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 step-wise 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.

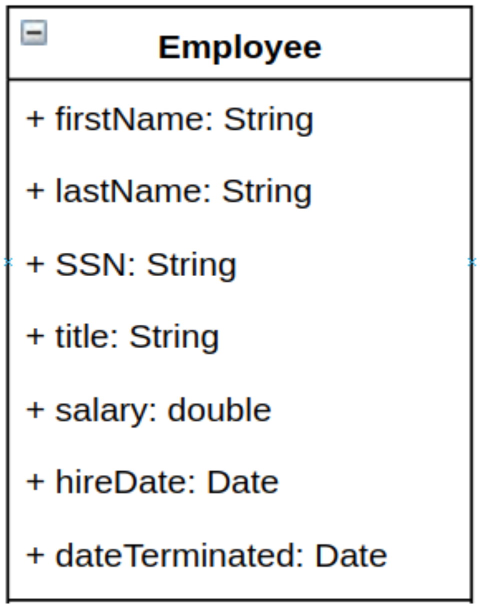
From here on out I will be using the Eclipse IDE to manage my source code.

The Employee Class

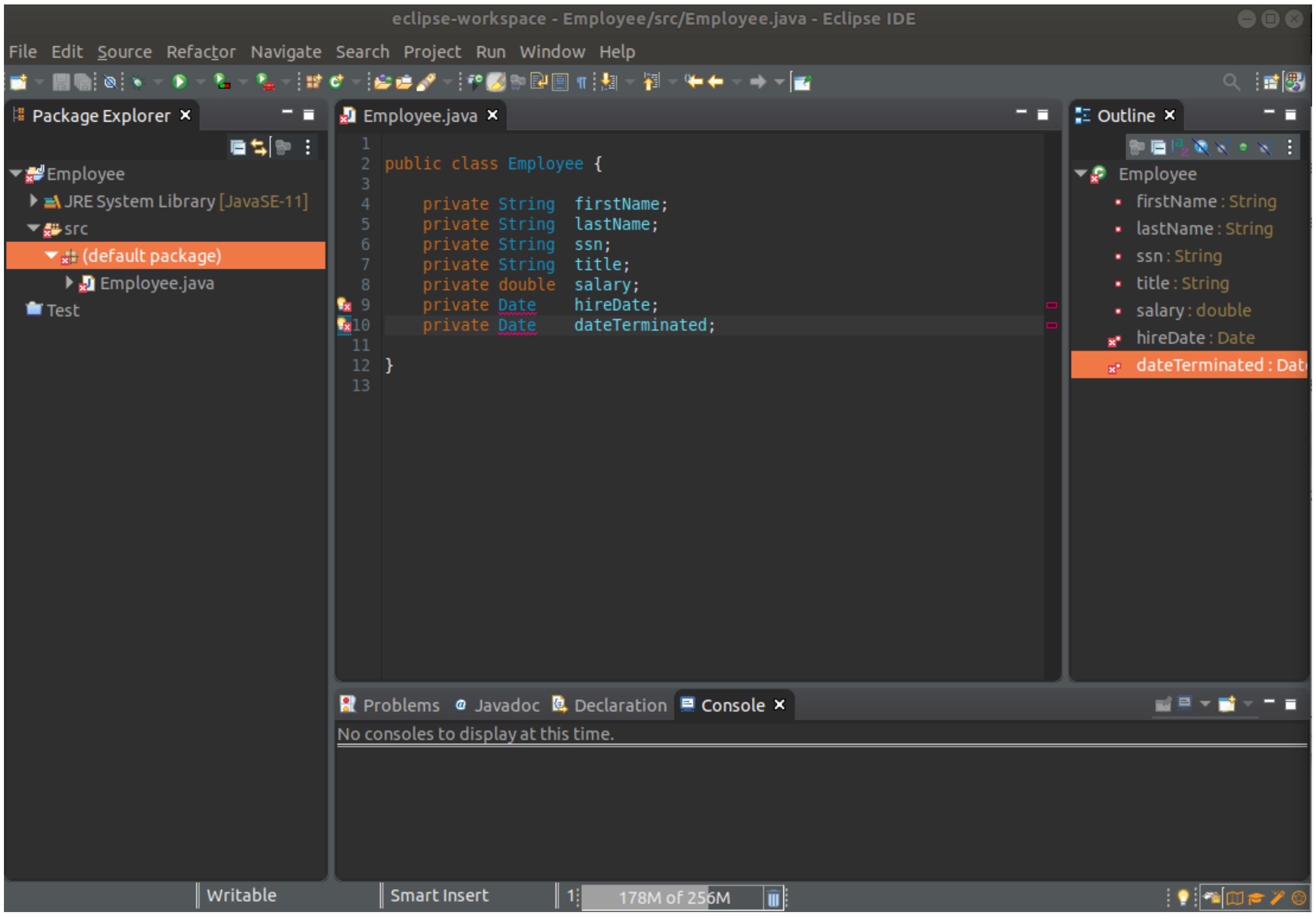
Let's start with a simple class to define an Employee. This class could be used in payroll processing, to associate an employee with a task

- 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.



I have created a new Project in Eclipse and added a new class called Employee



```java

public class Employee {

    private String  firstName;
    private String  lastName;
    private String  ssn;
    private String  title;
    private double  salary;
    private Date    hireDate;
    private Date    dateTerminated;

}

```

No consoles to display at this time.

You'll notice that the references to our custom Date class are not recognized. We will need to import this class into our project. The import routine can be found in the File menu. Remember that Eclipse copies files into its workspace when you perform an import. Any changes made to any of these classes will be saved in those workspace files.

I will now use Eclipse's ***Source control menu*** to generate the getters and setters. Be sure to set the correct insertion point for the generated getters and setters and you should also check the **Generate Comments** box; this will provide you with some JavaDoc comment stubs. Update these to your liking.

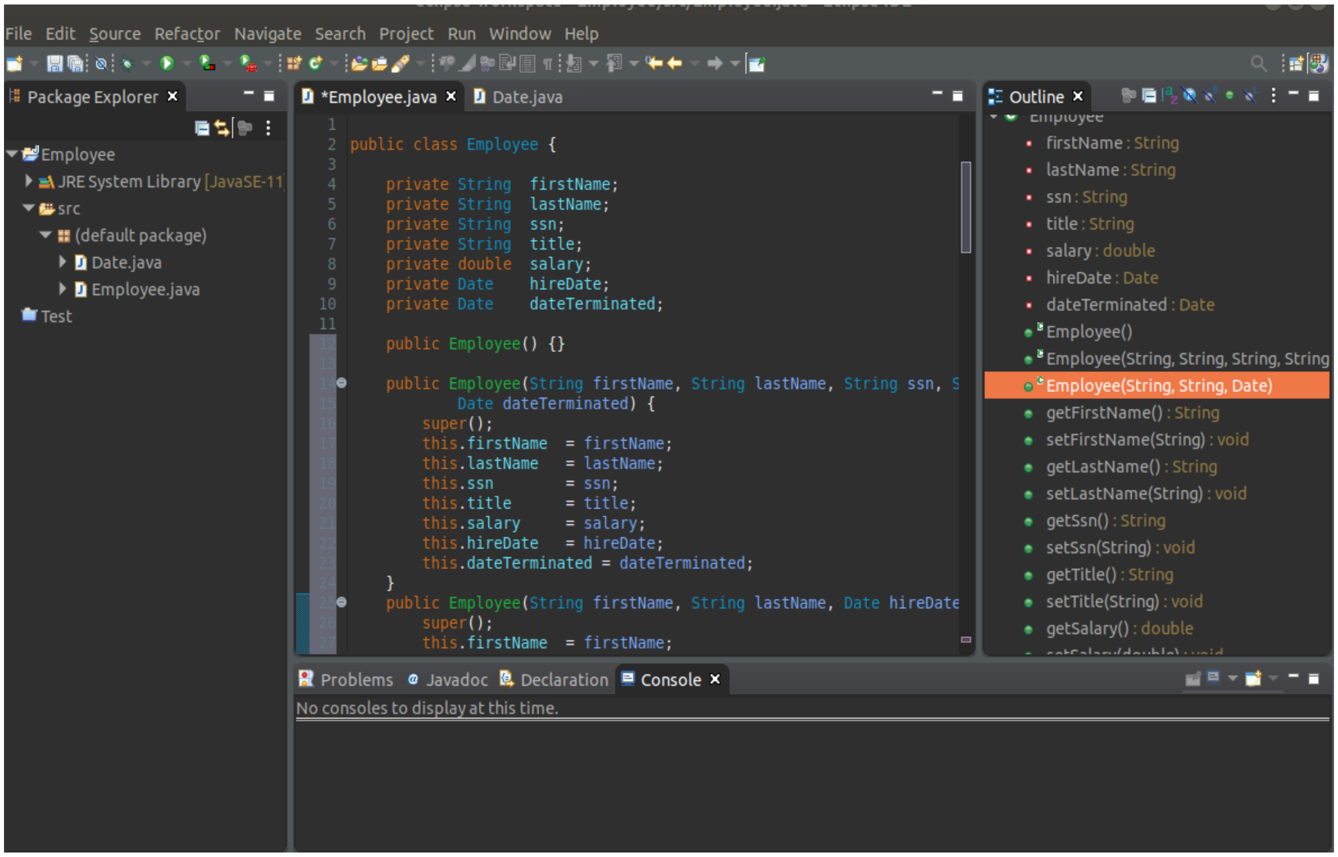
You can also use Eclipse's source control to generate toString and overloaded constructor. The constructor generator is a super helpful tool that will allow you to configure the exact number, type and order of parameters to your constructors. Be sure to select an appropriate insertion point for these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

I do not recommend having Eclipse 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.

The Eclipse interface now contains our project info including . . .

- The Java version
- All of our current source code files
- Outlines of our class structures
- As well as a tabbed editor for managing our multiple source files.



```
public class Employee {

    private String  firstName;
    private String  lastName;
    private String  ssn;
    private String  title;
    private double  salary;
    private Date    hireDate;
    private Date    dateTerminated;

    public Employee() {}

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

```
Employee
    firstName : String
    lastName : String
    ssn : String
    title : String
    salary : double
    hireDate : Date
    dateTerminated : Date
    Employee()
    Employee(String, String, String, String
    Employee(String, String, Date)
    getFirstName() : String
    setFirstName(String) : void
    getLastName() : String
    setLastName(String) : void
    getSsn() : String
    setSsn(String) : void
    getTitle() : String
    setTitle(String) : void
    getSalary() : double
```

```
No consoles to display at this time.
```

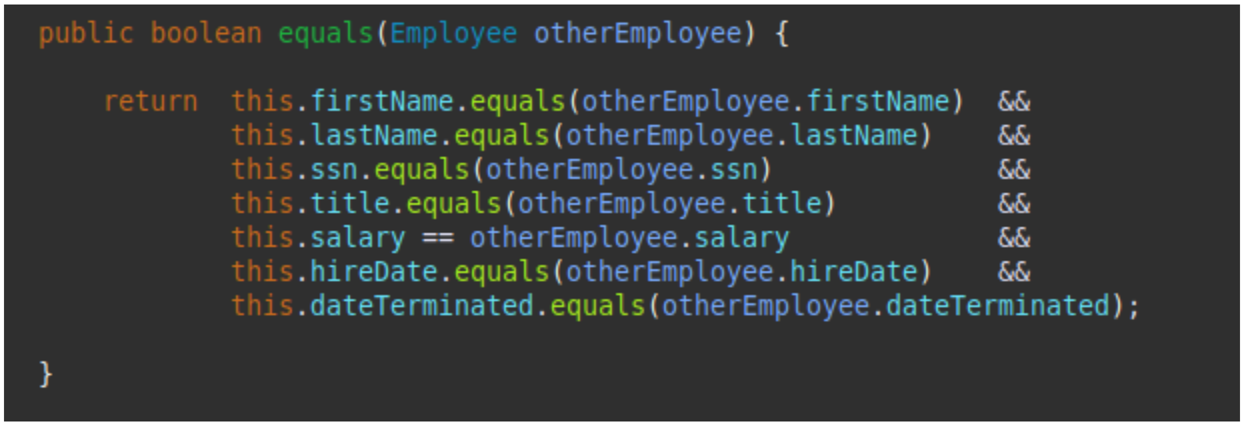
Let's now build our equals method. This is a method that we will continue to refine as we progress through the concepts of this course. It ends up being quite complicated, but it does not have to start that way.

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 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!)

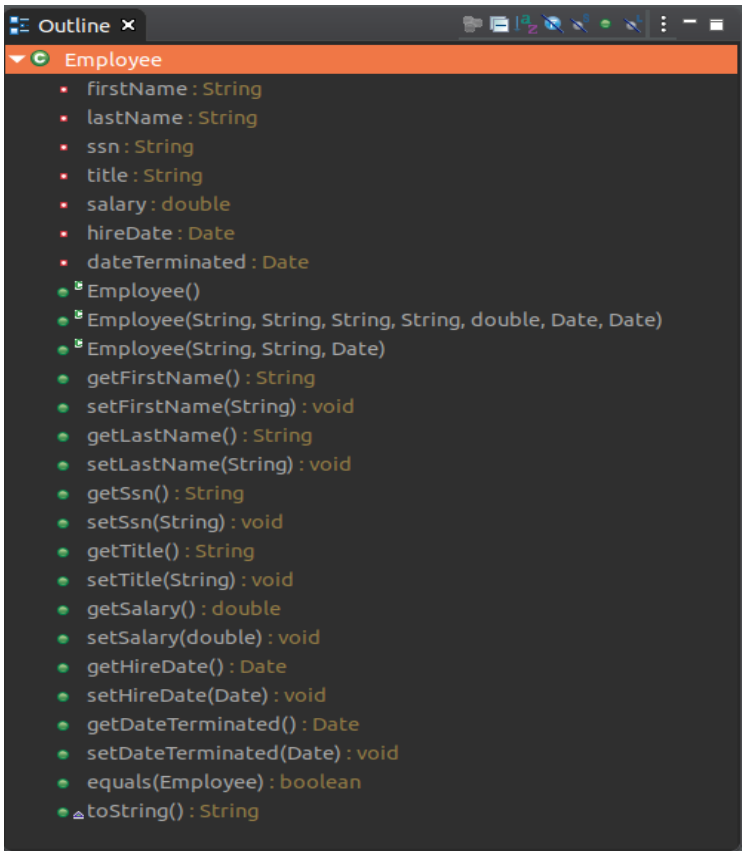
The salary field is a primitive, and as such we can use == to determine equality.



```java
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);

}
```

You may find that the Eclipse generated **toString** is not what you are looking for aesthetically. Feel free to make this more attractive. In this case I accepted the generated toString.

Here is the current class outline after completing all of the described actions



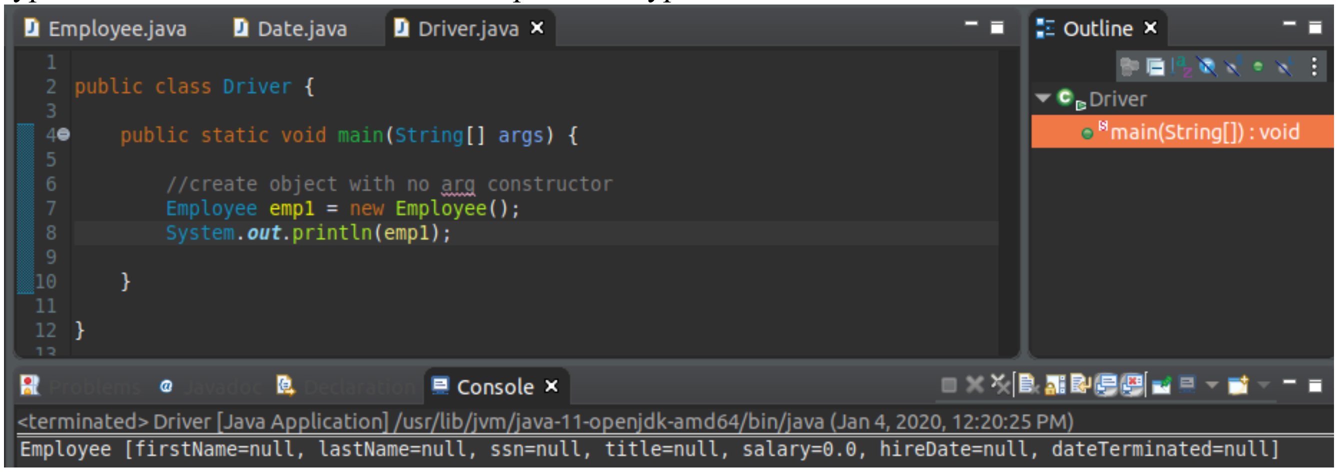
```
Outline
Employee
    firstName : String
    lastName : String
    ssn : String
    title : String
    salary : double
    hireDate : Date
    dateTerminated : Date
    Employee()
    Employee(String, String, String, String, double, Date, Date)
    Employee(String, String, Date)
    getFirstName() : String
    setFirstName(String) : void
    getLastName() : String
    setLastName(String) : void
    getSsn() : String
    setSsn(String) : void
    getTitle() : String
    setTitle(String) : void
    getSalary() : double
    setSalary(double) : void
    getHireDate() : Date
    setHireDate(Date) : void
    getDateTerminated() : Date
    setDateTerminated(Date) : void
    equals(Employee) : boolean
    toString() : String
```

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 constructor gives us all nulls for object types and the default 0.0 for the double primitive type



Employee.java   Date.java   Driver.java

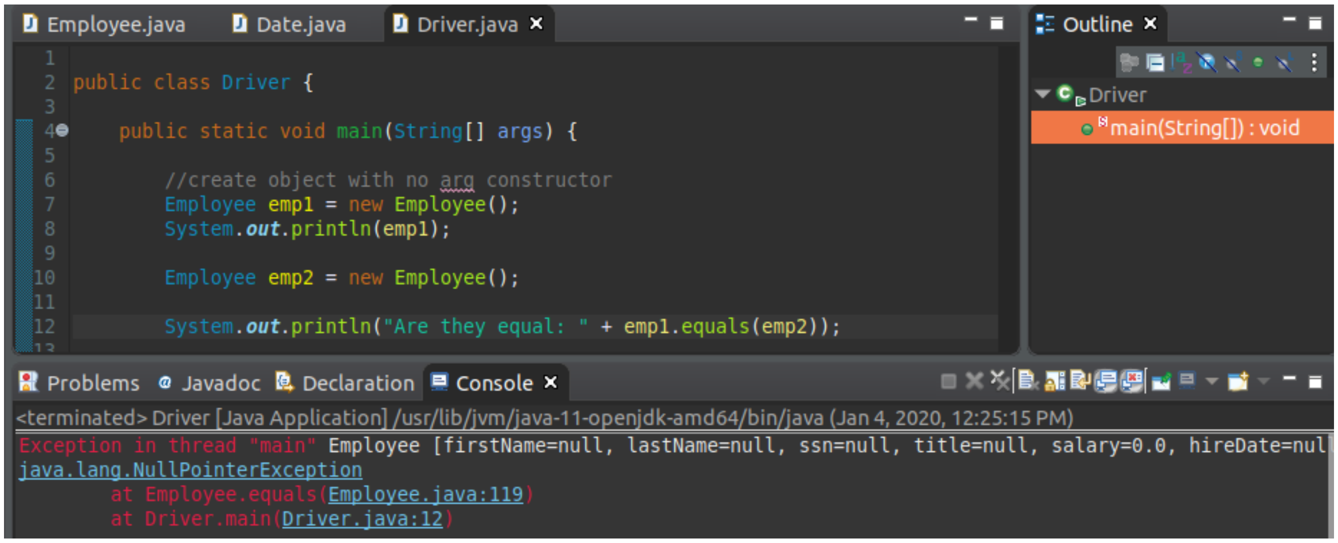
```
 1
 2 public class Driver {
 3
 4     public static void main(String[] args) {
 5
 6         //create object with no arg constructor
 7         Employee emp1 = new Employee();
 8         System.out.println(emp1);
 9
10     }
11
12 }
```

Outline: Driver — main(String[]) : void

Console

```
<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 12:20:25 PM)
Employee [firstName=null, lastName=null, ssn=null, title=null, salary=0.0, hireDate=null, dateTerminated=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 119 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 12 of Driver.main** which called **Employee.equals**



Employee.java   Date.java   Driver.java

```
 1
 2 public class Driver {
 3
 4     public static void main(String[] args) {
 5
 6         //create object with no arg constructor
 7         Employee emp1 = new Employee();
 8         System.out.println(emp1);
 9
10         Employee emp2 = new Employee();
11
12         System.out.println("Are they equal:  " + emp1.equals(emp2));
```

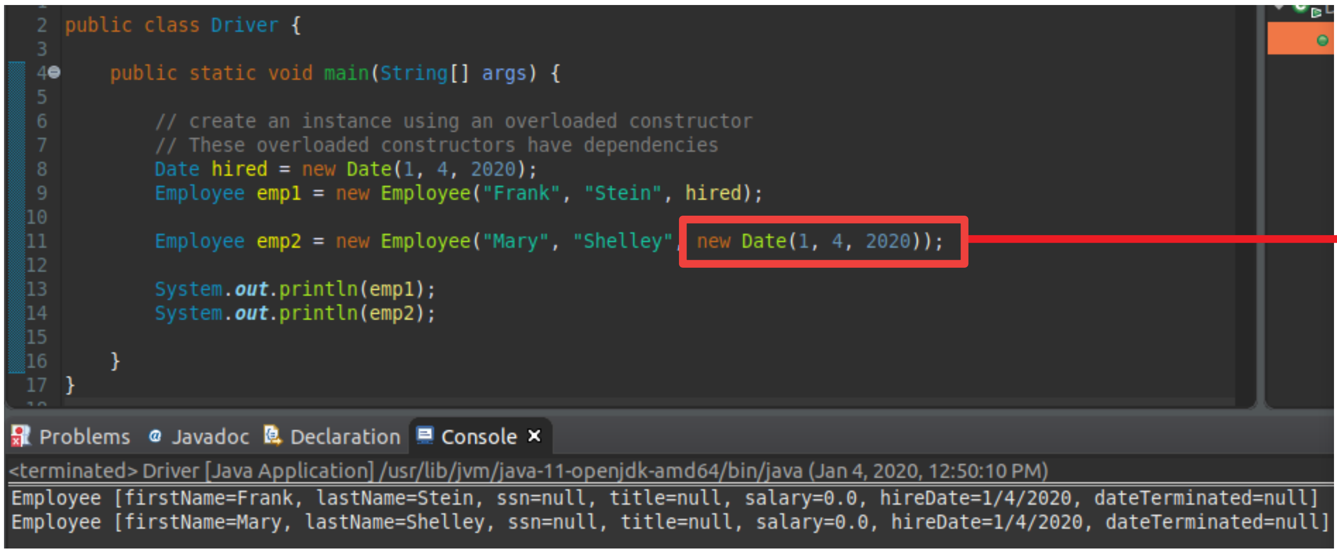
Outline: Driver — main(String[]) : void

Problems   Javadoc   Declaration   Console

```
<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 12:25:15 PM)
Exception in thread "main" Employee [firstName=null, lastName=null, ssn=null, title=null, salary=0.0, hireDate=nul
java.lang.NullPointerException
        at Employee.equals(Employee.java:119)
        at Driver.main(Driver.java:12)
```

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

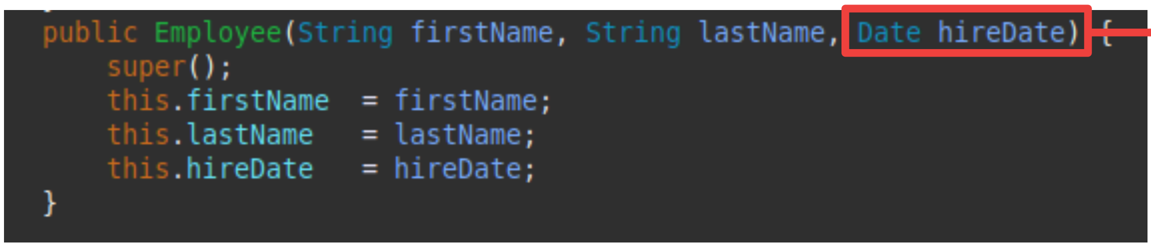
1. An Employee created with this constructor: **Employee(String, String, Date)** needs two String objects and a Date object . . . this is composition.
2. An Employee created with this constructor: **Employee(String, String, String, String, double, Date, Date)** needs 4 String objects and two Date objects



```
 2 public class Driver {
 3 
 4⊖     public static void main(String[] args) {
 5 
 6         // create an instance using an overloaded constructor
 7         // These overloaded constructors have dependencies
 8         Date hired = new Date(1, 4, 2020);
 9         Employee emp1 = new Employee("Frank", "Stein", hired);
10 
11         Employee emp2 = new Employee("Mary", "Shelley", new Date(1, 4, 2020));
12 
13         System.out.println(emp1);
14         System.out.println(emp2);
15 
16     }
17 }
```

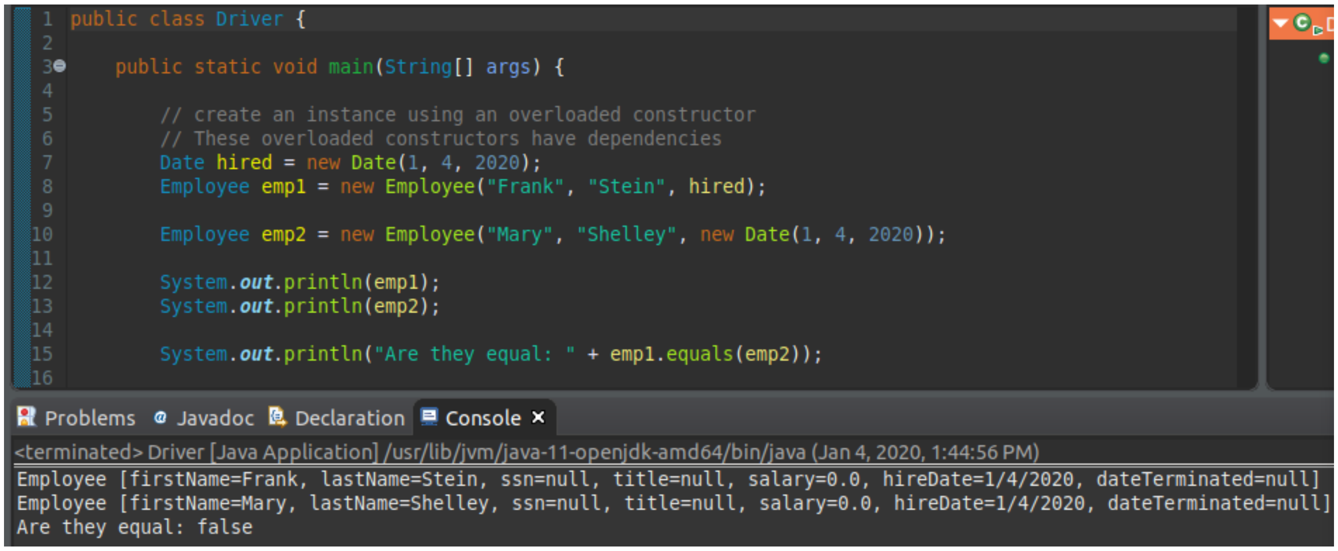
```
<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 12:50:10 PM)
Employee [firstName=Frank, lastName=Stein, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=null]
Employee [firstName=Mary, lastName=Shelley, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=null]
```

Take a close look at the constructor invocations on lines 9 and 11. In the first invocation an explicit Date object was created on line 8 and passed as an argument to the constructor. The invocation on line 11 uses a technique called an ***anonymous object.*** An anonymous object is one that is created without an explicit 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 method. 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.



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

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. **Thank you, shortcutting!**

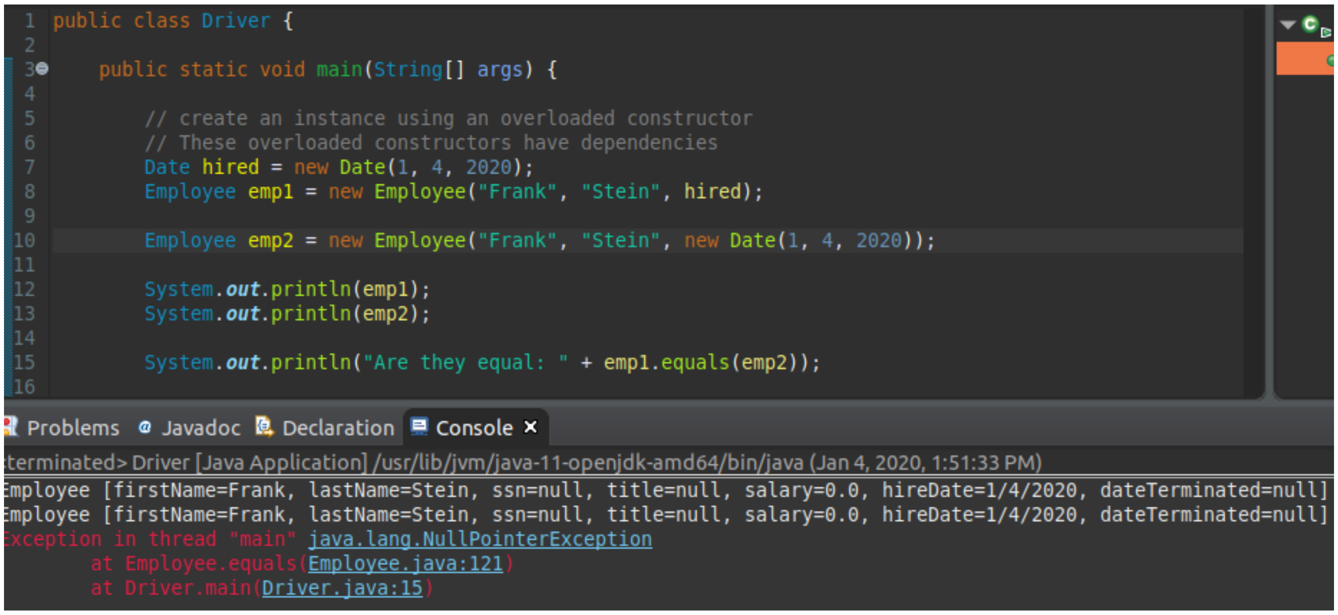


```java
public class Driver {

    public static void main(String[] args) {

        // create an instance using an overloaded constructor
        // These overloaded constructors have dependencies
        Date hired = new Date(1, 4, 2020);
        Employee emp1 = new Employee("Frank", "Stein", hired);

        Employee emp2 = new Employee("Mary", "Shelley", new Date(1, 4, 2020));

        System.out.println(emp1);
        System.out.println(emp2);

        System.out.println("Are they equal: " + emp1.equals(emp2));

```

Problems Javadoc Declaration Console

```
<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 1:44:56 PM)
Employee [firstName=Frank, lastName=Stein, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=null]
Employee [firstName=Mary, lastName=Shelley, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=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 NullPointerExceptions 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 121 now**

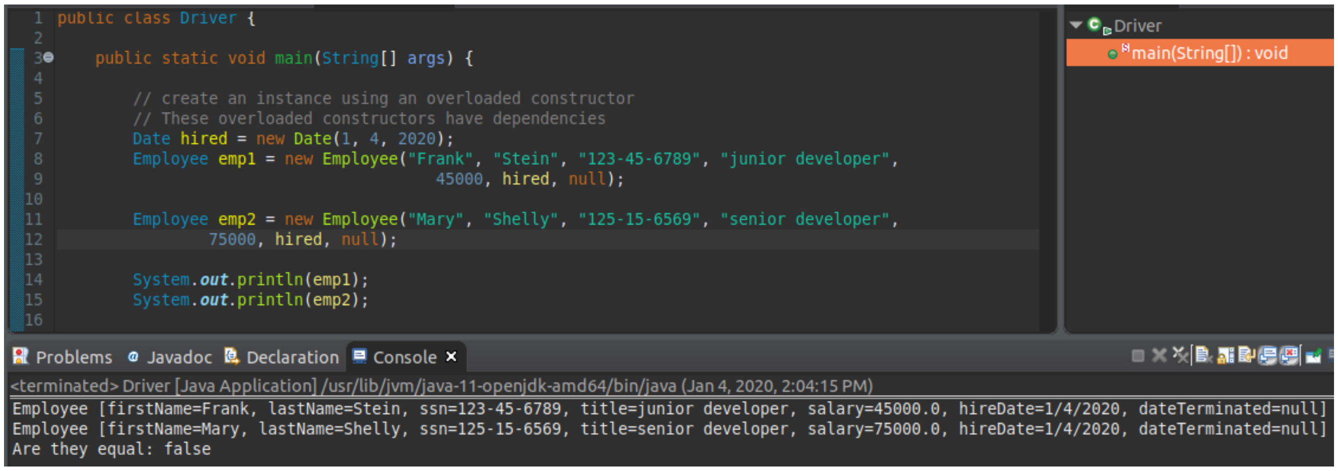


```java
public class Driver {

    public static void main(String[] args) {

        // create an instance using an overloaded constructor
        // These overloaded constructors have dependencies
        Date hired = new Date(1, 4, 2020);
        Employee emp1 = new Employee("Frank", "Stein", hired);

        Employee emp2 = new Employee("Frank", "Stein", new Date(1, 4, 2020));

        System.out.println(emp1);
        System.out.println(emp2);

        System.out.println("Are they equal: " + emp1.equals(emp2));

```

Problems Javadoc Declaration Console

```
terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 1:51:33 PM)
mployee [firstName=Frank, lastName=Stein, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=null]
mployee [firstName=Frank, lastName=Stein, ssn=null, title=null, salary=0.0, hireDate=1/4/2020, dateTerminated=null]
xception in thread "main" java.lang.NullPointerException
        at Employee.equals(Employee.java:121)
        at Driver.main(Driver.java:15)
```

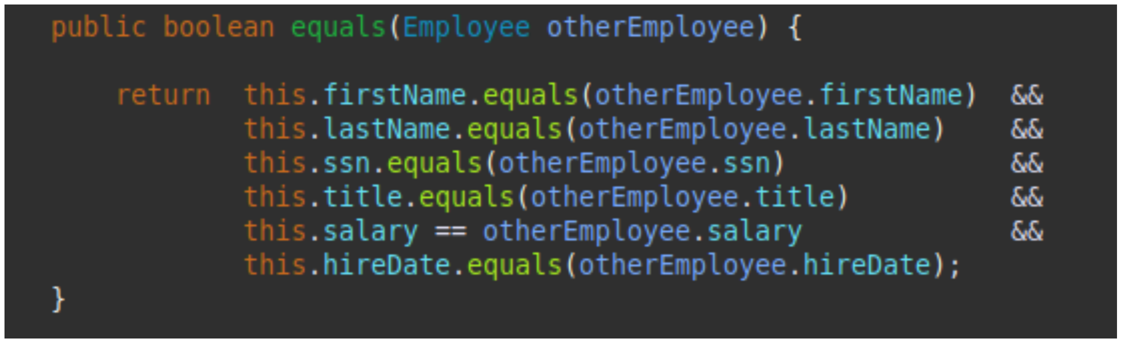
And finally, let’s instantiate some instances using the constructor that allows all field values to be passed it.



```java
public class Driver {

    public static void main(String[] args) {

        // create an instance using an overloaded constructor
        // These overloaded constructors have dependencies
        Date hired = new Date(1, 4, 2020);
        Employee emp1 = new Employee("Frank", "Stein", "123-45-6789", "junior developer",
                                     45000, hired, null);

        Employee emp2 = new Employee("Mary", "Shelly", "125-15-6569", "senior developer",
                75000, hired, null);

        System.out.println(emp1);
        System.out.println(emp2);

```

```
<terminated> Driver [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Jan 4, 2020, 2:04:15 PM)
Employee [firstName=Frank, lastName=Stein, ssn=123-45-6789, title=junior developer, salary=45000.0, hireDate=1/4/2020, dateTerminated=null]
Employee [firstName=Mary, lastName=Shelly, ssn=125-15-6569, title=senior developer, salary=75000.0, hireDate=1/4/2020, dateTerminated=null]
Are they equal: false
```

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 NullPointerExceptions thrown in the equals method. 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.



```java
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. 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 **hiredDate** 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?

The compareTo method
Another common method in the Java lexicon is the compareTo method. This method sits at the heart of a design pattern called the **Comparable Pattern.**

Design Pattern: Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time. Common design patterns are: Composition, Comparable and Inheritance.

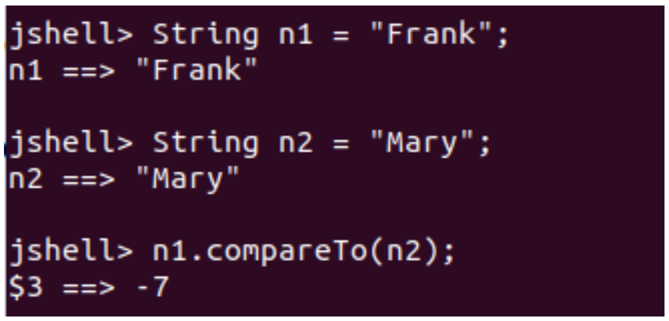
Design patterns in Java require the use of special structures called ***interfaces.*** We will be exploring this structure in detail when we cover abstract classes, but for now we will just be focusing on the compareTo method. The compareTo method is used to define the natural ordering of objects. Where the **equals** method allows us to determine direct equality, the compareTo method allows us to specify how to determine if one object **comes before or after** another object. We use this pattern to achieve sorting behavior among complex objects. Remember that our comparison operators are limited in the types that they can work with so we cannot compare two dates like this: **if(date1 < date2)** We have to define this behavior for ourselves. This method, much like the equals method requires advanced techniques to implement according to specification, but we can get our feet wet with this pattern by defining a simplistic implementation. Here is one of the specifications from the Java API

> compareTo
>
> This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. compareTo compares **this object** with **the specified object** for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

That last sentence tells us how we should define our behavior: **Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.**

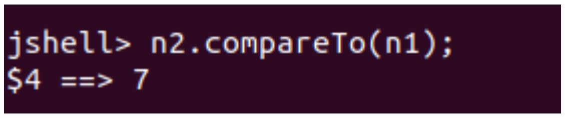
Let's examine the behavior of the String compareTo method. Notice below that compareTo returns a negative integer if this object **comes before** the specified object. The natural ordering of strings is alphabetic ... Frank comes before Mary alphabetically, so the method returns a negative integer. Don't worry too much about the value of the integer; it's relationship to zero is all we care about.

n1 => **"this object"**
n2 => "**specified object"**



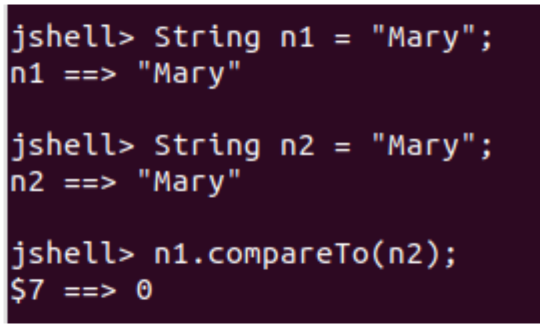
```
jshell> String n1 = "Frank";
n1 ==> "Frank"

jshell> String n2 = "Mary";
n2 ==> "Mary"

jshell> n1.compareTo(n2);
$3 ==> -7
```

What if we reverse the **this** and **specified** objects? Now compareTo returns a positive integer because this object ("Mary") comes after the specified object ("Frank");



```
jshell> n2.compareTo(n1);
$4 ==> 7
```

What if the objects are the same? compareTo returns 0.



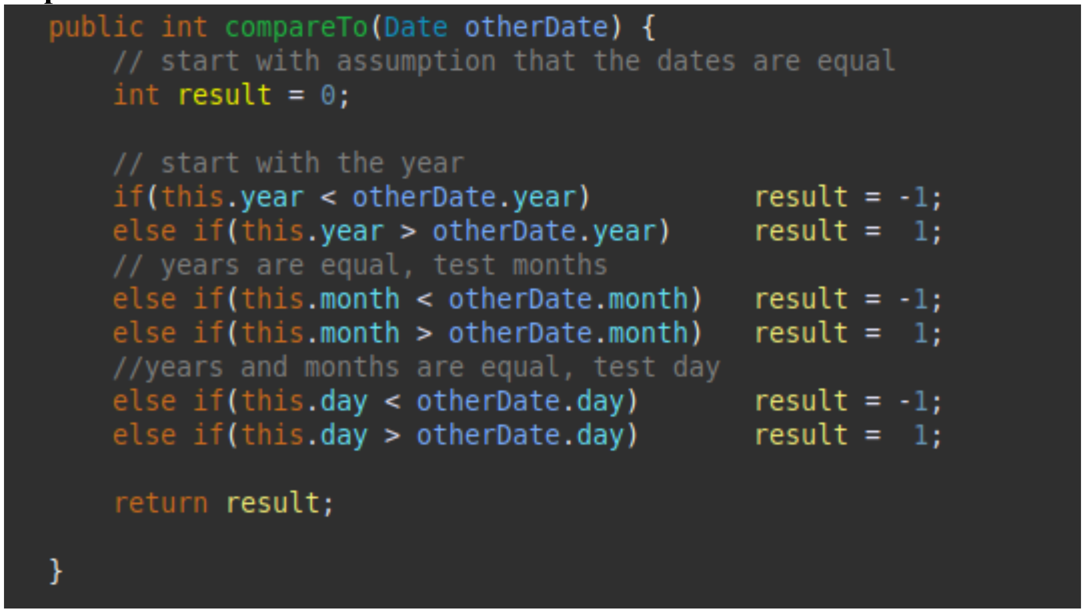
```
jshell> String n1 = "Mary";
n1 ==> "Mary"

jshell> String n2 = "Mary";
n2 ==> "Mary"

jshell> n1.compareTo(n2);
$7 ==> 0
```

Requirement: compareTo() == 0 must guarantee to match **equals() == true**

Signature: public int compareTo(Date otherDate)

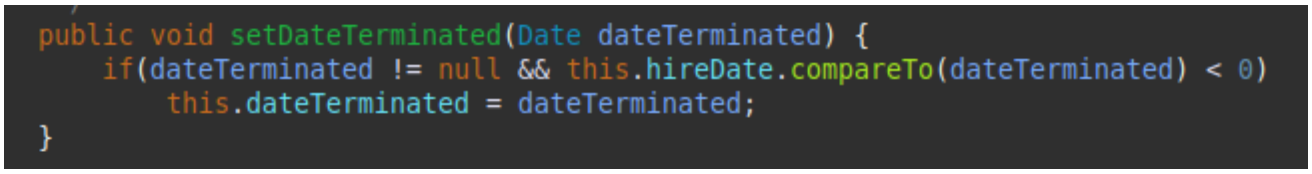
Understand that this is a Date behavior, not an Employee behavior.

Implementation



```
public int compareTo(Date otherDate) {
    // start with assumption that the dates are equal
    int result = 0;

    // start with the year
    if(this.year < otherDate.year)          result = -1;
    else if(this.year > otherDate.year)     result =  1;
    // years are equal, test months
    else if(this.month < otherDate.month)   result = -1;
    else if(this.month > otherDate.month)   result =  1;
    //years and months are equal, test day
    else if(this.day < otherDate.day)       result = -1;
    else if(this.day > otherDate.day)       result =  1;

    return result;

}
```

Now we can add this method into our **setDateTerminated** implementation to ensure that termination dates are greater than hire dates. You should run termination dates through this method anytime one is changed. Notice that I also added a null check to this method. This is important. If you have a field that could be null by choice, you need to make sure you never call a method on object unless it is not null.



```
public void setDateTerminated(Date dateTerminated) {
    if(dateTerminated != null && this.hireDate.compareTo(dateTerminated) < 0)
        this.dateTerminated = dateTerminated;
}
```

Unit Testing

Let's write a JUnit test to ensure that this code functions correctly. I'm going to individually test the compareTo method of the Date class and the setDateTerminated implementation in the Employee class.

But, before we go too much further into JUnit, let's talk a little bit about unit testing and regression testing and why they matter in general.

Unit testing is a form of white-box testing, in which test cases are based on an internal structure. **The tester chooses inputs to explore particular paths and determines the appropriate output.** The purpose of unit testing is to examine the individual components or pieces of methods/classes to verify functionality, ensuring the behavior is as expected.

The exact scope of a "unit" is often left to interpretation, but a nice rule of thumb is for a unit to contain the least amount of code that performs a standalone task (e.g. a single method or class). There is good reason that we limit scope when unit testing — if we construct a test that incorporates multiple aspects of a project, we have shifted focus from functionality of a single method to interaction between different portions of the code. If the test fails, and we don't know why it failed, we are left wondering whether the point of failure was within the method we were interested in or in the dependencies associated with that method.

Regression Testing

Complementing unit testing, regression testing makes certain that the latest fix, enhancement, or patch did not break existing functionality, by testing the changes you've made to your code. Changes to code are inevitable, whether they are modifications of existing code, or adding packages for new functionality; your code will certainly change. It is in this change that lies the most danger, so, with that in mind, regression testing is a must.

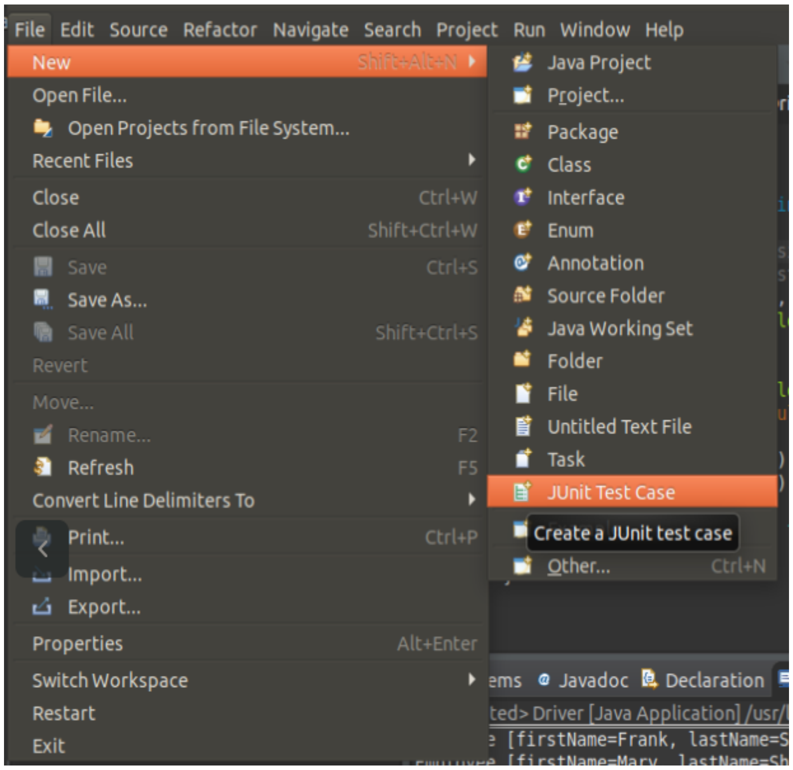
What Is JUnit?

JUnit is a unit testing framework for the Java programming language that plays a big role in regression testing. As an open-source framework, it is used to write and run repeatable automated tests.

As with anything else, the JUnit framework has evolved over time. The major change to make note of is the introduction of annotations that came along with the release of JUnit 4, which provided an increase in organization and readability of JUnits.

Most modern IDEs come with the ability to generate and run JUnit tests. Eclipse is no different. The following screen shots will walk you through the setup of a JUnit test for our compareTo method in the Date class in Eclipse.

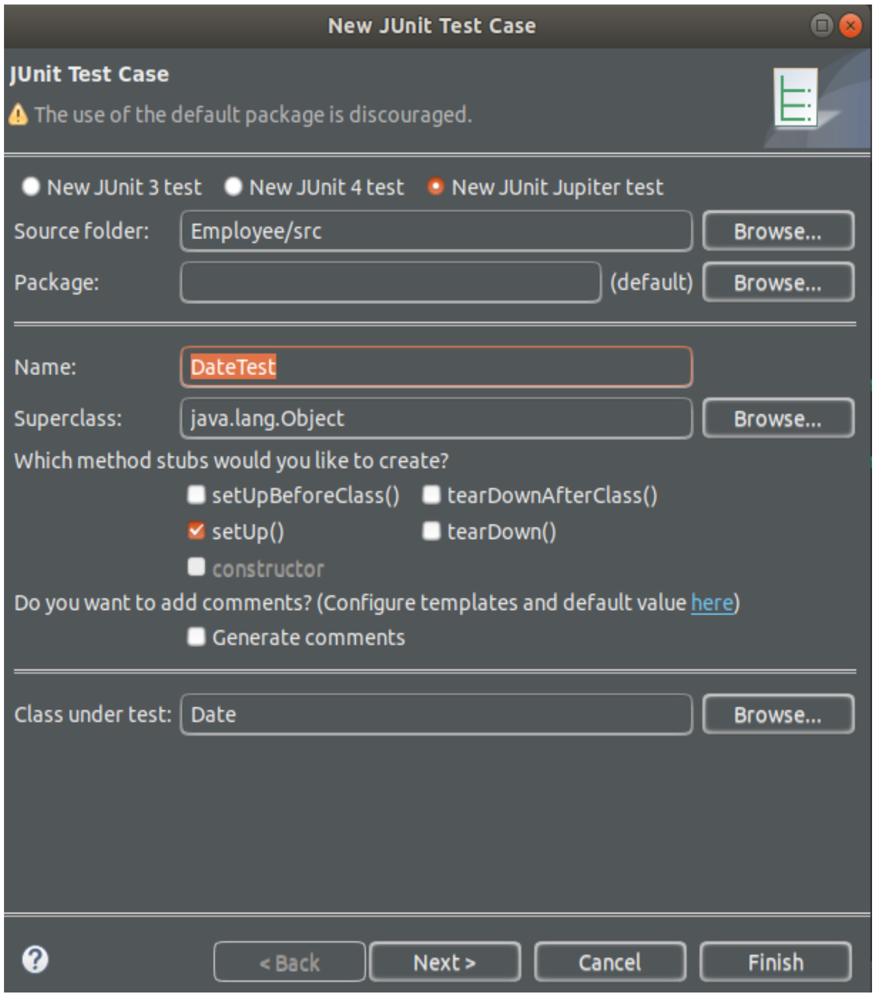
Select the Date class
Select New → JUnit Test Case



Accept the default class name of **DateTest**

Select the box to include a stub of the **setUp** method

Select Next



New JUnit Test Case

JUnit Test Case

The use of the default package is discouraged.

New JUnit 3 test   New JUnit 4 test   [✓ New JUnit Jupiter test]

Source folder: Employee/src

Package: (default)

Name: DateTest

Superclass: java.lang.Object

Which method stubs would you like to create?

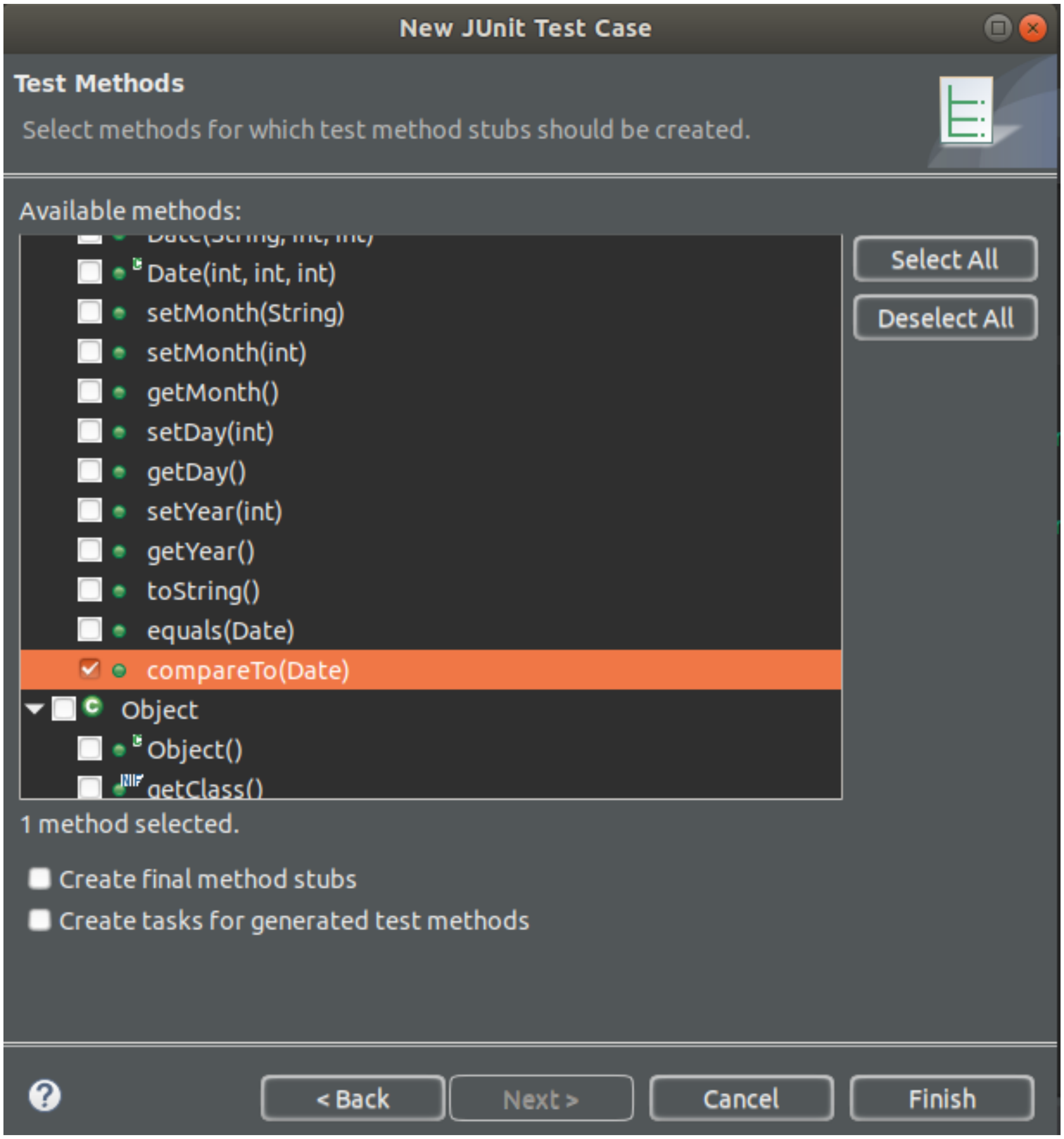
- setUpBeforeClass()
- tearDownAfterClass()
- [✓ setUp()]
- tearDown()
- constructor

Do you want to add comments? (Configure templates and default value here)

- Generate comments

Class under test: Date

Scroll down until you see **compareTo**, select the box next to this method name
Select Finish



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- [ ] Date(String, int, int)
- [ ] Date(int, int, int)
- [ ] setMonth(String)
- [ ] setMonth(int)
- [ ] getMonth()
- [ ] setDay(int)
- [ ] getDay()
- [ ] setYear(int)
- [ ] getYear()
- [ ] toString()
- [ ] equals(Date)
- [✓] compareTo(Date)
- [ ] Object
  - [ ] Object()
  - [ ] getClass()

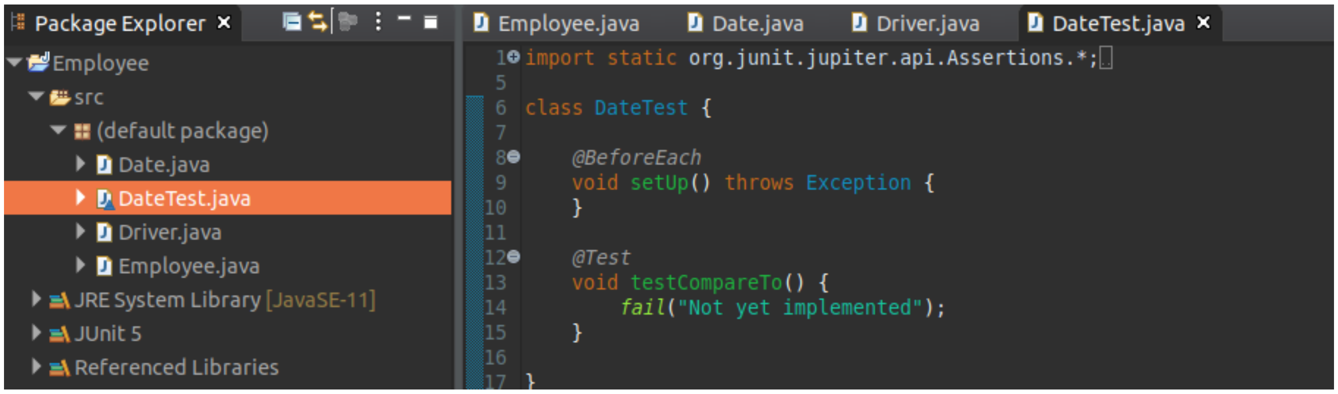
1 method selected.

- [ ] Create final method stubs
- [ ] Create tasks for generated test methods

You should now see the generated JUnit test file containing two method stubs

1. setUp
2. testCompareTo

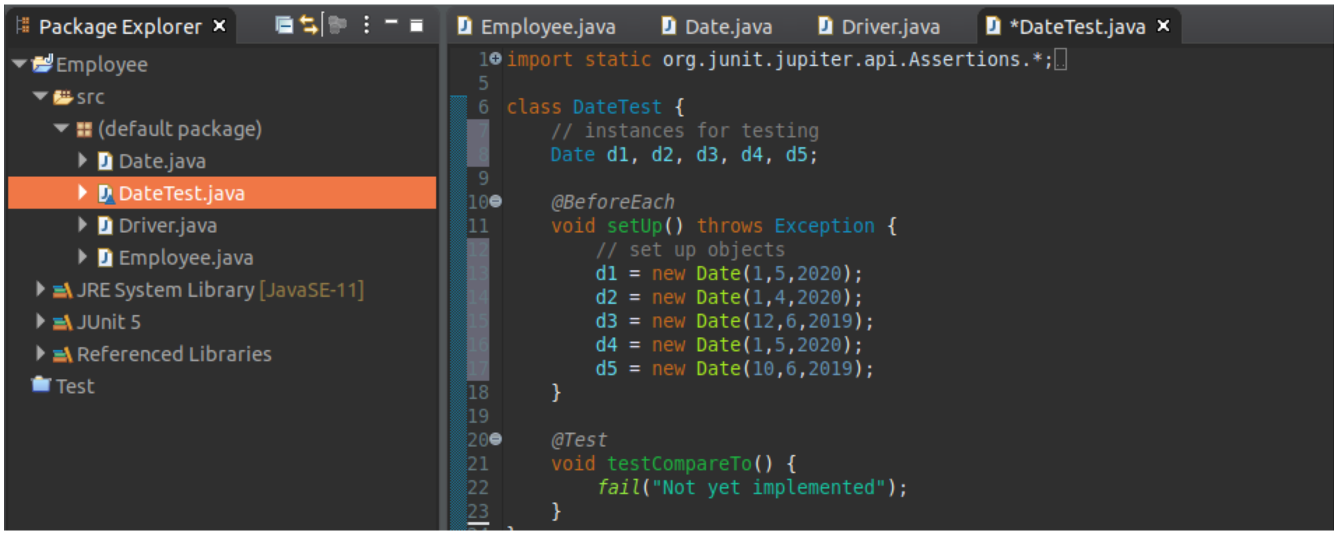
Notice that the testCompareTo stub is initialized with an explicit fail statement. Tests should report failures until they have actually been implemented.



```
import static org.junit.jupiter.api.Assertions.*;

class DateTest {

    @BeforeEach
    void setUp() throws Exception {
    }

    @Test
    void testCompareTo() {
        fail("Not yet implemented");
    }

}
```

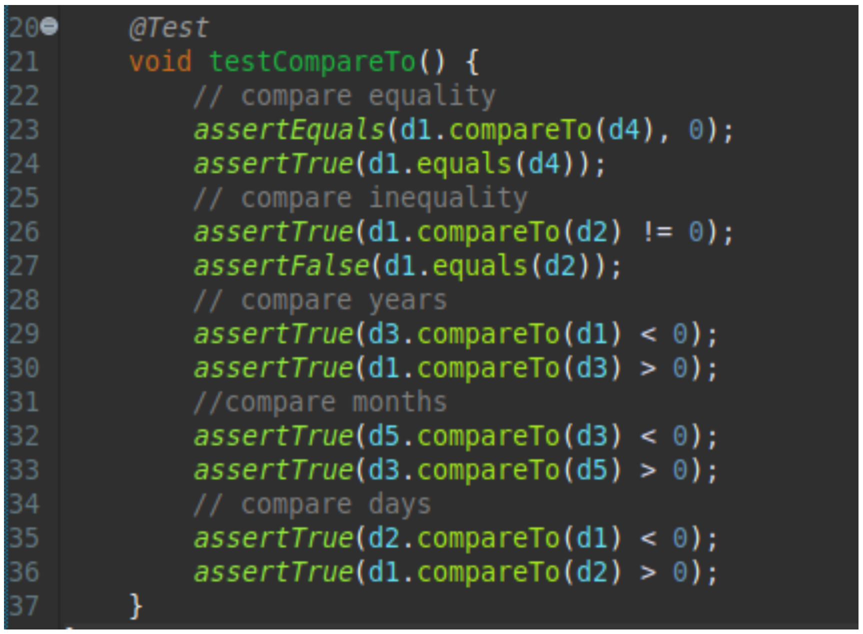
The setUp method allows us to specify any precondition code. We will use this area to instantiate our test objects. Be sure to include ample instances to test all of the conditions of this compareTo method.

1. Order dates based on the year
2. Order dates based on the month
3. Order dates based on the day
4. Test for direct equality
5. Ensure **compareTo( ) == 0** and **equals( ) == true**



```
import static org.junit.jupiter.api.Assertions.*;

class DateTest {
    // instances for testing
    Date d1, d2, d3, d4, d5;

    @BeforeEach
    void setUp() throws Exception {
        // set up objects
        d1 = new Date(1,5,2020);
        d2 = new Date(1,4,2020);
        d3 = new Date(12,6,2019);
        d4 = new Date(1,5,2020);
        d5 = new Date(10,6,2019);
    }

    @Test
    void testCompareTo() {
        fail("Not yet implemented");
    }
```

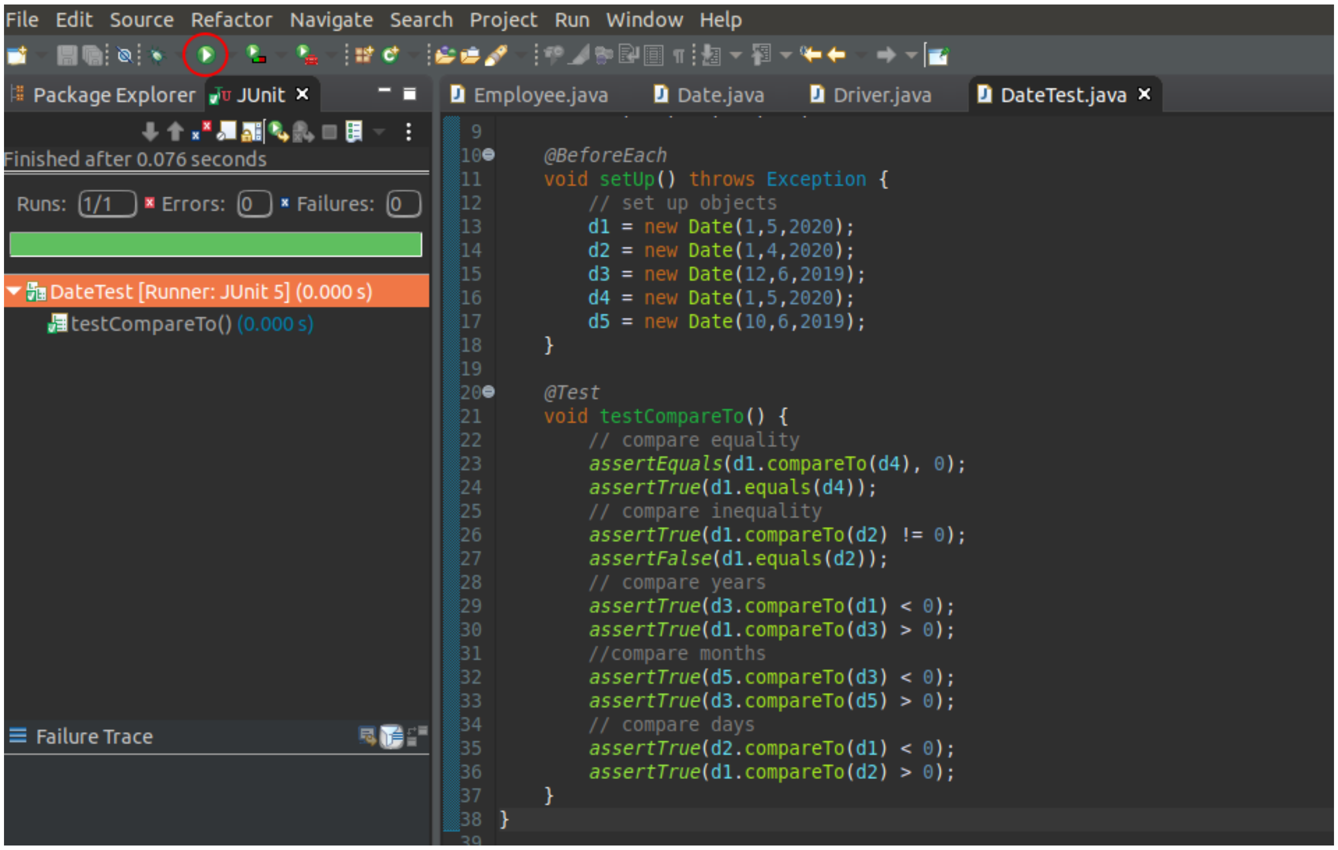
Use JUnit methods **assertEquals, assertFalse and assertTrue** to run the tests. Be careful and pay very close attention when you are designing these tests. Make sure you fully understand what the methods do, what they accept and what they return. If you are not sure, then you need to work this out on paper beforehand.



```
    @Test
    void testCompareTo() {
        // compare equality
        assertEquals(d1.compareTo(d4), 0);
        assertTrue(d1.equals(d4));
        // compare inequality
        assertTrue(d1.compareTo(d2) != 0);
        assertFalse(d1.equals(d2));
        // compare years
        assertTrue(d3.compareTo(d1) < 0);
        assertTrue(d1.compareTo(d3) > 0);
        //compare months
        assertTrue(d5.compareTo(d3) < 0);
        assertTrue(d3.compareTo(d5) > 0);
        // compare days
        assertTrue(d2.compareTo(d1) < 0);
        assertTrue(d1.compareTo(d2) > 0);
    }
```

Run the JUnit Tests

Press the Run arrow to execute the tests. The Eclipse perspective will change to reflect the JUnit context. We want the green bar with 0 errors and 0 Failures. This means that the tests all pass.



Finished after 0.076 seconds

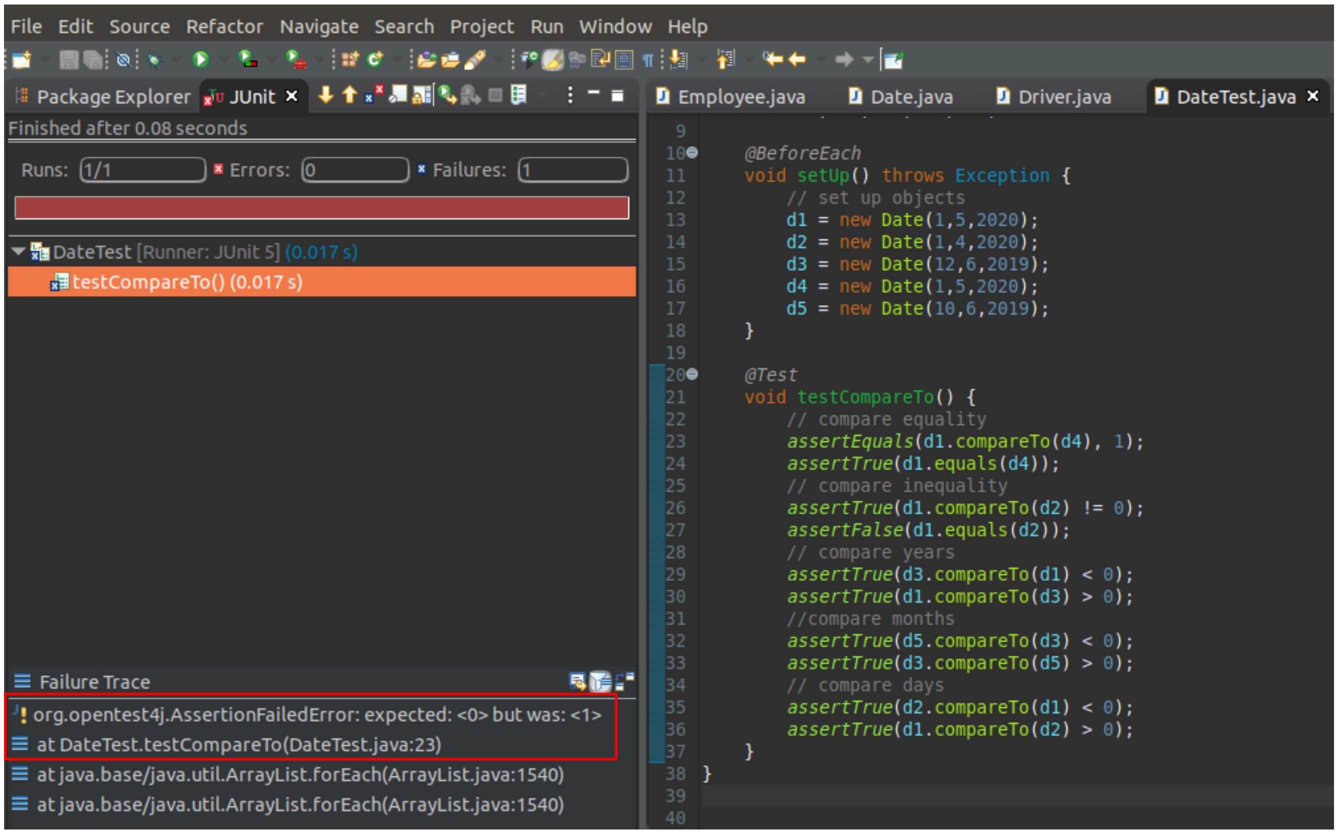
Runs: 1/1 Errors: 0 Failures: 0

DateTest [Runner: JUnit 5] (0.000 s)
testCompareTo() (0.000 s)

```

    @BeforeEach
    void setUp() throws Exception {
        // set up objects
        d1 = new Date(1,5,2020);
        d2 = new Date(1,4,2020);
        d3 = new Date(12,6,2019);
        d4 = new Date(1,5,2020);
        d5 = new Date(10,6,2019);
    }

    @Test
    void testCompareTo() {
        // compare equality
        assertEquals(d1.compareTo(d4), 0);
        assertTrue(d1.equals(d4));
        // compare inequality
        assertTrue(d1.compareTo(d2) != 0);
        assertFalse(d1.equals(d2));
        // compare years
        assertTrue(d3.compareTo(d1) < 0);
        assertTrue(d1.compareTo(d3) > 0);
        //compare months
        assertTrue(d5.compareTo(d3) < 0);
        assertTrue(d3.compareTo(d5) > 0);
        // compare days
        assertTrue(d2.compareTo(d1) < 0);
        assertTrue(d1.compareTo(d2) > 0);
    }
}
```

Here is what a failure would look like. Notice the section marked **Failure Trace.** This area will tell you

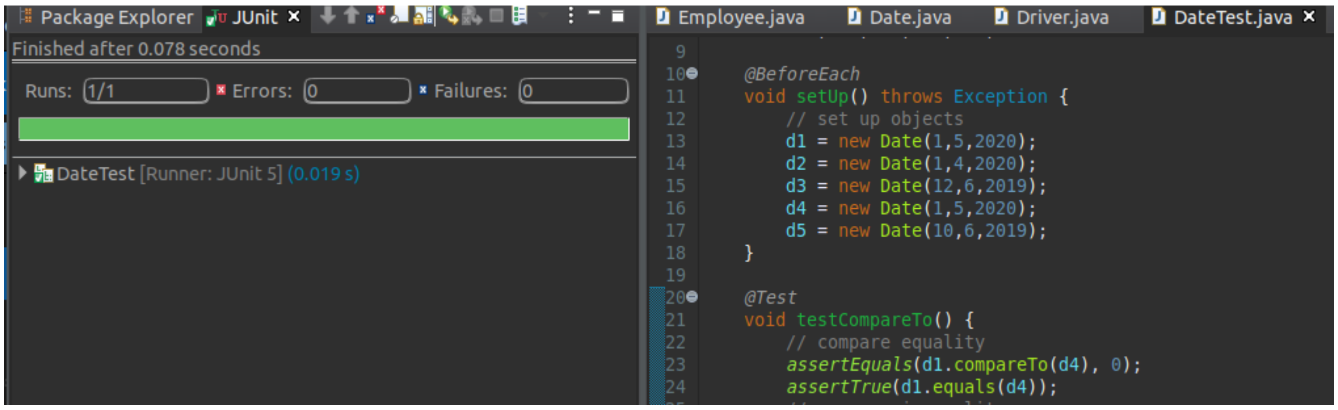
- What result was expected
- What result was received
- The line number of the test that failed



```
Finished after 0.08 seconds
Runs: 1/1    Errors: 0    Failures: 1

DateTest [Runner: JUnit 5] (0.017 s)
    testCompareTo() (0.017 s)

Failure Trace
org.opentest4j.AssertionFailedError: expected: <0> but was: <1>
at DateTest.testCompareTo(DateTest.java:23)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1540)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1540)
```

```java
 9
10      @BeforeEach
11      void setUp() throws Exception {
12          // set up objects
13          d1 = new Date(1,5,2020);
14          d2 = new Date(1,4,2020);
15          d3 = new Date(12,6,2019);
16          d4 = new Date(1,5,2020);
17          d5 = new Date(10,6,2019);
18      }
19
20      @Test
21      void testCompareTo() {
22          // compare equality
23          assertEquals(d1.compareTo(d4), 1);
24          assertTrue(d1.equals(d4));
25          // compare inequality
26          assertTrue(d1.compareTo(d2) != 0);
27          assertFalse(d1.equals(d2));
28          // compare years
29          assertTrue(d3.compareTo(d1) < 0);
30          assertTrue(d1.compareTo(d3) > 0);
31          //compare months
32          assertTrue(d5.compareTo(d3) < 0);
33          assertTrue(d3.compareTo(d5) > 0);
34          // compare days
35          assertTrue(d2.compareTo(d1) < 0);
36          assertTrue(d1.compareTo(d2) > 0);
37      }
38  }
39
40
```

If you inspect the test on line 23 closely, you can see that we wanted to test the compareTo for direct equality of two Date objects. CompareTo should return a 0 when the objects are equal, not a 1. This error was the result of a **faulty test.** If I change the 1 to a zero the test passes.



```
Finished after 0.078 seconds
Runs: 1/1    Errors: 0    Failures: 0

DateTest [Runner: JUnit 5] (0.019 s)
```

```java
 9
10      @BeforeEach
11      void setUp() throws Exception {
12          // set up objects
13          d1 = new Date(1,5,2020);
14          d2 = new Date(1,4,2020);
15          d3 = new Date(12,6,2019);
16          d4 = new Date(1,5,2020);
17          d5 = new Date(10,6,2019);
18      }
19
20      @Test
21      void testCompareTo() {
22          // compare equality
23          assertEquals(d1.compareTo(d4), 0);
24          assertTrue(d1.equals(d4));
```

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.

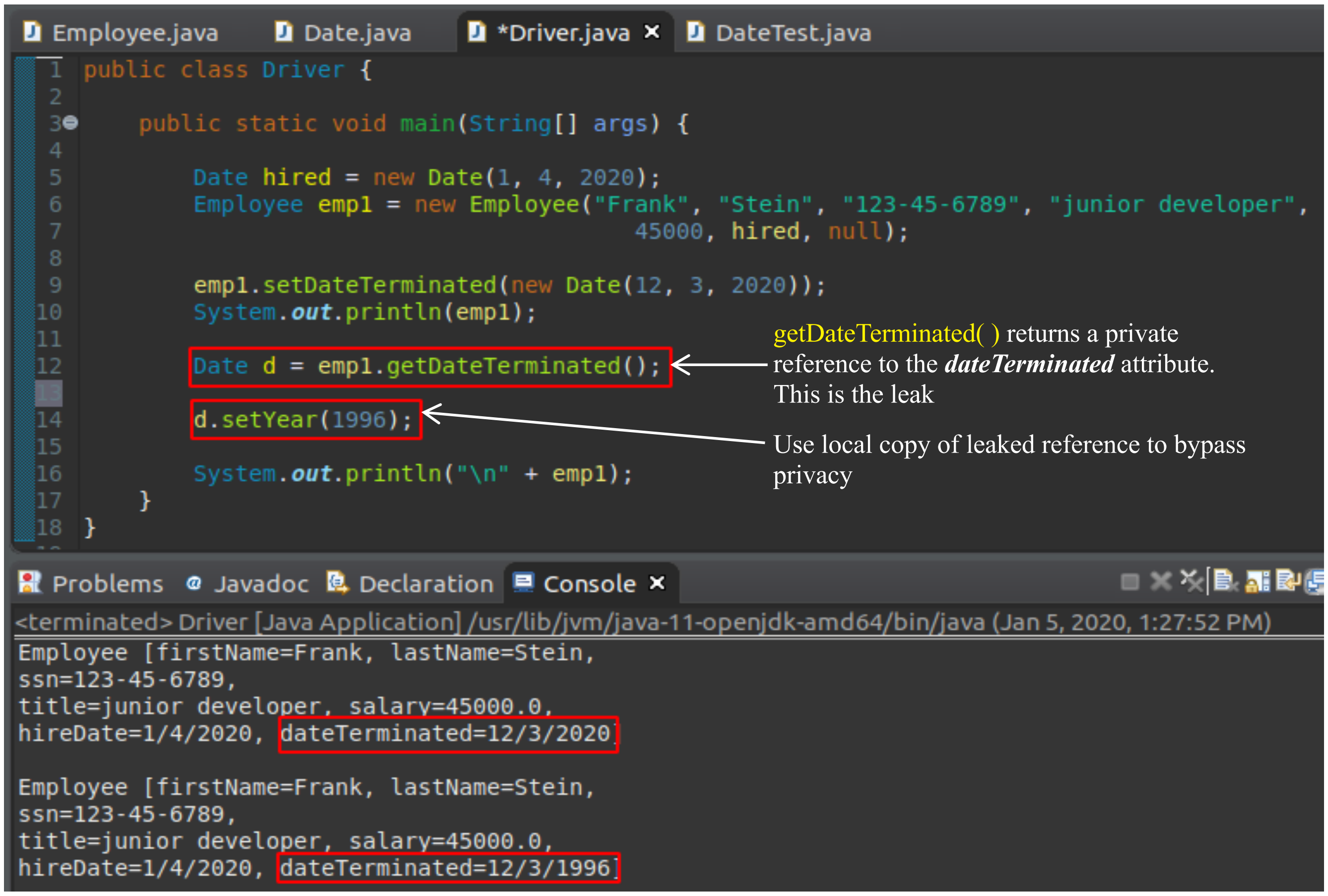
Employee Invariant: The terminated date must come after the hired date

We addressed this invariant in the following ways.

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

Surely that is sufficient? No. Our Employee class has privacy leaks in a variety of areas. This is because Eclipse automatically generates getters/setters with privacy leaks. **Bad Eclipse!**

Let's break it!



```
 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();
13
14         d.setYear(1996);
15
16         System.out.println("\n" + emp1);
17     }
18 }
```

getDateTerminated( ) returns a private reference to the ***dateTerminated*** attribute. This is the leak

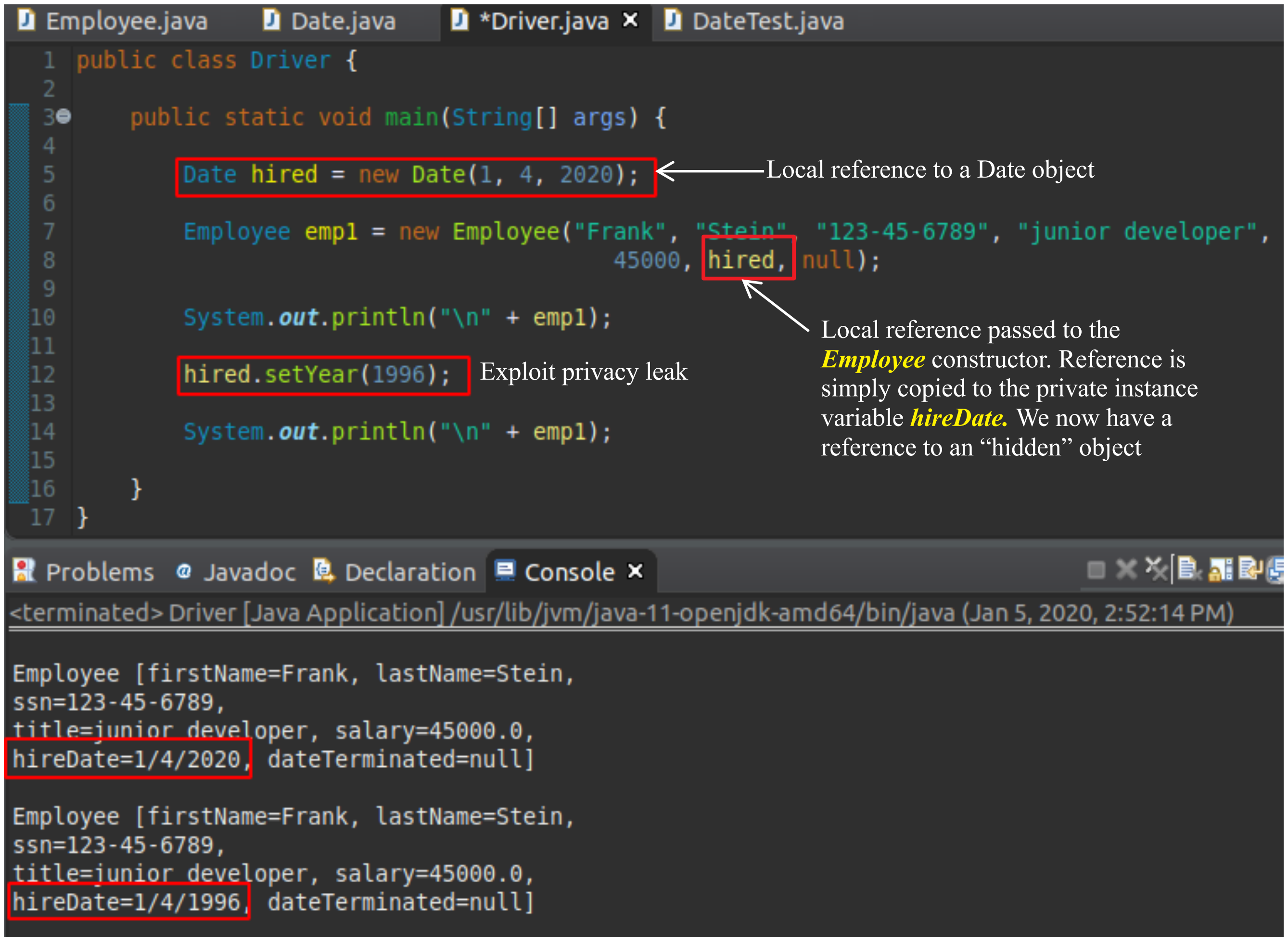
Use local copy of leaked reference to bypass privacy

```
<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 **(PRIVACY LEAK!!)**
6. Exploited the privacy leak by manipulating the returned Date object

Privacy Leak Two



```java
 1 public class Driver {
 2 
 3     public static void main(String[] args) {
 4 
 5         Date hired = new Date(1, 4, 2020);
 6 
 7         Employee emp1 = new Employee("Frank", "Stein", "123-45-6789", "junior developer",
 8                                     45000, hired, null);
 9 
10         System.out.println("\n" + emp1);
11 
12         hired.setYear(1996);
13 
14         System.out.println("\n" + emp1);
15 
16     }
17 }
```

Local reference to a Date object

Local reference passed to the ***Employee*** constructor. Reference is simply copied to the private instance variable ***hireDate.*** We now have a reference to an “hidden” object

Exploit privacy leak

```
<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

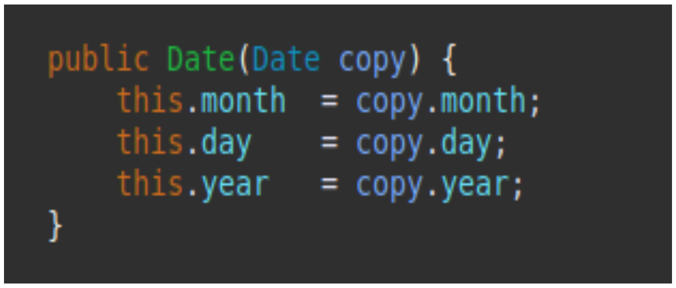
We cannot rely on class consumers to carefully guard privacy. For the classes we write, it is our responsibility. If you need to write methods that either accept or return objects, and privacy is a concern, then the best approach is to return a copy of the object. Copying the object allocates new memory 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 to 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.

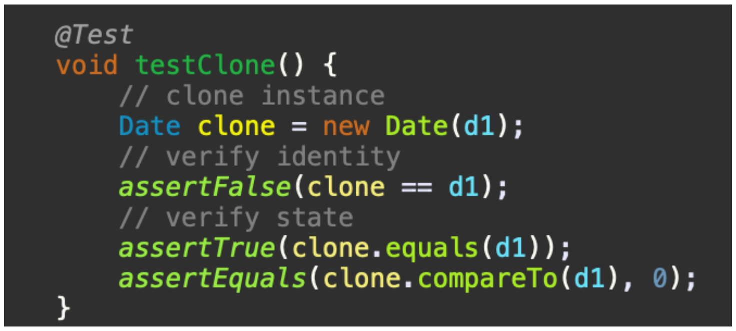


```
public Date(Date copy) {
    this.month  = copy.month;
    this.day    = copy.day;
    this.year   = copy.year;
}
```

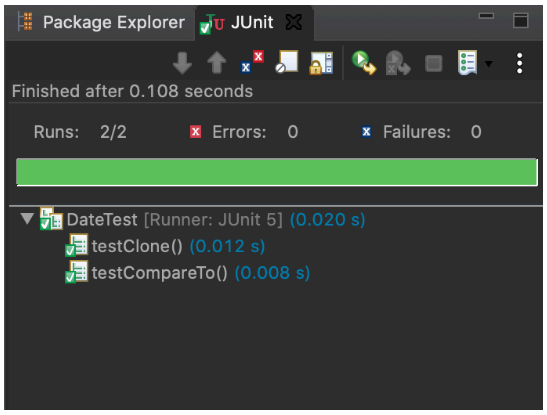
Clone Test: To verify a proper clone you should test for the following

- 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



```java
@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);
}
```



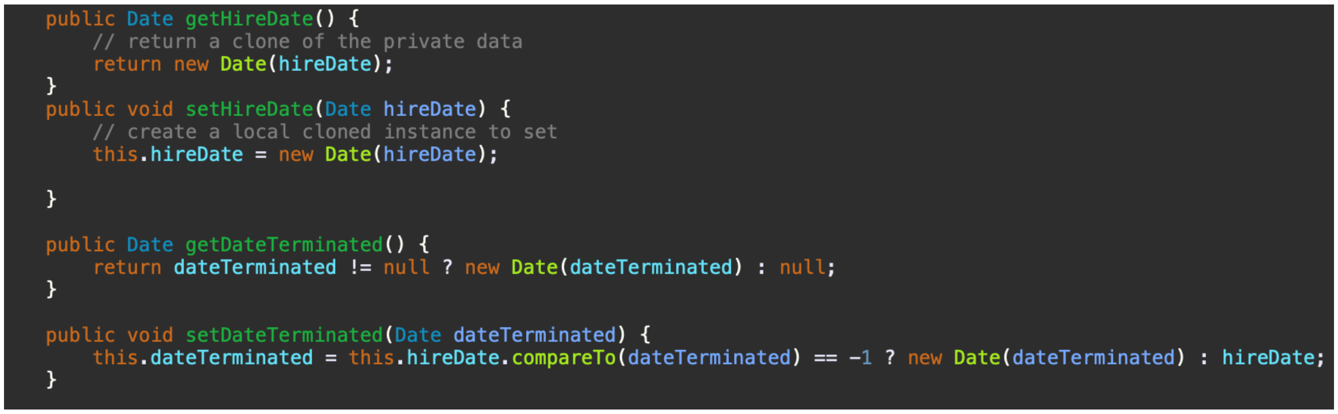
```
Finished after 0.108 seconds

Runs:  2/2        Errors:  0        Failures:  0

DateTest [Runner: JUnit 5] (0.020 s)
    testClone() (0.012 s)
    testCompareTo() (0.008 s)
```

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

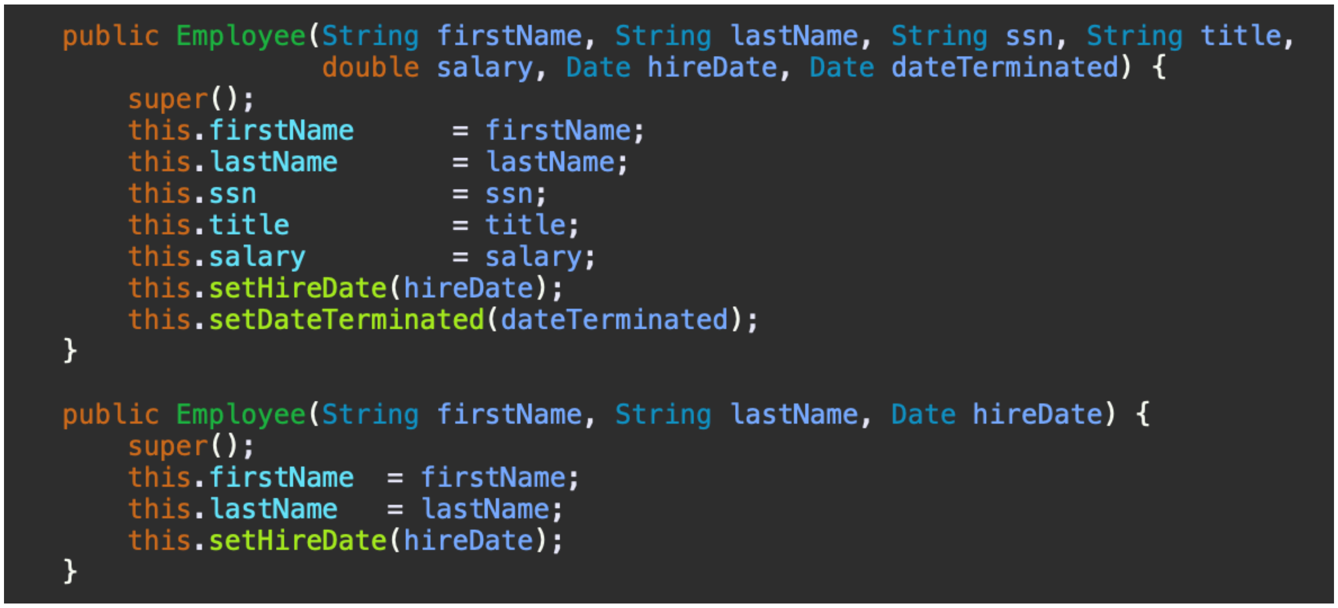
Let’s fix the privacy leaks in our Employee class by creating clones for each of the cases listed above. Cloning should be implemented wherever there is a private instance variable being returned from a public method, or an external reference being passed as an argument through a public method, that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

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



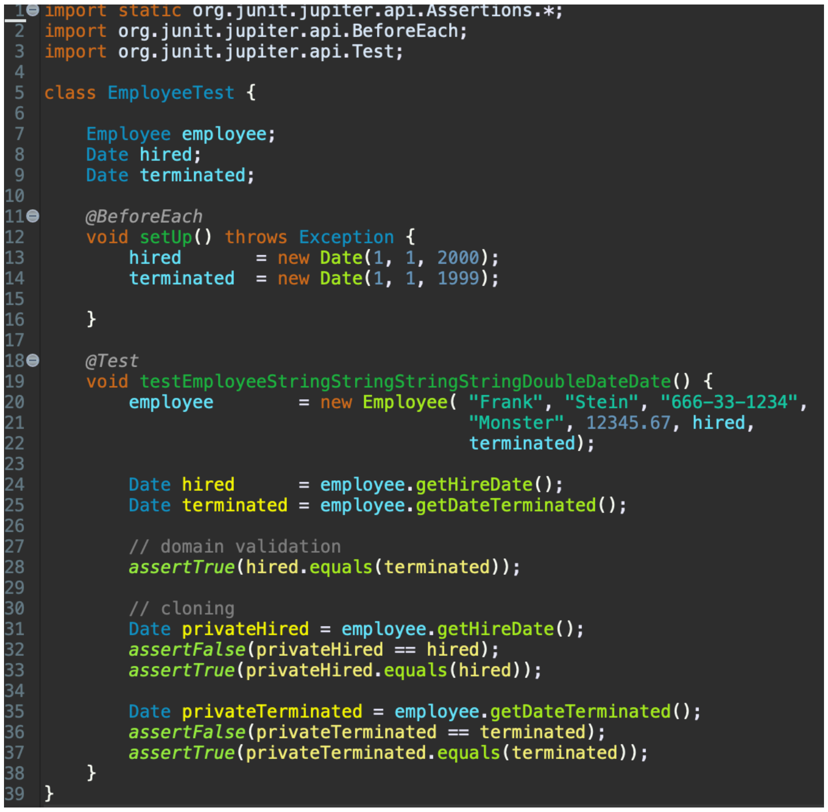
```java
public Date getHireDate() {
    // return a clone of the private data
    return new Date(hireDate);
}
public void setHireDate(Date hireDate) {
    // create a local cloned instance to set
    this.hireDate = new Date(hireDate);

}

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

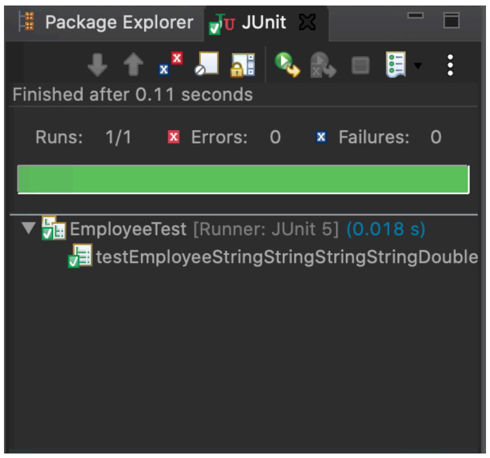


```java
public Employee(String firstName, String lastName, String ssn, String title,
                double salary, Date hireDate, Date dateTerminated) {
    super();
    this.firstName      = firstName;
    this.lastName       = lastName;
    this.ssn            = ssn;
    this.title          = title;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



```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class EmployeeTest {

    Employee employee;
    Date hired;
    Date terminated;

    @BeforeEach
    void setUp() throws Exception {
        hired       = new Date(1, 1, 2000);
        terminated  = new Date(1, 1, 1999);

    }

    @Test
    void testEmployeeStringStringStringStringDoubleDateDate() {
        employee        = new Employee( "Frank", "Stein", "666-33-1234",
                                        "Monster", 12345.67, hired,
                                        terminated);

        Date hired      = employee.getHireDate();
        Date terminated = employee.getDateTerminated();

        // domain validation
        assertTrue(hired.equals(terminated));

        // cloning
        Date privateHired = employee.getHireDate();
        assertFalse(privateHired == hired);
        assertTrue(privateHired.equals(hired));

        Date privateTerminated = employee.getDateTerminated();
        assertFalse(privateTerminated == terminated);
        assertTrue(privateTerminated.equals(terminated));
    }
}
```



```
Finished after 0.11 seconds

Runs:  1/1    Errors:  0    Failures:  0

EmployeeTest [Runner: JUnit 5] (0.018 s)
    testEmployeeStringStringStringStringDouble
```