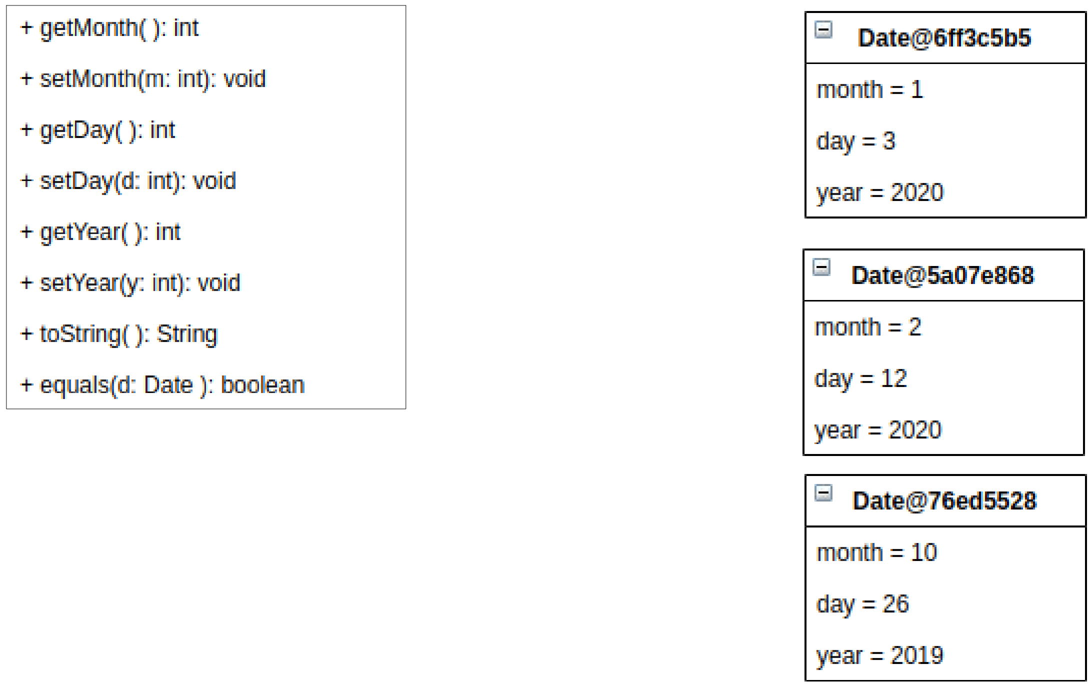
Method Overloading

Method overloading is a feature of most object-oriented programming languages in which two or more methods share the **same name but have different parameters**. Specifically, the number, data type, and/or order of the parameters are different. When the code is compiled, the correct method will be automatically selected based on how it is called. Methods are also known as functions in some programming languages, so method overloading is sometimes referred to as **function overloading**

The primary requirement for method overloading is that the methods share the same name. Their method signatures — the method name, the number of parameters, and the parameter data types — should otherwise be unique. In this way, the compiler can determine which method to bind to the call.

Method Binding: A class’s instance method definitions are stored in a separate area of memory from the instance variables. This should make sense as it is the variables that define the object’s state and allow one instance of a class to be distinct from a separate instance of the class and it would be a waste of space to store redundant copies of method definitions.

Method Pool **Instances**



When a method is invoked upon an object,

int day = d.getDay();

the JVM **binds** the method definition (**getDay()**)to the invoking instance (**d**). Programmers communicate the binding with the ***dot operator*** which is then intrepeted by the compiler and the JVM. The is one of the fundamental concepts behind **polymorphism.**

Constructors, the methods used to instantiate objects, are often overloaded. This is done to initialize an object with non-default values. For example, an employee object with two fields (name and date of birth, or dob) might have the following overloaded constructors:

- Employee()
- Employee(name)
- Employee(name, dob)

The first constructor creates an employee object with blank name and dob fields. The second sets the name field, but leaves the dob field blank, and the third defines both the name and dob fields. The compiler is able to determine which version of the method to bind based on the signature.

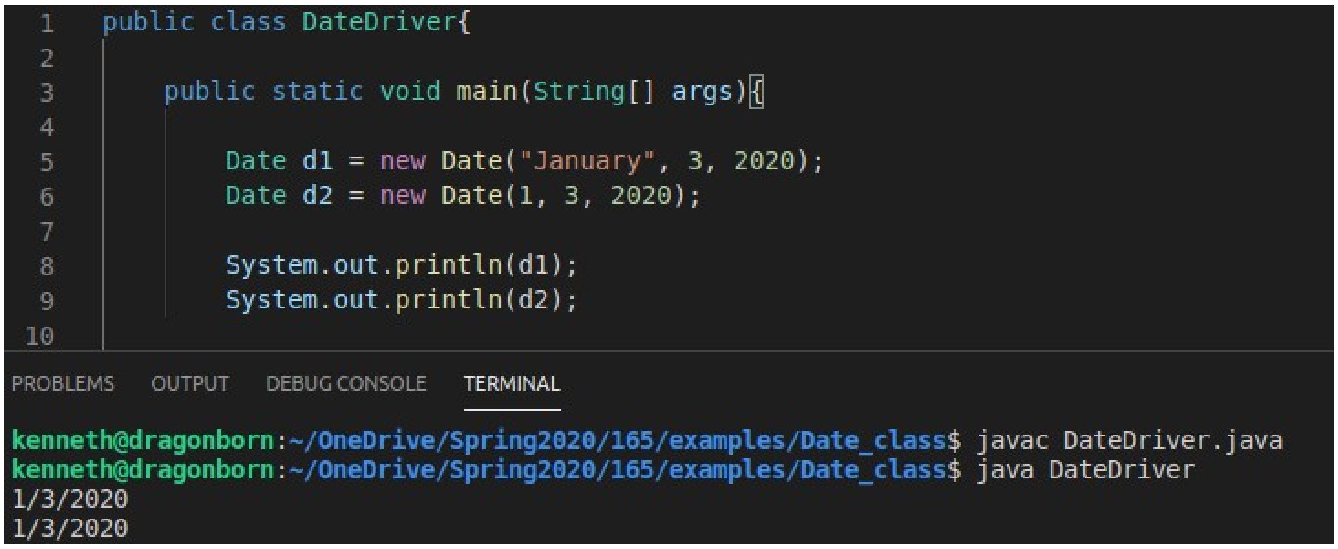
Methods are also overloaded to preserve backward compatibility. A method that does a complex calculation could be given a new requirement to optionally perform the same calculation with a slight change. A new parameter is added to the method that will determine how to perform the calculation — the old way or the new way.

To avoid having to find all cases in which the method is called and add the new parameter, the method can be overloaded. The new method will have the old signature and be called by existing code. It will not contain any logic itself, and will simply call the modified method and pass in a default of “old way” for the new parameter. New code will call the modified method and pass the new parameter with the appropriate value, old way or new way.

Method overloading is a type of polymorphism, in which the same logical method can be, in practice, used in multiple ways. Method overloading is not the same as method overriding. Overriding is where the definition of a method in a parent class is changed by a child class. In this case, both methods will have the same signature. More on this in the sections on inheritance.

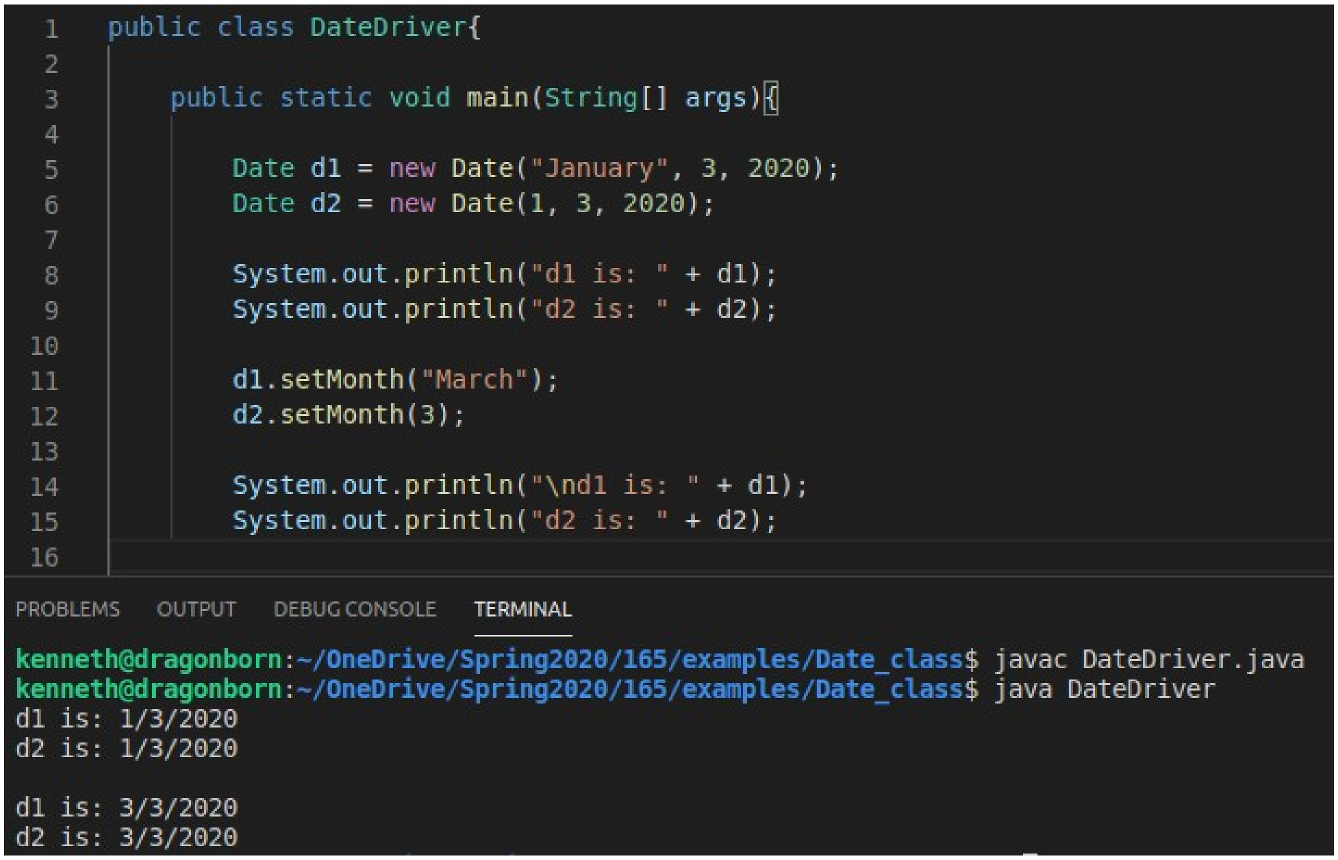
Let’s add an overloaded method to our Date class. Suppose we want to provide the ability to treat the month as a string {“January”, “February” . . . etc}. We will still store it as integer in the instance variables but will allow the setting of the data as a String and an int. To ensure consistency we will provide an overloaded setter and constructor.

These new behaviors will provide the following functionality.


```java
public class DateDriver{

    public static void main(String[] args){

        Date d1 = new Date("January", 3, 2020);
        Date d2 = new Date(1, 3, 2020);

        System.out.println(d1);
        System.out.println(d2);

```

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ java DateDriver
1/3/2020
1/3/2020
```

Notice the calls to the different constructors. They both result in setting the month to 1. The internals of how this actually happens is of no concern to the consumer . . . obviously it is to us though and we will be inspecting approaches to handling this.


```java
public class DateDriver{

    public static void main(String[] args){

        Date d1 = new Date("January", 3, 2020);
        Date d2 = new Date(1, 3, 2020);

        System.out.println("d1 is: " + d1);
        System.out.println("d2 is: " + d2);

        d1.setMonth("March");
        d2.setMonth(3);

        System.out.println("\nd1 is: " + d1);
        System.out.println("d2 is: " + d2);

```

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ java DateDriver
d1 is: 1/3/2020
d2 is: 1/3/2020

d1 is: 3/3/2020
d2 is: 3/3/2020
```

Notice the calls to the overloaded **setMonth** methods on lines 11 and 12. The compiler is able to determine which versions of these methods to bind based on the data type of the argument.

Method Overloading Rules: For a method to be properly overloaded follow these rules

1. The methods must be in the same class or a sub class (more on that later)
2. The methods have the same identifier.
3. The formal parameter list must be different. This could take many forms . . . different data types, different number of parameters or different order. Simply changing the identifiers of the parameters has **no effect.**
4. Changing the return type has no effect on method overloading

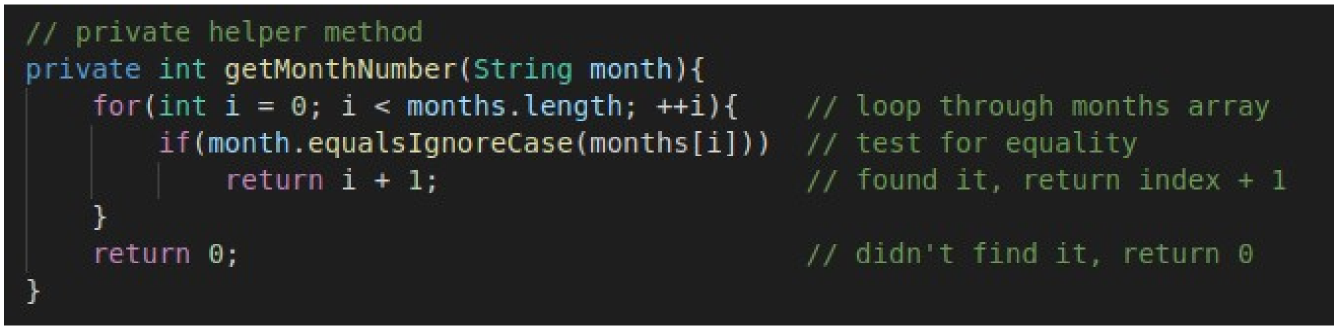
Let's take a look at the Date class modifications that were necessary for this method overloading. There are some issues that need to flushed out.

- How do we validate the month as a string?

To accomplish this I defined an array of type String to hold all of the month names. The associated index values for the months will be the set of positive integers {0 – 11}. When the month is passed in as a String we can then iterate through the array calling the equals method on each element. We will either find a match or not.

- If a match is found return the index of the match + 1 (why +1?)
- If no match is found we can return 0 to signify **no match**

To begin I defined a private helper method called getMonthNumber. Helper methods are a great way to subdivide complex processes so that our code does not become overly complex.

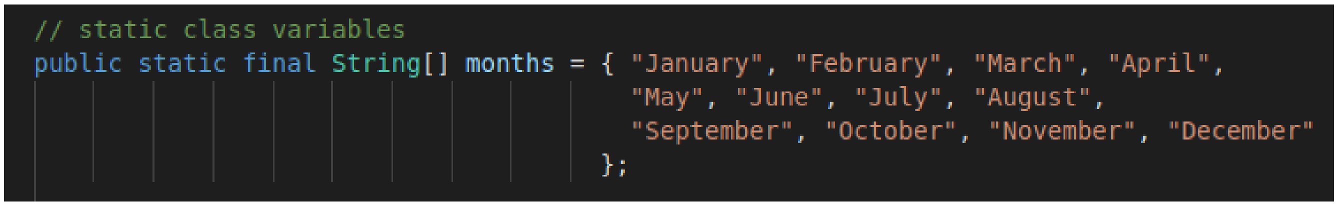


```java
// private helper method
private int getMonthNumber(String month){
    for(int i = 0; i < months.length; ++i){    // loop through months array
        if(month.equalsIgnoreCase(months[i]))  // test for equality
            return i + 1;                      // found it, return index + 1
    }
    return 0;                                  // didn't find it, return 0
}
```

Why is the method's access declared **private?** This was a design decision. If the method is only a helper function that other methods depend on then there is no need to expose the method to the general public. You may decide that this method is useful enough to be declared public, or that it is necessary in external operations. If this situation meets your needs then make the method public. You have complete control.

Notice the usage of the String method **equalsIgnoreCase**. This allows for case insensitive equality checks.

Also notice the reference to the array named **months**. Here is its definition



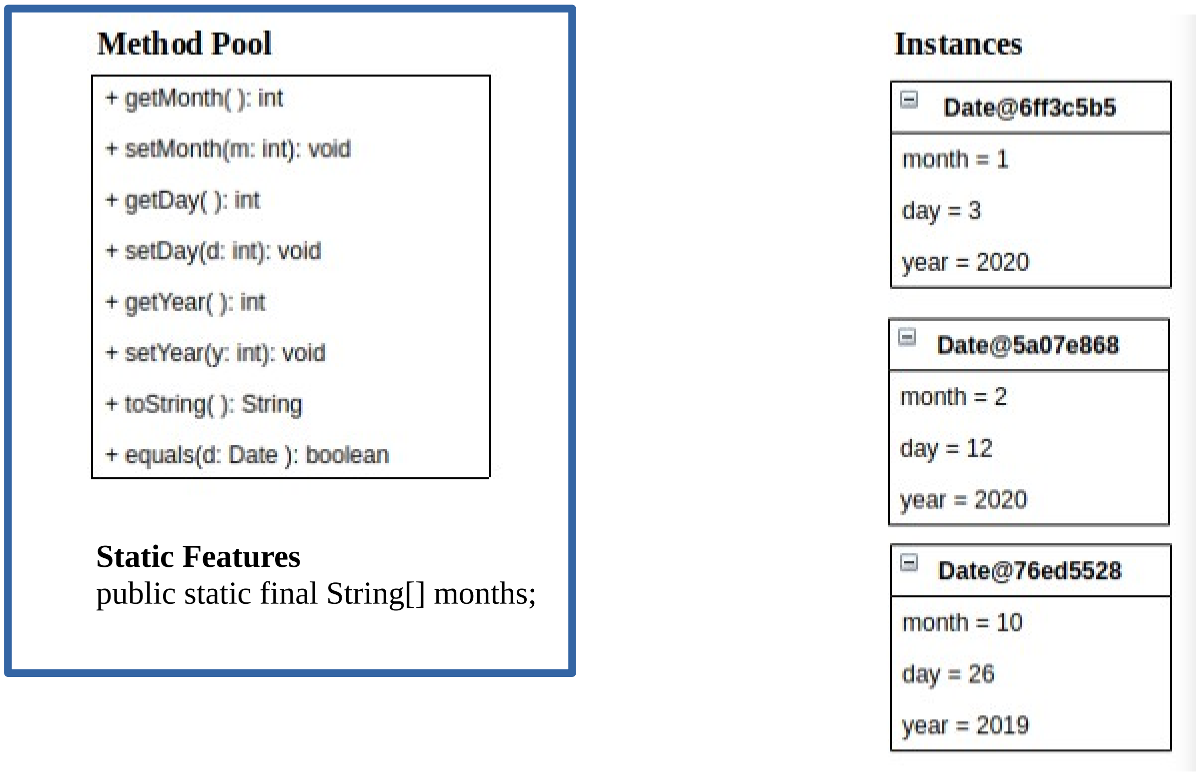
```java
// static class variables
public static final String[] months = { "January", "February", "March", "April",
                                        "May", "June", "July", "August",
                                        "September", "October", "November", "December"
                                      };
```

Design Choices

1. The array is **public**. It may be useful to provide this list of months to consumers. They could then save the time required to write this themselves
2. The array is **static**. The keyword **static** defines the ownership of the feature. Is it an instance feature that is duplicated and given to each instance, or is it a class feature that is not duplicated? A **static** feature is owned by the class. Only a single instance of a static feature will ever exist in memory. This makes sense for this array. There is no need for every single Date instance to have a copy of this array. The array should only exist once and all instances can refer back to this single array. “Static” accomplishes this for us.
3. The array is **final**. Because it is public we should take some precautions to prevent it from being changed. The **final** keyword makes things **constant and unchanging.**

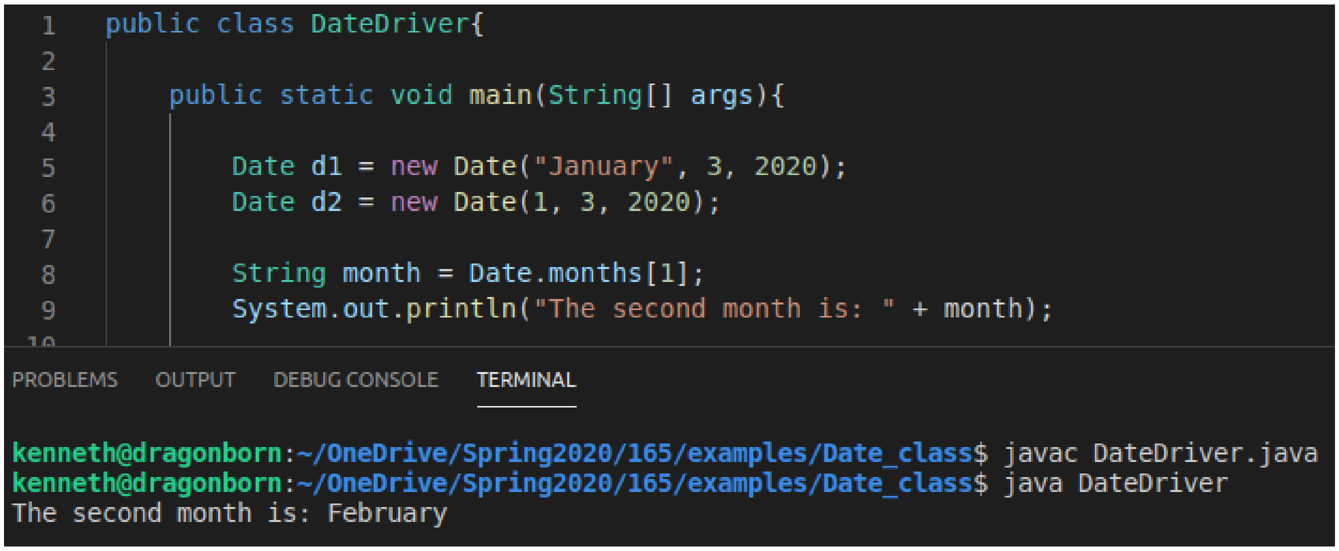
Notes on static class features

I want this issue to be clear. Static features are stored with the class. The class actually gets loaded into memory when the first instance is created. This is where static features reside. They are not part of an instance.



Referencing Static Features

This ownership issue can be further demonstrated syntactically. If a feature is public and static you can access this feature directly through the class. You do not need an instance. Take a look at line 8 in this snippet. We can access that array directly through the class name . . . no instance required.

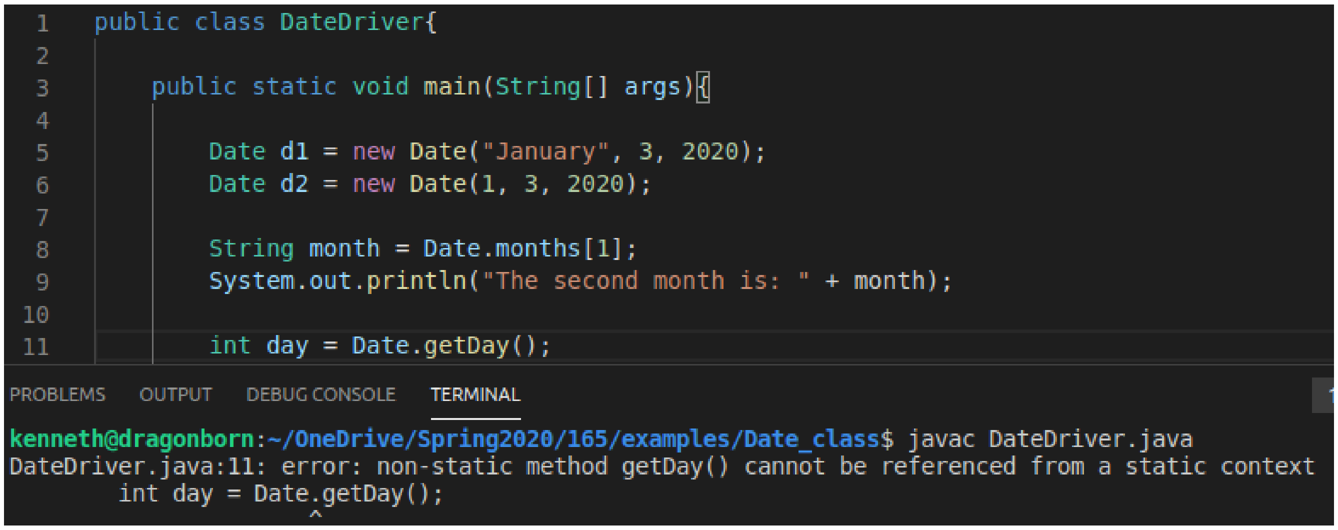


```
1   public class DateDriver{
2
3       public static void main(String[] args){
4
5           Date d1 = new Date("January", 3, 2020);
6           Date d2 = new Date(1, 3, 2020);
7
8           String month = Date.months[1];
9           System.out.println("The second month is: " + month);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ java DateDriver
The second month is: February
```

What happens if I try to access a non-static feature through the class name?



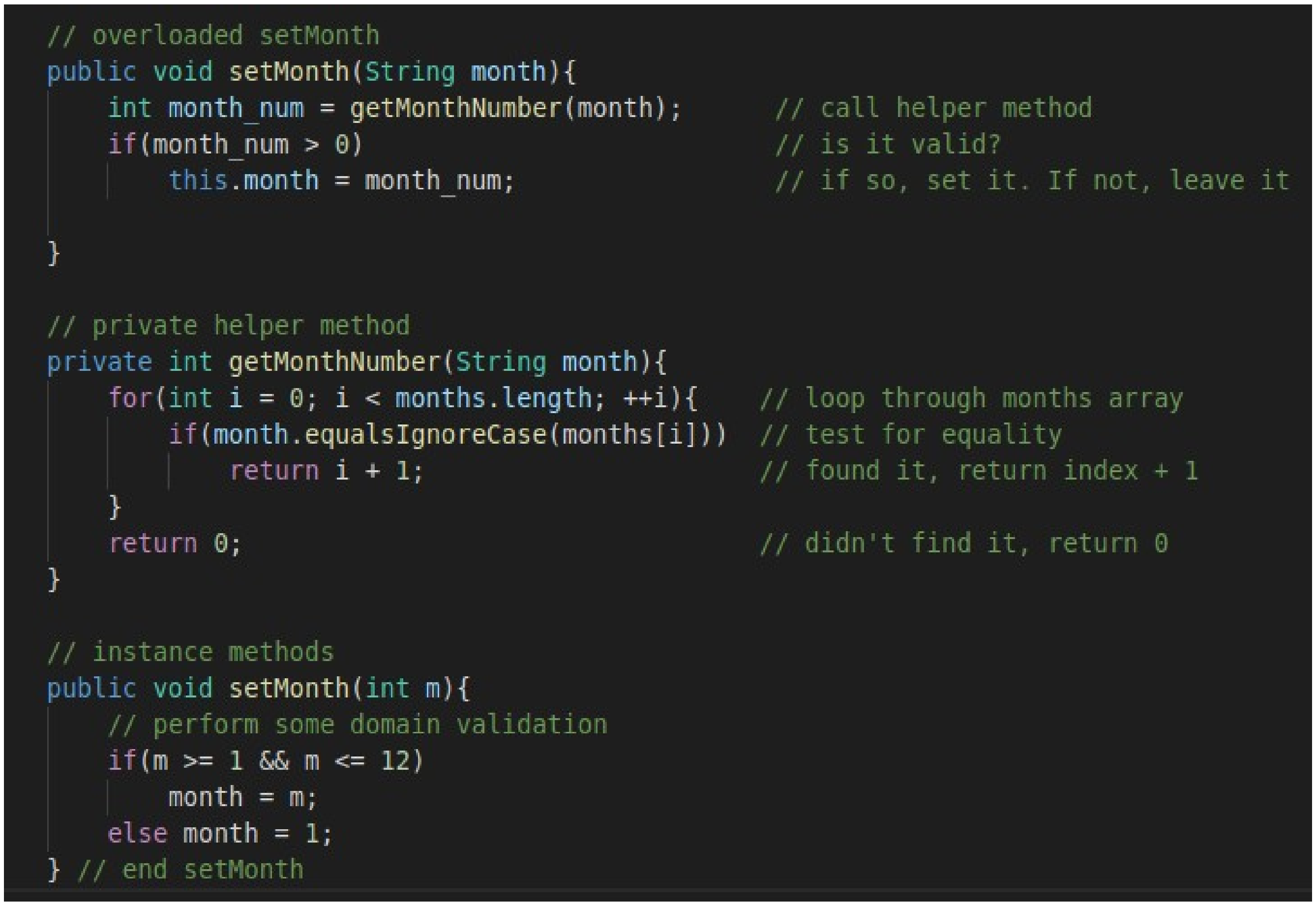
```
1   public class DateDriver{
2
3       public static void main(String[] args){
4
5           Date d1 = new Date("January", 3, 2020);
6           Date d2 = new Date(1, 3, 2020);
7
8           String month = Date.months[1];
9           System.out.println("The second month is: " + month);
10
11          int day = Date.getDay();
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
DateDriver.java:11: error: non-static method getDay() cannot be referenced from a static context
        int day = Date.getDay();
                      ^
```

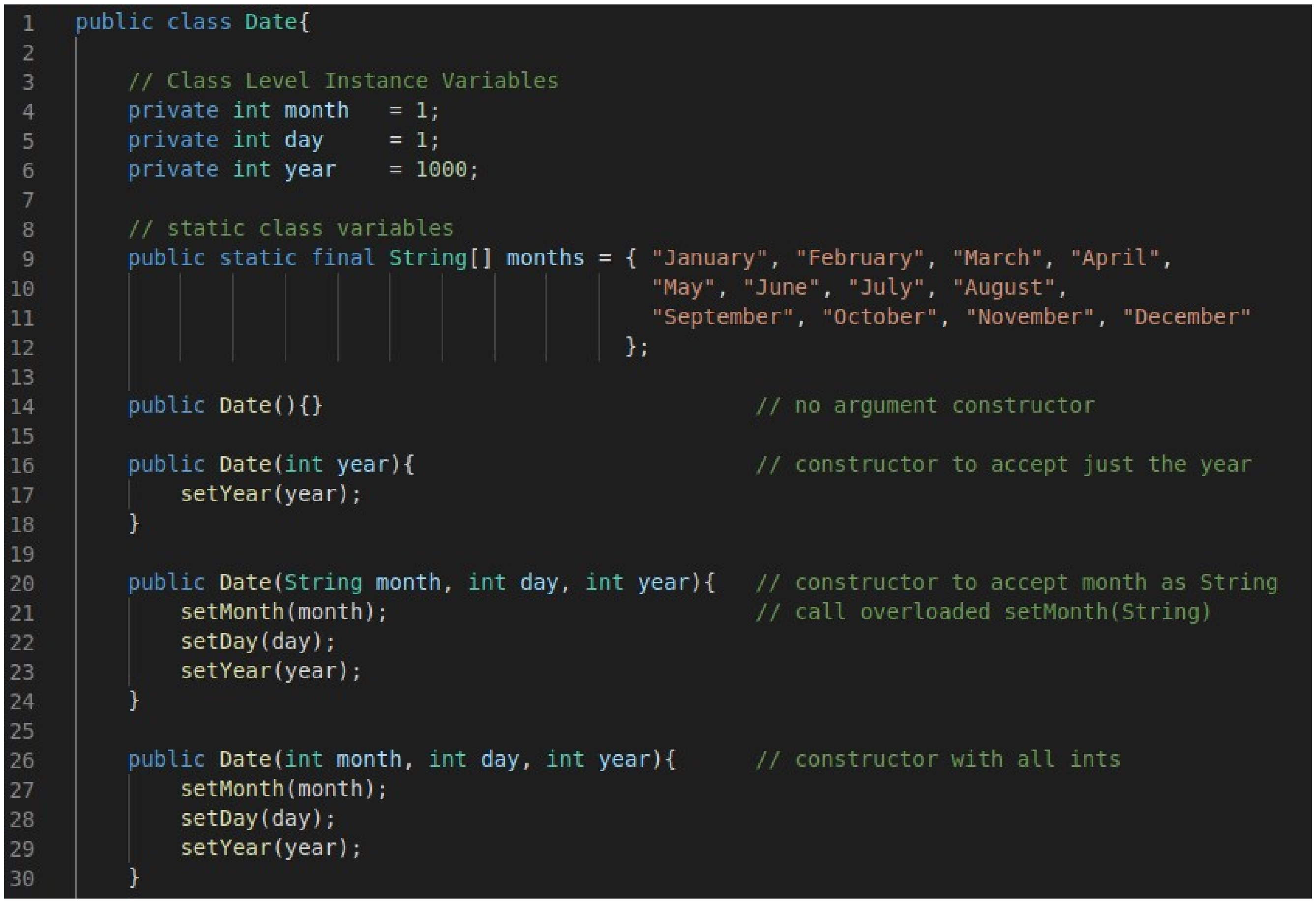
Hopefully this error makes sense. A non-static feature belongs to the instance. In order to get the day, we have to have a reference to a valid Date instance. Each instance could potentially have a different value for the day variable. Notice that the error message states that a reference through a class name is a **static context.**

Ok, back to the issue of validating a month as a string. Trace the invocation of **setMonth(String)**



```
// overloaded setMonth
public void setMonth(String month){
    int month_num = getMonthNumber(month);      // call helper method
    if(month_num > 0)                            // is it valid?
        this.month = month_num;                  // if so, set it. If not, leave it

}

// private helper method
private int getMonthNumber(String month){
    for(int i = 0; i < months.length; ++i){    // loop through months array
        if(month.equalsIgnoreCase(months[i]))  // test for equality
            return i + 1;                      // found it, return index + 1
    }
    return 0;                                  // didn't find it, return 0
}

// instance methods
public void setMonth(int m){
    // perform some domain validation
    if(m >= 1 && m <= 12)
        month = m;
    else month = 1;
} // end setMonth
```

Here is the beginning of the Date class showing the instance variables, class variables and overloaded constructors.

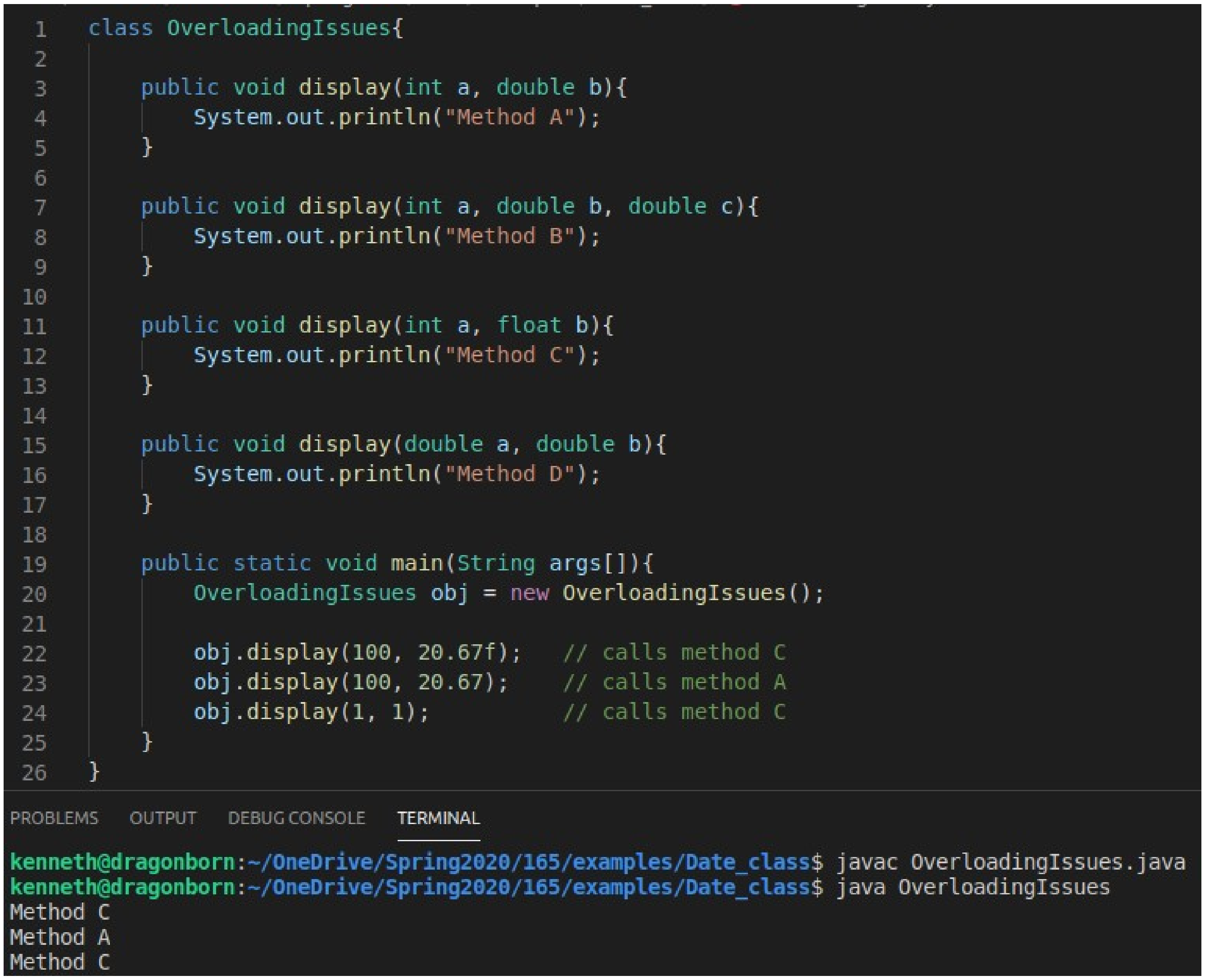


```java
public class Date{

    // Class Level Instance Variables
    private int month   = 1;
    private int day     = 1;
    private int year    = 1000;

    // static class variables
    public static final String[] months = { "January", "February", "March", "April",
                                            "May", "June", "July", "August",
                                            "September", "October", "November", "December"
                                          };

    public Date(){}                                 // no argument constructor

    public Date(int year){                          // constructor to accept just the year
        setYear(year);
    }

    public Date(String month, int day, int year){   // constructor to accept month as String
        setMonth(month);                            // call overloaded setMonth(String)
        setDay(day);
        setYear(year);
    }

    public Date(int month, int day, int year){      // constructor with all ints
        setMonth(month);
        setDay(day);
        setYear(year);
    }
```

Overloading Pitfalls
Overloading is an incredible powerful feature of a language but you have to be careful. There are some situations that could be problematic.

Method Overloading and Automatic Type Promotion
When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

This is important to understand as you may think the program will throw a compilation error but in fact the program will run fine because of type promotion. Take a look at this example. Pay close attention to the data types of each parameter and the data types of each argument and realize that automatic type promotion can allow the invocation of a method even without a direct type match in the parameter list. This is a subtle and important concept and will come back in full when we begin discussing inheritance and polymorphism.



```java
class OverloadingIssues{

    public void display(int a, double b){
        System.out.println("Method A");
    }

    public void display(int a, double b, double c){
        System.out.println("Method B");
    }

    public void display(int a, float b){
        System.out.println("Method C");
    }

    public void display(double a, double b){
        System.out.println("Method D");
    }

    public static void main(String args[]){
        OverloadingIssues obj = new OverloadingIssues();

        obj.display(100, 20.67f);   // calls method C
        obj.display(100, 20.67);    // calls method A
        obj.display(1, 1);          // calls method C
    }
}
```

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac OverloadingIssues.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ java OverloadingIssues
Method C
Method A
Method C
```

Here are some sample method overloading cases. See if you can predict the compiler outcome

Case 1:

```java
int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```java
int mymethod(int a, int b)
int mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Data types of arguments are different.

Case 3:

```
int mymethod(int a, int b)
int mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Number of arguments are different.

Case 4:

```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b)
float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.