We can now declare and instantiate a student just like we would declare and instantiate an instance variable. Let's say a new student is adding the class. We can create a variable for the new student as follows. Creating a new object uses the new keyword and also uses parentheses—these will make sense in a moment (see the discussion below on "constructors").

```java
Student addingTheClass = new Student();
```

Remember that a specific instance of a class is called an *object*. So we would say that `addingThe-Class` is an object of the class `Student`.

Now, we can access and modify data for an object using the `.` operator, using the same notation as we do for normal variables.

```java
addingTheClass.name = "Sam";
addingTheClass.graduationYear = 2010;
addingTheClass.courses[0] = "CS 136";
System.out.println("The new student is named " + addingTheClass.name +
    "and graduates in " + addingTheClass.graduationYear);
```

Arrays of Objects. Let's look at how we can now store 20 `Student` objects in a single array.

To create our array, we now can declare a single array. Each entry in the array holds the data for a single student.

```java
Student[] stuList = new Student[20]; //create an array of 20 Students
```

Unlike in arrays of primitive types, each object in the array must be instantiated. In other words, we need to write new `Student()` for *each* `Student` we create.

```java
for(int index = 0; index < stuList.length; index++) {
    stuList[index] = new Student();
}
```

Note that we use parentheses here. Looking ahead, we will soon see the rationale behind why each object in the array must be instantiated individually, as well as why we have parentheses when we do: each time Java sees new `Student()`, it is calling the constructor method for that `Student`.

With this setup, we have now stored the students within a single array of `Student` objects. We can look up the name of the 8th student using `stuList[7].name`; we can look up their graduation year using `stuList[7].graduationYear`.

Methods for the Student Example. Classes don't just help us organize data, they help us organize methods that interact with the data.

Continuing the previous example, it seems reasonable to have a method `findStudentYear` that uses their graduation year to figure out what year they are as a student: "first-year", "sophomore", "junior", or "senior". We can add that method directly to the student class as below. Note that we are *not* using the `static` keyword this time. We will discuss this distinction soon.

```java
public class Student {
    public String name;
    public String[] courses = new String[4];
    public int graduationYear;

    public String findStudentYear(){
        int currentYear = 2026;
        if(graduationYear < currentYear || graduationYear - currentYear
             > 3) {
            return "Undefined";
        }
        String[] years = { "Senior", "Junior", "Sophomore", "First-year
            " };
        return years[graduationYear - currentYear];
    }
}
```

A key takeaway of this example is that the methods of a class have access to the instance variables—we did not need to pass `graduationYear` in as a parameter in the above.

You may also note that unlike in Python, we do not need to list "self" (or anything like it) as a parameter to the method.

Methods of a class are invoked using the `.` notation. Assume that `s1` is a student in the following example.

```java
System.out.print("s1 is a " + s1.findStudentYear());
```

Accessing Instance Variables Using Methods

There is an issue with how we have been interacting with objects so far. In object-oriented programming, *the instance variables should only be accessed through class methods*. More specifically, we had an example with the following lines of code:

```java
addingTheClass.name = "Sam";
addingTheClass.graduationYear = 2010;
addingTheClass.courses[0] = "CS 136";
System.out.println("The new student is named " + addingTheClass.name +
    "and graduates in " + addingTheClass.graduationYear);
```

This should never be done. Instead, each class should have associated methods to access the data, called *getter* and *setter* methods.

Getter and Setter Methods. A method that accesses object data is called a *getter* method, and a method that changes the data is called a *setter* method.

Let's extend the `Student` class to include getter and setter methods. After, we'll use those methods to fix our code from before.

Here is the new `Student` class. You may notice that the `public` keyword has changed into `private` in the declaration of the instance variables, and the `static` keyword has disappeared from the method declarations—we'll come back to this soon.

```java
public class Student {
    private String name;
    private String[] courses = new String[4];
    private int graduationYear;
    private int idNumber;

    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
    public int getGraduationYear() {
        return graduationYear;
    }
    public void setGraduationYear(int newValue) {
        graduationYear = newValue;
    }
    public String[] getCourses() {
        return courses;
    }
    public int getIdNumber(){
        return idNumber;
    }
}
```

Let's rewrite our code from before using these methods to access and modify the instance variables.

```java
addingTheClass.setName("Sam");
addingTheClass.setGraduationYear(2010);
System.out.println("The new student is named " + addingTheClass.getName
    () + "and graduates in " + addingTheClass.getGraduationYear);
```

Accessing Instance Variables Through Methods. It bears emphasizing that methods of a class always have access to the instance variables of the class; they do not need to be passed as parameters. This is why `getName()` is able to access `name` in the above example.

Why Use Getters and Setters? One may quite reasonably wonder why we are doing this: it requires quite a lot of extra typing.

Let's start with a concrete observation about what we've done so far. We wrote setter methods for the `name` and `graduationYear` instance variables, but we did not have a setter method for `idNumber`. This means that `idNumber` cannot be changed.

To be clear, this is not a matter of security per se. Changing `idNumber` is not difficult—you can just write a setter function. The goal here is to help keep data separated.

For example, imagine you are part of a team of 5 people working on a code base, and you are responsible for writing the `Student` class. You ask your teammates to only interact with your class using getters and setters. That means that you know that your teammates, whatever they are doing with their code, will not change `idNumber`. This means you are safe to assume that a `Student`'s `idNumber` does not change while the code executes.

Another upside is that you can control *how* people access your data. This can be quite useful for things like error handling. For example, let's say I wanted to write a setter for the `courses` array that takes in a new index, and a new value to store at that index. I can check to make sure that the index is valid within the setter.

```java
public void setCourse(int index, String newCourse) {
    if(index < 0 || index >= courses.length) {
        System.out.println("Error: course access out of bounds!");
    } else {
        courses[index] = newCourse;
    }
}
```

Now, I don't need to check that the index is valid every time I update an assignment. I can just call the setter, and it does it for me.

Constructors. There is a very common and very important type of method that we must go over: a *constructor*. In Python, a constructor is the `__init__` method for a class.

A constructor method is called when the object is instantiated. It is responsible for any "setup" tasks for the object.

A constructor always has the same name as the class it is in. Let's create a constructor for the `Student` class. It's up to us what kind of "setup" it does. Some reasonable things may be to assign default values: let's set name to be the empty string, and `graduationYear` and `idNumber` to be -1.

```java
public Student() {
    name = "";
    graduationYear = -1;
    idNumber = -1;
}
```

Now, when we create a `Student`, these instance variables will be set to these default values.

Constructors can also have parameters. Let's say that we want to ensure that each student has a name. We could set up the constructor like this.

```java
public Student(String inName) {
    name = inName;
    graduationYear = -1;
    idNumber = -1;
}
```

To call a constructor with parameters, we place the arguments in the parentheses when instantiating the object. We could use the above constructor as follows:

```java
Student s1 = new Student("Sam");
//s1 is now a Student with name Sam, and graduationYear and idNumber -1
```

It is allowed to have multiple constructors in a single class so long as they have different parameter types. When we put together all of our methods into a single class, we'll see that it has both the `Student()` and `Student(String inName)` constructors.

Constructors and Instantiation. One common usage of constructors is to instantiate instance variables.

Arrays are one use case where this is important. You may have noticed that in our `Student` class, we declare and instantiate the `courses` variable at the same time. This may not make sense in some contexts. For example, maybe the number of courses is different for each student.

Let's look at a simpler Student class that uses a constructor to instantiate its `courses` array.

```java
public class Student{
    private String[] courses;
    public Student(int numAssignments) {
        courses = new String[numAssignments];
    }
}
```

Constructors and Helper Methods. Constructors should be lightweight, and should only be used to initialize the state of the object. They should not do any further work.

If initializing the state of an object requires computation, that computation should probably be split off into a *helper method* that is called by the constructor.

Going back to the `Student` example, let's say that we also store a `String` for what year the student is. This can be calculated at instantiation time using the graduation year. This functionality can then be split off into another method. Note that `setStudentYear()` actually sets the `studentYear` variable; it does not return anything.

```java
public class Student {
    private String name;
    private String[] courses = new String[4];
    private int graduationYear;
    private int idNumber;
    private String studentYear;

    public void setStudentYear(){
        int currentYear = 2026;
        if(graduationYear < currentYear || graduationYear - currentYear > 3) {
            studentYear = "Undefined";
        }
        String[] years = { "Senior", "Junior", "Sophomore", "First-year" };
        studentYear = years[graduationYear - currentYear];
    }
    public Student(String inName, int inIdNumber, int inGraduationYear) {
        name = inName;
        idNumber = inIdNumber;
        graduationYear = inGraduationYear;
        setStudentYear();
    }
}
```

Final Example. Let's put together all the pieces of our `Student` class. We will wind up with a single class where we can change the name or graduation year (but not the idNumber); we have getters to obtain the values of all instance variables, and `studentYear` is set by the constructor automatically.

Remember that this class must be in a file called `Student.java`.

```java
public class Student {
    private String name;
    private String[] courses = new String[4];
    private int graduationYear;
    private int idNumber;
    private String studentYear;

    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
    public int getGraduationYear() {
        return graduationYear;
    }
    public void setGraduationYear(int newValue) {
        graduationYear = newValue;
    }
    public String[] getCourses() {
        return courses;
    }
    public int getIdNumber(){
        return idNumber;
    }
    public void setStudentYear(){
        int currentYear = 2026;
        if(graduationYear < currentYear || graduationYear - currentYear
             > 3) {
            studentYear = "Undefined";
        }
        String[] years = { "Senior", "Junior", "Sophomore", "First-year
           " };
        studentYear = years[graduationYear - currentYear];
    }
    public Student(String inName, int inIdNumber, int inGraduationYear)
         {
        name = inName;
        idNumber = inIdNumber;
        graduationYear = inGraduationYear;
        setStudentYear();
    }
    public Student() {
        name = "";
        graduationYear = -1;
        idNumber = -1;
        setStudentYear();
    }
}
```

Access Modifiers

Let's finally discuss the `public` keyword that we have been seeing in our code.

Each instance variable in Java has an *access modifier* that states how it can be used by other classes. A `public` instance variable can be read and modified by any code. However, a `private` instance variable can only be accessed by the methods of that specific class.

Access modifiers tie in directly with the getter and setter methods we discussed earlier. A `private` variable can *only* be accessed through getter and setter methods. A `public` variable can be accessed directly, without using methods. Earlier, we said that you should always use getter and setter methods to access your variables. For the same reason, instance variables in a Java class should almost never be `public`.

Let's look at a new example. Let's say we want to keep track of transactions at a cafe. We use two classes to do this. The first, `public class Transaction`, is responsible for keeping track of a single transaction. The second, `public class Cafe`, keeps track of the information for the cafe: its name and the list of transactions.

The `Transaction` class has `private` instance variables.

```
public class Transaction{
    private String item;
    private double cost;

    public String getItem() {
        return item;
    }
    public double getCost() {
        return cost;
    }
    public Transaction(String inItem, double inCost) {
        item = inItem;
        cost = inCost;
    }
}
```

First, let's look at an *incorrect* version of the `Cafe` class, that attempts to access the `private` instance variables directly.

```java
public class Cafe{ //INCORRECT version; gives error
    private Transaction[] dailySales = new Transaction[100]; //keeps track of up to 100 transactions per day
    private int salesSoFar;
    private double currentProfit;

    public void addSale(Transaction newSale) {
        dailySales[salesSoFar] = newSale;
        salesSoFar++;
        currentProfit += newSale.cost; //Error!  Accesses private variable
    }

    public Cafe() {
        salesSoFar = 0;
        currentProfit = 0;
    }
}
```

A correct version will use the getter and setter methods, which are `public` and can be accessed directly.

```java
public class Cafe{ //Correct version
    private Transaction[] dailySales = new Transaction[100]; //keeps track of up to 100 transactions per day
    private int salesSoFar;
    private double currentProfit;

    public void addSale(Transaction newSale) {
        dailySales[salesSoFar] = newSale;
        salesSoFar++;
        currentProfit += newSale.getCost(); //Works!
    }

    public Cafe() {
        salesSoFar = 0;
        currentProfit = 0;
    }
}
```

Access Modifiers for Methods

Access modifiers can be applied to methods too. A `public` method can be accessed by any other class. This is why getters and setters are generally public—we want others to be able to access them. This is also why constructors are (for now) always `public`, and why the `main` method is always `public`.

A `private` method, on the other hand, cannot be accessed by other classes. This is often used for helper methods that are meant to be internal to the class. For example, the `setStudentYear()` method for `Student` may be a method that we only want `Student` to use internally (we may not want others to call it). We can set it to `private` to ensure that others will not call it.

```java
public class Student {
    private String name;
    private String[] courses = new String[4];
    private int graduationYear;
    private int idNumber;
    private String studentYear;

    /* Setters and getters go here; omitted for space */

    private void setStudentYear(){
        int currentYear = 2026;
        if(graduationYear < currentYear || graduationYear - currentYear
            > 3) {
            studentYear = "Undefined";
        } else {
            String[] years = { "Senior", "Junior", "Sophomore", "First-
                year" };
            studentYear = years[graduationYear - currentYear];
        }
    }
    public Student(String inName, int inIdNumber, int inGraduationYear)
         {
        name = inName;
        idNumber = inIdNumber;
        graduationYear = inGraduationYear;
        setStudentYear();
    }
}
```

When to set Methods to Public or Private. Some methods, like getters, setters, or constructors, are specifically designed to interface with the class. These should always be `public`.

Some methods are not designed to be interfaced with, such as internal helper methods. These should always be `private`.

The `private` keyword can come in quite handy as we scale our code. As we design more complicated classes, we will also see examples of helper methods that do "dangerous" operations to the data. For example, consider a class that maintains a sorted list. We may have a method that swaps two elements of the list. This method could be "dangerous" in that it may leave the list unsorted: we need to swap elements internally, but we do not want other users to be able to swap things around haphazardly.

Overall, access modifiers to methods are a significant help to creating good, scalable Java code. They communicate how the class is intended to be used. The `public` methods are like an API: they are the way the class is meant to be interacted with. The `private` methods, on the other hand, are the internals of the class—they detail how the back-end of the class works, in a way you do not need to interact with.

Details of Access Modifiers

So far we have discussed `public` and `private`. This is all we will be using right now. However, there are in fact four access modifiers in Java.

- `public` – the method or variable is accessible from anywhere
- `private` – the method or variable can only be accessed from within the class itself
- `protected` – the method or variable can only be accessed from within the class, or from classes that inherit from it. (We haven't seen inheritance yet.)
- *default* (this is the access modifier when there is no keyword)– the method or variable can only be accessed from within the class, or from class in the same package. (A "package" is a way to group classes; we won't discuss packages in this class. Classes stored in the same folder are considered to be in the same package by default.)

Access Modifiers for Classes

There is one last usage of `public` that we have not yet discussed: when we create a class, as in: `public class Student`. All of the classes we have seen so far have been `public`, and we will continue to work almost exclusively with `public` classes.

It is also possible to use other access modifiers for classes. This is most often used for small classes that act as a "helper class" that are stored *within* a `public` class. We will usually only be using `public` classes in this course.