

Object Oriented Composition

Composition (along with inheritance and polymorphism) is another fundamental concept of object-oriented programming and design. It describes a class that references one or more objects of other classes in instance variables. ***An object that is composed of other objects.*** This allows you to model a ***has-a*** association between objects.

You can find such relationships quite regularly in the real world. A car, for example, has an engine, an engine has a transmission, a transmission has gears and a clutch . . . etc, etc.

Main benefits of composition

Given its broad use in the real world, it's no surprise that composition is also commonly used in carefully designed software components. When you use this concept, you can:

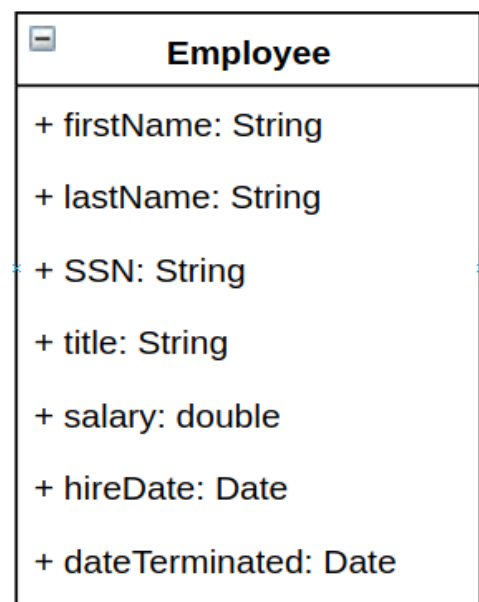
1. Reuse existing code. Classes that have been designed for one purpose could easily be plugged into other applications. A weapon object that was designed for Game A could easily be plugged into Game B, Game C . . . etc.
2. Change the implementation of a class used in a composition without adapting any external clients . . . **as long as the interface stays constant.** We could improve our collision detection mechanics without changing the actual interactions between the objects.
3. Apply the concepts of stepwise refinement and functional decomposition to your development process. We can problem solve, design, build and test a single object out of context of the larger application.

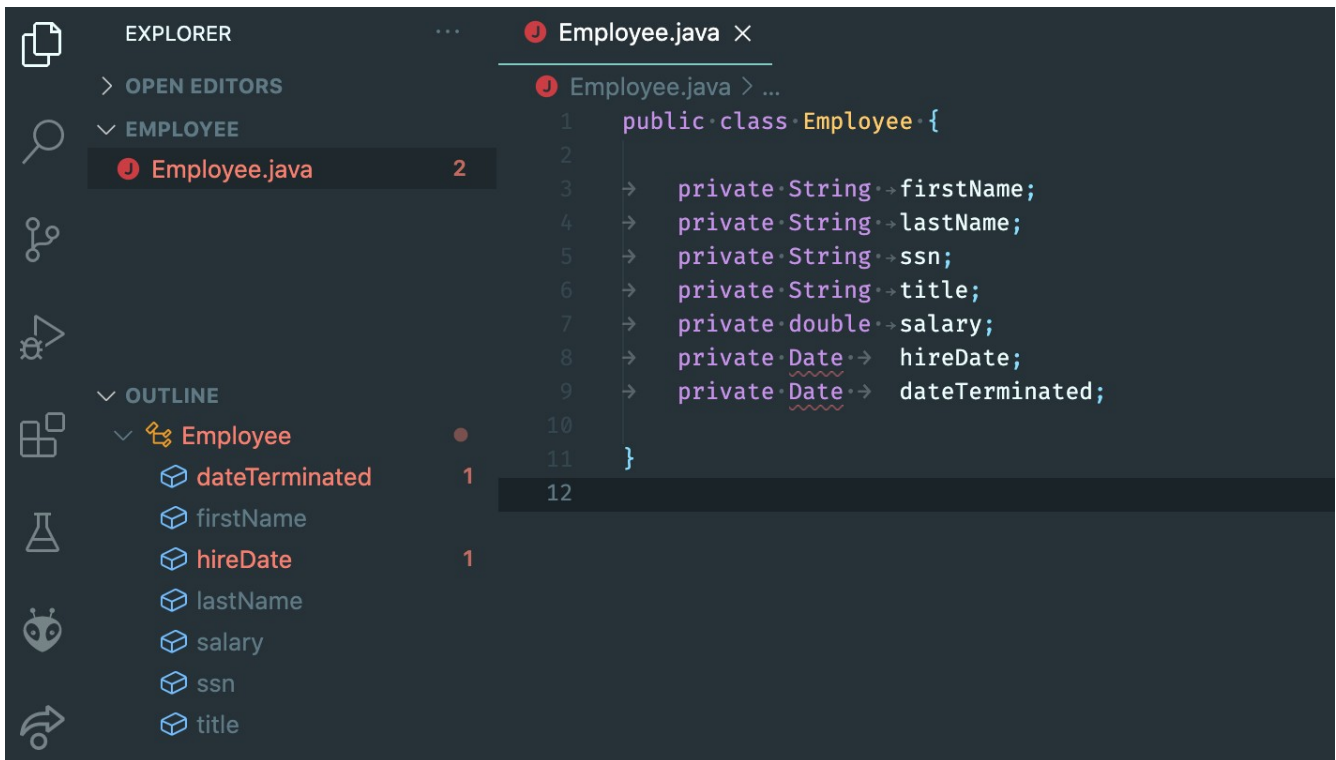
The Employee Class

Let's start with a simple class to define an Employee. This class could be used in payroll processing, to associate an employee with a task

- An employee sold a car
- An employee checked in a patient
- An employee accessed a card swipe in a protected room
- These examples are endless

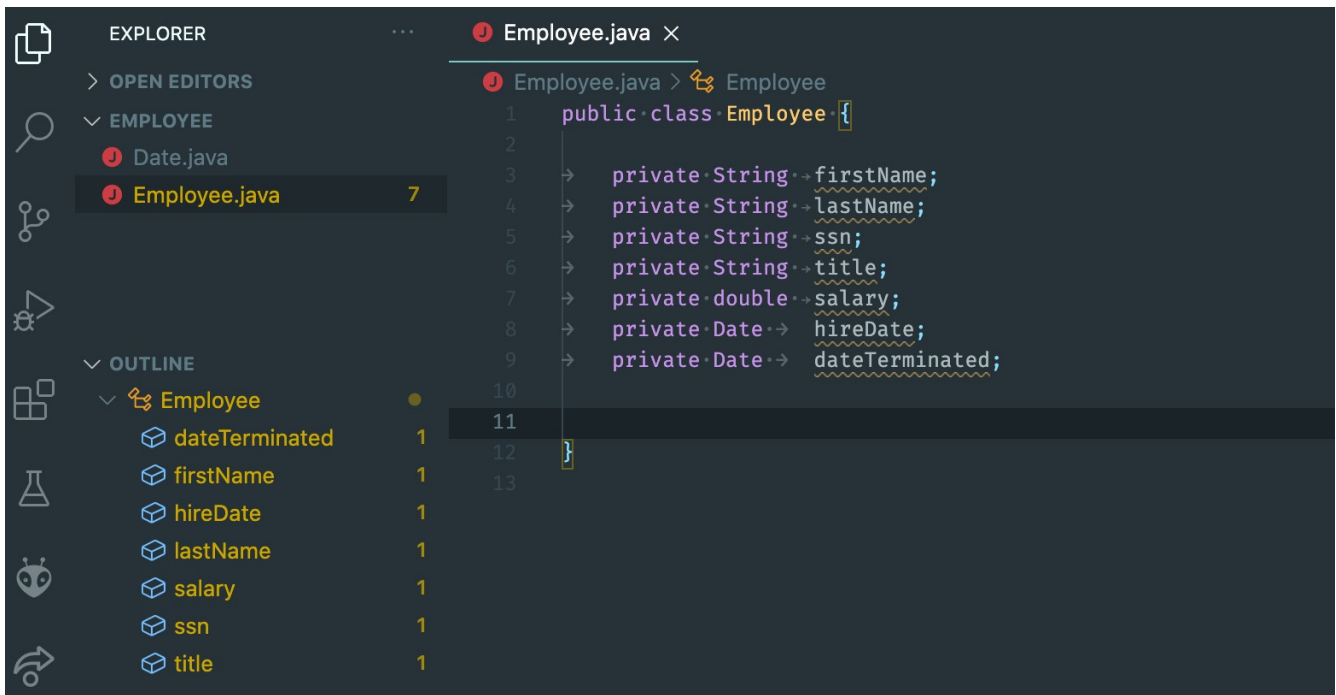
I have decided on the following attributes. The employee object is composed of various other objects. There are multiple String instances and two Date objects. As this code is getting implemented, I will be re-using the Date object that we designed in earlier lessons, and obviously we are re-using the String class from the Java API; this is also a prime example of composition that you have been using since day one.





You'll notice that the references to our custom Date class are not recognized. The easy way around this situation is to copy the source code file **Date.java** into the same directory as **Employee.java**. This will allow the Java compiler, as well VS Code to “see” the file and we will not need to import it. Obviously, this is **not** the optimal solution and it would be better to use an **import** statement and have your custom classes stored in a common directory . . . similar to how we configured the JavaFX library.

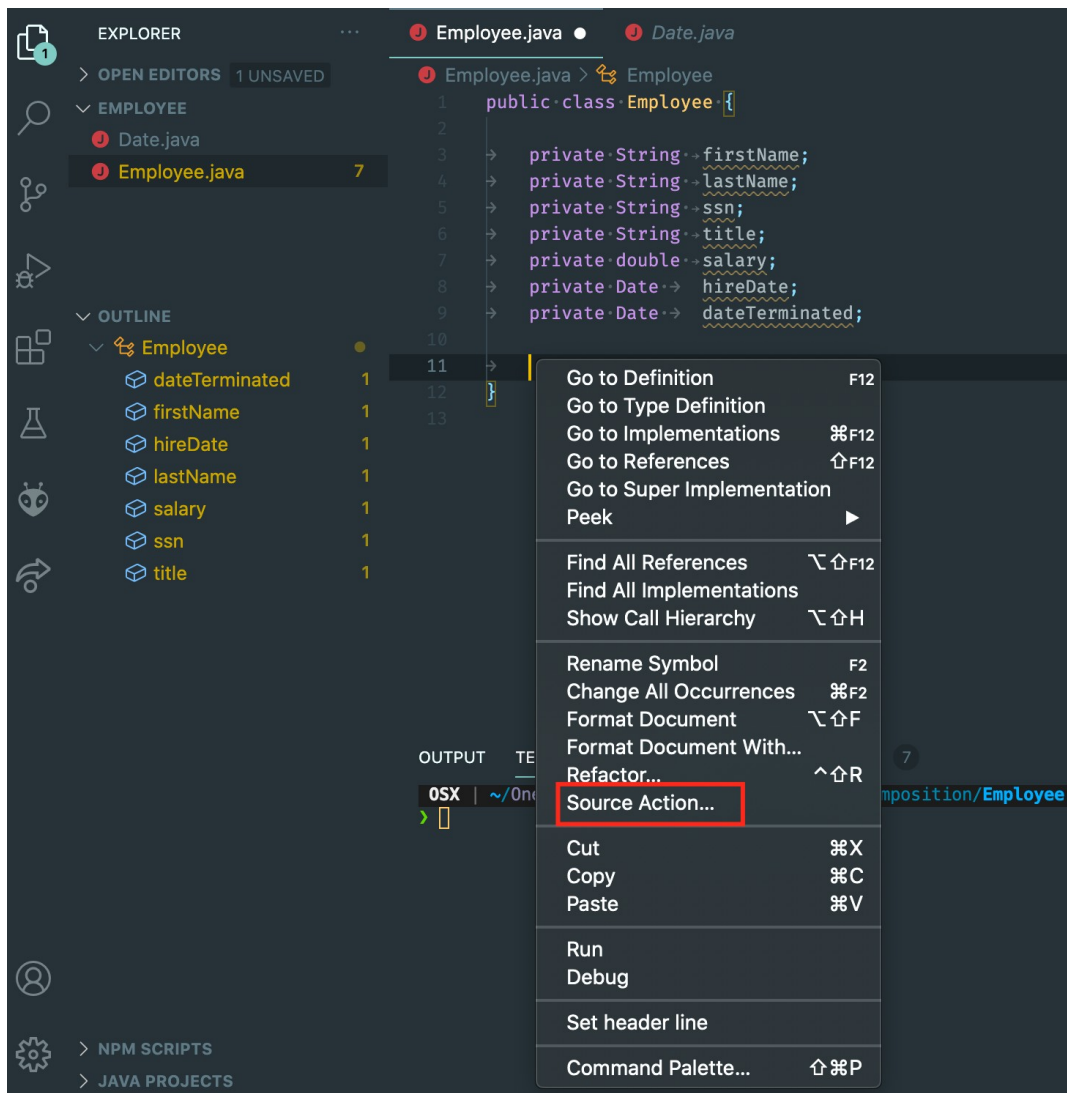
My workspace now has two Java source code files, and you'll notice that **Date.java** is recognized.



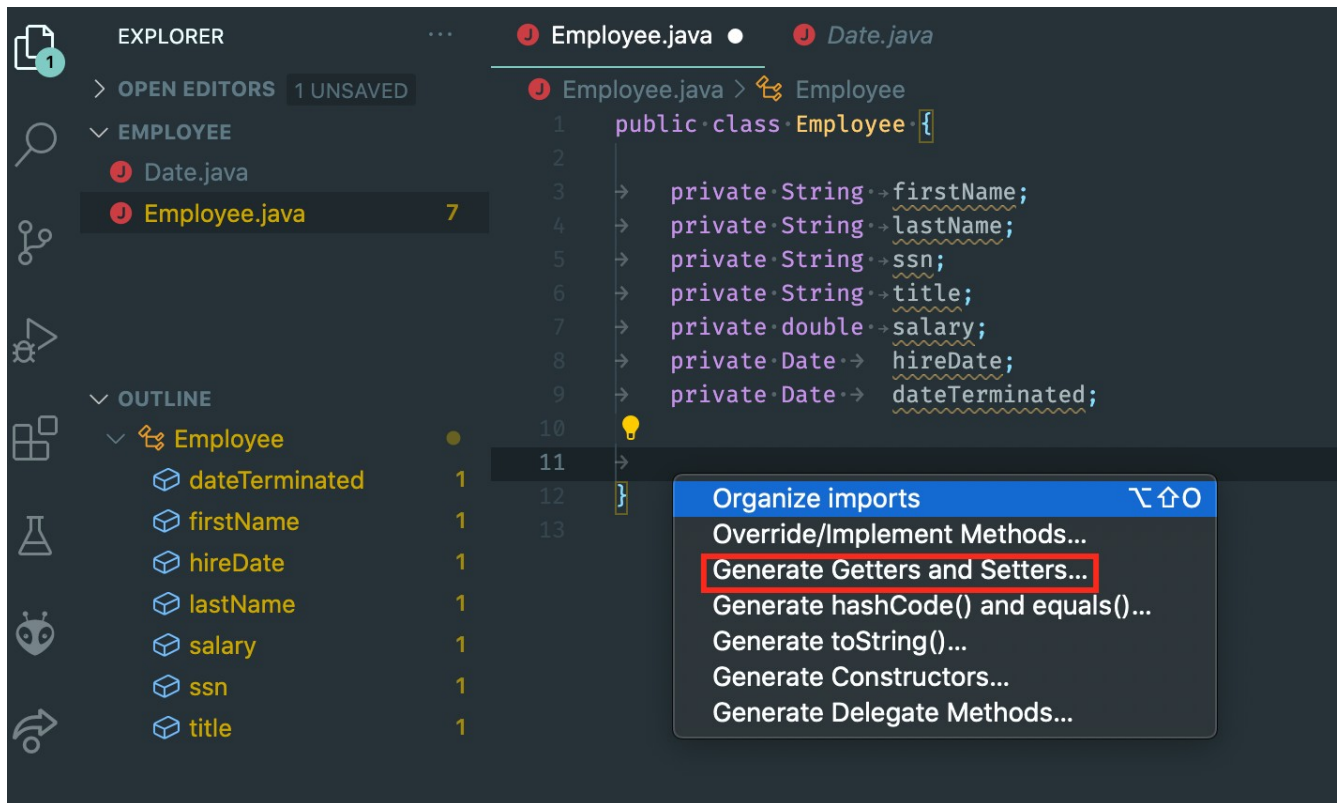
Composition Concept: When dealing with objects that are composed of other objects it is important to understand how Java deals with the references. When defining a method that **returns an object**, Java does not **actually return the object itself**. Instead of returning the object, a **reference to the object** is returned. Java is saying “here is the address of this object, you can now visit this address and manipulate the object”. When defining a method that **accepts an object**, Java does not **actually pass the object itself**. Instead, a **reference to the object** is passed. Passing and returning a single integer (memory address) is more efficient than passing around complete objects (which could potentially be huge). As we are going to see though, this type of argument passing and returning creates unique privacy complications. This concept will be important in upcoming examples and ideas.

VS Code’s cool “Source Action” ability

I will now use VS Code’s **Source Action** to generate the getters and setters. This saves quite of typing and allows us to focus on problem solving. Be sure to set the correct insertion point for the generated getters and setters. You can find the **Source Action** tool by right clicking on **Employee.java**



After clicking **Source Action** you will be shown the various commands available. Choose **Generate Getters and Setters**



I am going to choose all of the fields for getter and setter generation. It is debatable as to whether we would even want a setter for a field like ***Social Security Number***. Just realize that and keep the concept in the back of your mind. Which fields you have getters and setters for are up to your team



NOTE: This is a super handy and time saving ability **BUT** it has the potential to generate insecure code and obviously it does not generate any validation or necessary algorithms. It just gives you the basic stubs and you will need to be careful to go and fill in the important details.

You can also use source action to generate `toString` and overloaded constructors. The constructor generator is a super helpful tool that will allow you to configure the exact number, type and order of parameters to your constructors. Be sure to select an appropriate insertion point for these generated methods. The convention is to have *instance variables* → *constructors* → *getters/setters* and then other methods.

Here are the constructors I generated. Obviously, number and configuration of constructors is up to you and your team. I have kept it simple with a no argument constructor, an overloaded constructor with name and hire date only and a single overloaded constructor that accepts everything.

```
public Employee(){}

public Employee(String firstName, String lastName, Date hireDate){
    → this.firstName → = firstName;
    → this.lastName → = lastName;
    → this.hireDate → = hireDate;
}

public Employee(String firstName, String lastName, String ssn,
    → String title, double salary, Date hireDate,
    → Date dateTerminated){
    → this.firstName → = firstName;
    → this.lastName → = lastName;
    → this.ssn → = ssn;
    → this.title → = title;
    → this.salary → = salary;
    → this.hireDate → = hireDate;
    → this.dateTerminated = dateTerminated;
}
```

You will notice the inclusion of the **@Override** annotation when generating `toString`. Just accept it for now, we will be covering this topic in later modules. The **toString** format that VS Code generates is not the prettiest, so take some time to beautify it a bit. Maybe you don't want to show all fields, maybe there is a required format that you must follow.

```
@Override
public String toString(){
    → return "Employee [dateTerminated="+dateTerminated+", firstName="+firstName+", hireDate="+hireDate
    → +", lastName="+lastName+", salary="+salary+", ssn="+ssn+", title="+title+"]";
}
```

Please use these abilities, but I do expect you to modify the implementation a bit. Especially the `toString`

I do not recommend having VS Code generate your equals method at this point; for two reasons

1. There will be concepts included that you may not be familiar with, including the generation of the hash code method
2. It is good practice for you to design these methods yourself, then we can gradually add these concepts in instead of throwing them all at you at once.

A basic equals() method implementation with composition

Let's now build our equals method. This is a method that we will continue to refine as we progress through the concepts of this course. It ends up being quite complicated, but it does not have to start that way. We can write a functional equals method without following the exact recommendations from the Java engineers (again . . . these contain concepts that we will learn in the future)

The signature of our equals method is: **public boolean equals(Employee otherEmployee)**

Obviously, you can change the parameter identifier to whatever you want. In order for two Employees to be considered *equal* we need to drill through each field of *this Employee* and determine equality with the associated fields in *otherEmployee*. All fields must be equal (or whatever fields we deem as important to equality), or the two Employees are not equal . . . this is a **compound boolean “and” condition**. Because we are working with an object that is designed with composition, we will need to invoke the equals methods on these objects. There are two cases in the Employee class that require this

1. All of the String objects
2. The Date objects (good thing we already defined this!)

The salary field is a primitive, and as such we can use == to determine equality.

This equals method should also contain NULL CHECKS! I left these out so we could focus on the composition aspect of a deep comparison. KEEP THIS IN MIND!! This looks cleaner but it has the ability to throw NullPointerExceptions.

For now, focus on the *chaining together of various equals methods*. Understand that this chaining can go to any depth. This is one of the main reasons that classes are expected to have **equals definitions**. Your classes could be composed of objects that are in turn composed of objects etc, etc, etc . . .

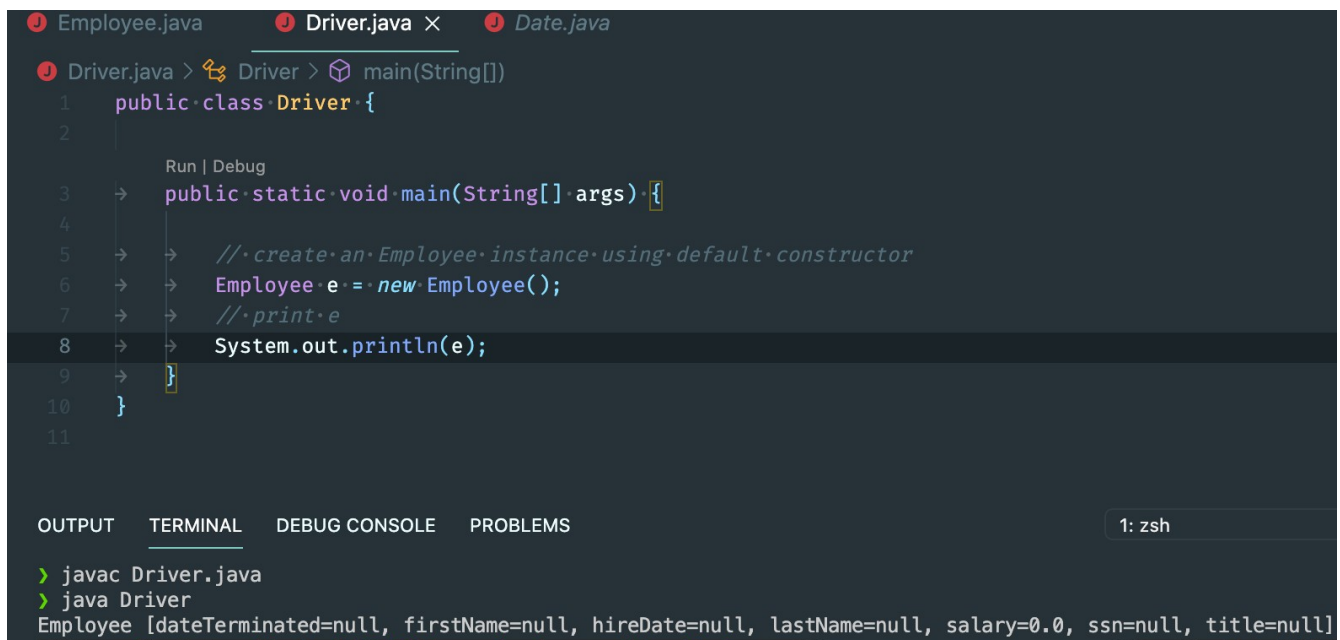
```
public boolean equals(Employee otherEmployee){
→   return this.firstName.equals(otherEmployee.firstName)→  &&
→   →   →   this.lastName.equals(otherEmployee.lastName)→  &&
→   →   →   this.ssn.equals(otherEmployee.ssn)→   →   →   &&
→   →   →   this.title.equals(otherEmployee.title)→   →   →   &&
→   →   →   this.salary == otherEmployee.salary→   →   →   &&
→   →   →   this.hireDate.equals(otherEmployee.hireDate)→   &&
→   →   →   this.dateTerminated.equals(otherEmployee.dateTerminated);
→ }
}
```


Let's now write a Driver to illustrate how to interact with an object designed with composition. There are 3 options for instantiating an Employee object

1. The no argument constructor
2. The constructor with just first name, last name and hire date
3. The constructor with all fields

Let's practice with each.

The no-argument constructor: Creating an Employee with a constructor gives us all nulls for object types and the default 0.0 for the double primitive type. Ensure that you understand why.



```
Employee.java  Driver.java x  Date.java
Driver.java > Driver > main(String[])
1  public class Driver {
2
3      Run | Debug
4      public static void main(String[] args) {
5          // create an Employee instance using default constructor
6          Employee e = new Employee();
7          // print e
8          System.out.println(e);
9      }
10 }
11

OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS
1: zsh
> javac Driver.java
> java Driver
Employee [dateTerminated=null, firstName=null, hireDate=null, lastName=null, salary=0.0, ssn=null, title=null]
```

Be careful with this as we essentially have an Employee instance that is composed of **NullPointers**. This issue is evident if we try to invoke any methods on those objects. For instance, if we try to invoke the equals method with these null pointers our program will crash. I have illustrated this in the following screen shot. The program throws a **NullPointerException** on line 94 of the Employee class. This line is a method invocation on the firstName variable, but firstName is null, therefore the method **does not exist**. Make sure you can read these error messages correctly. The message includes something called a **stack trace**. This is essentially the order in which the methods were invoked. You can see from the message that the process started on **line 13 of Driver.main** which called **Employee.equals**

```
Employee.java Driver.java X Date.java
Driver.java > Driver > main(String[])
Run | Debug
3 public static void main(String[] args) {
4
5     // create an Employee instance using default constructor
6     Employee e = new Employee();
7     // print e
8     System.out.println(e);
9
10    Employee e2 = new Employee();
11
12    // are they equal?
13    System.out.println("Are they equal?" + e.equals(e2));
14 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 1: zsh

```
> javac Driver.java
> java Driver
Employee [dateTerminated=null, firstName=null, hireDate=null, lastName=null, salary=0.0, ssn=null, title=null]
Exception in thread "main" java.lang.NullPointerException
    at Employee.equals(Employee.java:94)
    at Driver.main(Driver.java:13)
```

Let's instantiate some Employees using overloaded constructors. These overloaded constructors have dependencies . . . **objects that need to exist before we can create an Employee.**

1. An Employee created with this constructor: **Employee(String, String, Date)** needs two String objects and a Date object . . . this is composition. There is an explicit dependency in the Employee object's composition. In order to have a valid Employee object you need to have instances of other classes.

```
Employee.java Driver.java X
Driver.java > ...
14
15 // first create a Date object
16 Date hired = new Date(12, 1, 1969);
17 // pass the date object into the Employee constructor
18 Employee e3 = new Employee("Tony", "Iommi", hired);
19 // create Employee with an anonymous Date instance
20 Employee e4 = new Employee("Geezer", "Butler", new Date(6, 6, 1970));
21
22 System.out.println(e3);
23 System.out.println(e4);
24 }
25 }
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2: Java Debug Console +

```
> javac Driver.java
> java Driver
Employee [dateTerminated=null, firstName=Tony, hireDate=12/1/1969, lastName=Iommi, salary=0.0, ssn=null, title=null]
Employee [dateTerminated=null, firstName=Geezer, hireDate=6/6/1970, lastName=Butler, salary=0.0, ssn=null, title=null]
```


IMPORTANT! The chain of events here is crucial to understand.

- On line 16 a Date object is created. The overloaded Date constructor is called.
- On line 18 an Employee object is created. The reference to the Date object is passed as an argument into the Employee constructor. This reference becomes encapsulated in the Employee object *e3*.
- On line 19 the *toString* method belonging to the object *e3* is called.

```
@Override
public String toString(){
→   return "Employee [dateTerminated="+dateTerminated+", firstName="+firstName+", hireDate="+hireDate
→   +", lastName="+lastName+", salary="+salary+", ssn="+ssn+", title="+title+"]";
}
```

- When *e3's* toString is executing, it is calling all of the toString methods for it's fields (expect for any primitive types). The toString methods for each of the String object are called (we did not write those). Then the toString for *our* Date objects *hireDate* and *dateTerminated* is called.

```
public String toString(){
→   return month + "/" + day + "/" + year;
}
```

Take a close look at the constructor invocations on lines 18 and 20. In the first invocation an **explicit Date object** was created on line 16 and passed as an argument to the constructor. The invocation on line 20 uses a technique called an **anonymous object**. An anonymous object is one that is created without an explicit variable reference. In this case the **new** operator calls the constructor and returns the reference to the new object. Instead of storing that reference directly in a variable, it is simply passed as an argument to the constructor. The method takes the argument and assigns it to the correct private instance variable. In this situation, that Date reference will be assigned to the *hireDate* instance variable. This is a clever way to protect privacy as we shall see.

Understand that there are still null pointers with these two instances. You can see that by examining the toString output in the screenshot above. If we were to run the equals method on these objects we could still get null pointer exceptions. Notice that I said **could**. These two instances would actually be compared perfectly because the equals method would **return immediately** upon finding two fields that aren't equal as long as it does not encounter a null pointer first. **Thank you, shortcutting!**

In this situation our equals method **returns false** as soon as it compares the first names. That prevents the execution from hitting the *ssn* field which is null. You have to be very careful when organizing your code and it is crucial to understand how these methods stack on top of each other.

```
Employee.java Driver.java X
Driver.java > Driver > main(String[])
15  →  // first create a Date object
16  →  Date hired = new Date(12, 1, 1969);
17  →  // pass the date object into the Employee constructor
18  →  Employee e3 = new Employee("Tony", "Iommi", hired);
19  →  // create Employee with an anonymous Date instance
20  →  Employee e4 = new Employee("Geezer", "Butler", new Date(6, 6, 1970));
21
22  →  System.out.println(e3);
23  →  System.out.println(e4);
24
25  →  System.out.println("Are they equal?" + e3.equals(e4));
26  →  }

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2 2: Java Debug Console +
> javac Driver.java
> java Driver
Employee [dateTerminated=null, firstName=Tony, hireDate=12/1/1969, lastName=Iommi, salary=0.0, ssn=null, title=null]
Employee [dateTerminated=null, firstName=Geezer, hireDate=6/6/1970, lastName=Butler, salary=0.0, ssn=null, title=null]
Are they equal? false
```

BUT if we happen to have two Employees named Frank Stein, hired on the same day and we create our objects like above we will get `NullPointerException` because the `equals` method would step through the equal fields until it made its way to a null pointer. **These issues can be tricky to track.** Make sure you play close attention!! **Notice here that the `NullPointerException` appears on line 96 now**

```
Employee.java Driver.java X
Driver.java > Driver > main(String[])
23  →  //System.out.println(e4);
24
25  →  //System.out.println("Are they equal?" + e3.equals(e4));
26
27  →  Employee e5 = new Employee("Frank", "Stein", new Date(1, 11, 2021));
28  →  Employee e6 = new Employee("Frank", "Stein", new Date(1, 11, 2021));
29  →  System.out.println("Are they equal?" + e5.equals(e6));
30
31  →  }
32  →  }
33

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 4 2: Java Debug Console v
> javac Driver.java
> java Driver
Exception in thread "main" java.lang.NullPointerException
    at Employee.equals(Employee.java:96)
    at Driver.main(Driver.java:29)
```

Use composition objects to manipulate embedded data. When you have multiple levels of composition in this fashion you access data via method calls. These method calls can be stacked to infinite levels. The following example illustrates two levels of composition

Level One: The Employee object

Level Two: The internal Date object representing the “hire date”

```
20 Employee e4 = new Employee("Geezer", "Butler", new Date(6, 6, 1970));
21
22 // What year was Geezer Butler hired?
23 Date geezer_hire = e4.getHireDate(); // returns a reference to e4's hire date
24 int year = geezer_hire.getYear();    // use local reference to get the year
25 System.out.println("Geezer was hired in " + year);
26
27 // Do it all at once by chaining method calls
28 // Because Java returns references, these can be chained to any length
29 // It's not a great idea to get too lengthy with these chains
30 // as they are difficult to debug
31 int y = e4.getHireDate().getYear();
32 System.out.println("Geezer was hired in " + y);
33
```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL

> java Driver
Geezer was hired in 1970
Geezer was hired in 1970

And finally, let’s instantiate some instances using the constructor that allows all field values to be passed it.

```
Employee.java Driver.java x
Driver.java > Driver > main(String[])
31 → → Date d = new Date(1, 11, 2021);
32 → → Employee e7 = new Employee("Ozzy", "Osborne", "123-45-6789", "jr. developer",
33 → → → → → 45000, d, null);
34 → → Employee e8 = new Employee("Bill", "Ward", "567-83-9283", "sr. developer",
35 → → → → → 90000, new Date(1, 6, 2020), null);
36 → → System.out.println(e7);
37 → → System.out.println(e8);
38 → → }
39 }
40
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 6

2: Java Debug Console

> javac Driver.java
> java Driver
Employee [dateTerminated=null, firstName=Ozzy, hireDate=1/11/2021, lastName=Osborne, salary=45000.0, ssn=123-45-6789, title=jr. developer]
Employee [dateTerminated=null, firstName=Bill, hireDate=1/6/2020, lastName=Ward, salary=90000.0, ssn=567-83-9283, title=sr. developer]

Design Issues

How should we handle the **dateTerminated** field? If the employee still works for the company then logically there is **no termination date** so it makes sense to leave that field null, but as we have seen there could be `NullPointerException`s thrown in the `equals` method. Maybe your team feels that having the **dateTerminated** field be equal to the **dateHired** field, thus removing the potential for a null pointer. Due to these issues and the relevancy of including `dateTerminated` in the `equals` method, I have decided to remove that field from comparisons. My `equals` method now looks like this.

```
public boolean equals(Employee otherEmployee){  
→   return this.firstName.equals(otherEmployee.firstName)→  &&  
→   →   →   this.lastName.equals(otherEmployee.lastName)→  &&  
→   →   →   this.ssn.equals(otherEmployee.ssn)→   →   →   &&  
→   →   →   this.title.equals(otherEmployee.title)→   →   →   &&  
→   →   →   this.salary == otherEmployee.salary→   →   →   &&  
→   →   →   this.hireDate.equals(otherEmployee.hireDate);  
→   →   →  
}
```

There is a constant give and take and refinement process that happens to our objects. Note that another valid approach for this would be to have the termination date equal the hire date . . . then it would not be null, and we could make the assertion that when these dates are equal, the Employee has not been terminated.

Class Invariant

An invariant is a situation that must always hold true no matter what. If we identify any invariants during our problem-solving phase, we must make sure that they are enforced throughout the lifespan of the application. A good example of an invariant with this Employee class is that the **dateTerminated** field **must always** have a value that is greater than the **hireDate** field. In other words, our code needs to guarantee that the hired dates come before the termination dates, otherwise our object could end up in an illogical state. We would not want to have an Employee in our records who was hired on 1/4/2020 and terminated on 4/16/2018. How can we enforce this?

Privacy Leaks

A common programming error when dealing with composition is the privacy leak of encapsulated data. This is of particular interest when dealing with getters and setters of composite object types. Colloquially we can define a privacy leak as

When somebody **outside** of our class gets a copy of an object reference meant to be secured privately **inside** the class.

Specifically, we would say that a privacy leak exists when

A consumer of a Java class gains the ability to modify the internal values of a private attribute

We encapsulate classes with private attributes to

- Enforce a consistent state with invariants
- Make objects of the class immutable if needed

A privacy leak will break these features, rendering the **private access modifier** useless. Let's demonstrate how we can break the privacy protections on our Employee class.

Employee Invariant: The terminated date must come after the hired date

We address this invariant in the following ways.

1. **Access Modifier:** Defined private access to both *hireDate* and *dateTerminated* fields in the Employee class
2. **Object Ordering:** Wrote a *compareTo* method in the Date class to specify ordering
3. **Validation:** Included calls to *compareTo* in our Date setters in Employee to ensure this relationship holds.
4. **Testing:** We wrote unit tests for these methods to ensure that they function correctly.

References to these invariant protections follow. You can examine them in their working state by inspecting the source code files in the repo.

Access Modifiers

```
1 public class Employee {
2
3     private String firstName;    // composition: using a String object
4     private String lastName;    // composition: using a String object
5     private String ssn;         // composition: using a String object
6     private String title;       // composition: using a String object
7     private double salary;      // NO COMPOSITION: primitive
8     private Date hireDate;      // composition: using a Date object
9     private Date dateTerminated; // composition: using a Date object
10 }
```

Date class compareTo to use for ensuring Dates are ordered correctly.

```
public int compareTo(Date otherDate){
    // begin with the year
    if (this.year < otherDate.year)    return -1;
    if (this.year > otherDate.year)    return 1;

    // years are equal, test the month
    if (this.month < otherDate.month)  return -1;
    if (this.month > otherDate.month)  return 1;

    // both year and month are equal, test days
    if (this.day < otherDate.day)      return -1;
    if (this.day > otherDate.day)      return 1;

    // all fields must be equal
    return 0;
}
```

Implementation of setDateTerminated in the Employee class

Composition Task: Call the compareTo method defined in the Date class to enforce invariant

```
public void setDateTerminated(Date dateTerminated) {
    // do some validation testing on the argument
    // Policy: the date terminated must be greater than the hire date
    if (dateTerminated != null && this.hireDate.compareTo(dateTerminated) < 0)
        this.dateTerminated = dateTerminated;
}
```

Surely that is sufficient? No. Our Employee class has privacy leaks in a variety of areas. This is because IDE's automatically generates getters/setters with privacy leaks.

Let's break it! (The remaining screen shots were made using Eclipse. It looks a little different)


```
1 public class Driver {
2
3     public static void main(String[] args) {
4
5         Date hired = new Date(1, 4, 2020);
6         Employee emp1 = new Employee("Frank", "Stein", "123-45-6789", "junior developer",
7                                     45000, hired, null);
8
9         emp1.setDateTerminated(new Date(12, 3, 2020));
10        System.out.println(emp1);
11
12        Date d = emp1.getDateTerminated(); ← This method returns a private reference
13
14        d.setYear(1996); ← Use local copy of leaked reference to bypass privacy
15
16        System.out.println("\n" + emp1);
17    }
18 }
```

Problems Javadoc Declaration Console x

<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 5, 2020, 1:27:52 PM)

Employee [firstName=Frank, lastName=Stein, ssn=123-45-6789, title=junior developer, salary=45000.0, hireDate=1/4/2020, dateTerminated=12/3/2020]

Employee [firstName=Frank, lastName=Stein, ssn=123-45-6789, title=junior developer, salary=45000.0, hireDate=1/4/2020, dateTerminated=12/3/1996]

Code Summary

1. Created a Date instance on line 6
2. Passed that Date instance into the Employee constructor on line 7
3. Set a valid termination date on line 10
4. Printed the employee to see that the termination date took . . . it did.
5. Requested the employee's termination date via a call to the getter. The getter returned a **private reference (PRIVACY LEAK!!)**
6. Exploited the privacy leak by manipulating the returned Date object

Privacy Leak Two

```
Employee.java Date.java *Driver.java x DateTest.java
1 public class Driver {
2
3     public static void main(String[] args) {
4
5         Date hired = new Date(1, 4, 2020); ← Local reference to a Date object
6
7         Employee emp1 = new Employee("Frank", "Stein", "123-45-6789", "junior developer",
8                                     45000, hired, null); ← Pass local reference into object
9
10        System.out.println("\n" + emp1);
11
12        hired.setYear(1996); ← Exploit privacy leak by using local reference
13
14        System.out.println("\n" + emp1);
15
16    }
17 }
```

Problems Javadoc Declaration Console x

<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 5, 2020, 2:52:14 PM)

Employee [firstName=Frank, lastName=Stein, ssn=123-45-6789, title=junior developer, salary=45000.0, hireDate=1/4/2020, dateTerminated=null]

Employee [firstName=Frank, lastName=Stein, ssn=123-45-6789, title=junior developer, salary=45000.0, hireDate=1/4/1996, dateTerminated=null]

Code Summary

1. Create a local Date instance on line 5. The reference returned by **new** is local to the main method
2. Create an Employee instance, using the reference to the Date created on line 5. Java passes objects **by value** so it is the reference to the Date object that is passed. This reference is copied into the private Employee variable **hireDate**. **(PRIVACY LEAK!!)** There are now two references to that single Date object
 1. One local to Driver.main
 2. One local to Employee
3. Using the saved reference to the **hired** object we can bypass the privacy protection by invoking methods on this object (line 12)
4. **Note:** Had we used an anonymous reference to construct the Employee instance (line 7) this problem could be avoided . . . this does not change the fact that a privacy leak exists.

Privacy Leak Best Practice

1. Understand the relationship between a reference and an object
2. When an object is passed into or returned from a method, it is the reference that is being moved around
3. Whenever you write a public method to return a private encapsulated object, be aware that privacy could be leaked because you are returning the address, not the object itself.
4. Whenever you write a public method to accept a reference to an object, be aware that that address could have a local copy stored in a variable elsewhere. That variable could then be used to bypass privacy and manipulate the object directly.
5. Only mutable objects are affected by this situation
6. String objects do not suffer from privacy leaks because **Strings are immutable**

Solution

We cannot rely on class consumers to carefully safeguard privacy. For the classes we write, it is our responsibility. If you need to write methods that either accept or return objects, and privacy is a concern, then the best approach is to return a **copy of the object**. Copying the object allocates new memory and provides us with a new reference. This allows us to manage which references are allowed in and out of our encapsulated classes. The copied object is considered a clone.

Cloning Objects Using Copy Constructors

Cloning objects in Java is a contentious subject, with many differing opinions on best practices. The standard convention requires some material that we have not yet covered, but we can provide a workable solution to our current issue. This solution comes in the form of a special type of constructor called the copy constructor.

Copy Constructor: Constructor that accepts an instance of its own class to clone. Performs a deep copy on each field. Returns these copies to the new instance.

Here is the copy constructor definition for the Date class. Notice how values of each of the fields are copied over to the new instance **this**. This copy does not go any deeper because each of these fields are of primitive type and their values will simply be returned. If there were other mutable objects in these fields the cloning would need to recurse.

```
public Date(Date copy) {  
    this.month = copy.month;  
    this.day   = copy.day;  
    this.year  = copy.year;  
}
```

Clone Test: To verify a proper clone you should test for the following

- The clone should have a different reference (Identity check: **clone != og**)
- The clone should be logical identical (State check: **clone.equals(og) == true**, or **clone.compareTo(og) == 0**)

Let's write a Unit Test to verify this

```
@Test
void testClone() {
    // clone instance
    Date clone = new Date(d1);
    // verify identity
    assertFalse(clone == d1);
    // verify state
    assertTrue(clone.equals(d1));
    assertEquals(clone.compareTo(d1), 0);
}
```

Copy Constructor Application: If there is a class invariant that needs to be held, and this invariant includes accepting and returning object references then you should clone these references

- Before returning a private reference from a public method
- Before setting a private reference to an argument

Let's fix the privacy leaks in our Employee class by creating clones for each of the cases listed above. Cloning should be implemented wherever there is a private instance variable being returned from a public method, or an external reference being passed as an argument through a public method, that would then be set to a private field. For the Employee class this includes

- The constructors
- The get/set methods for the Date fields.

```
public Date getHireDate() {
    // return a clone of the private data
    return new Date(hireDate);
}

public void setHireDate(Date hireDate) {
    // create a local cloned instance to set
    this.hireDate = new Date(hireDate);
}

public Date getDateTerminated() {
    return dateTerminated != null ? new Date(dateTerminated) : null;
}

public void setDateTerminated(Date dateTerminated) {
    this.dateTerminated = this.hireDate.compareTo(dateTerminated) == -1 ? new Date(dateTerminated) : hireDate;
}
```

Now we can route constructor arguments through these methods

```

public Employee(String firstName, String lastName, String ssn, String title,
                double salary, Date hireDate, Date dateTerminated) {
    super();
    this.firstName    = firstName;
    this.lastName     = lastName;
    this.ssn          = ssn;
    this.title        = title;
    this.salary        = salary;
    this.setHireDate(hireDate);
    this.setDateTerminated(dateTerminated);
}

public Employee(String firstName, String lastName, Date hireDate) {
    super();
    this.firstName    = firstName;
    this.lastName     = lastName;
    this.setHireDate(hireDate);
}

```

Let's Unit Test:

```

1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.BeforeEach;
3 import org.junit.jupiter.api.Test;
4
5 class EmployeeTest {
6
7     Employee employee;
8     Date hired;
9     Date terminated;
10
11     @BeforeEach
12     void setUp() throws Exception {
13         hired      = new Date(1, 1, 2000);
14         terminated  = new Date(1, 1, 1999);
15     }
16
17     @Test
18     void testEmployeeStringStringStringStringDoubleDateDate() {
19         employee    = new Employee( "Frank", "Stein", "666-33-1234",
20                                     "Monster", 12345.67, hired,
21                                     terminated);
22
23         Date hired      = employee.getHireDate();
24         Date terminated  = employee.getDateTerminated();
25
26         // domain validation
27         assertTrue(hired.equals(terminated));
28
29         // cloning
30         Date privateHired = employee.getHireDate();
31         assertFalse(privateHired == hired);
32         assertTrue(privateHired.equals(hired));
33
34         Date privateTerminated = employee.getDateTerminated();
35         assertFalse(privateTerminated == terminated);
36         assertTrue(privateTerminated.equals(terminated));
37     }
38 }
39

```

Notes on testing clones

- **Identity Check:** The cloned object's reference **must not equal** the reference of the original object. You can see this test above as **`assertFalse(privateTerminated == terminated)`**
- **State Check:** `clone.equals(original)` **must be true**. The original object and the cloned object's state must be identical. You can see this test above as **`assertTrue(privateTerminated.equals(terminated));`**