Classes and Objects

Overview
- Defining a class creates a new object type with the same name.
- Every object belongs to some object type; that is, it is an instance of some class.
- A class definition is like a template for objects: it specifies what attributes the objects have and what methods can operate on them.
- The **new** operator instantiates objects, that is, it creates new instances of a class and returns a reference to the instance.
- Think of a class like a blueprint for a house: you can use the same blueprint to build any number of houses.
- The methods that operate on an object type are defined in the class for that object.

As we know, a Java program is a collection of interacting objects, where each object is a module that encapsulates a portion of the program's attributes and actions. Objects belong to classes, which serve as templates or blueprints for creating objects. Think of the cookie cutter analogy. A class is like a cookie cutter. Just as a cookie cutter is used to shape and create individual cookies, a class definition is used to shape and create individual objects.

Programming in Java is primarily a matter of designing and defining class definitions, which are then used to construct objects. The objects perform the program's desired actions. To push the cookie cutter analogy a little further, designing and defining a class is like building the cookie cutter. Obviously, very few of us would bake cookies if we first had to design and build the cookie cutters. We'd be better off using a pre-built cookie cutter. By the same token, rather than designing our own classes, it will be easier to get into "baking" programs if we begin by using some predefined Java classes.

So far you have gotten experience creating and interacting with various objects from the Java API

- String
- array
- Scanner
- File
- FileReader
- BufferedFileReader
- And many others you may have researched and implemented.

These objects are all built from their associated class definitions. Once the objects have been initialized we communicate with them by calling methods and passing arguments. We discovered the details of the object's interface by studying the Java API specification.

We will now examine how to create our own classes and objects and Java style API documentation. We will also develop some UML diagramming techniques along the way. Let's begin by modeling a Date class that will be used to store and process basic Date data.

When first designing a class we begin with some abstraction. As the work continues on the class, more and more details will be filled in until we have a working, tested and secure definition.

For each object, we must answer the following basic design questions:

- What role will the object perform in the program?
- What data or information will it need?
- What actions will it take?
- What interface will it present to other objects?
- What information will it hide from other objects?

For our Date object, the answers to these questions are shown below. Note that although we talk about "designing an object," we are really talking about designing the object's class. A class defines the collection of objects that belong to it (the object's *classification*). The class can be considered the object's type. This is the same as for real-world objects. Thus,

- Seabiscuit is a horse—that is, Seabiscuit is an object of type horse.
- Legolas is an elf – that is, Legolas is an object of type elf
- Aragorn is a human – that is, Aragorn is an object of type human
- Bilbo is a halfling – that is, Bilbo is an object of type halfling

Similarly, an individual Date, such as **12/31/1999** is a Date. That is, it is an object of type Date.

Class Name	Date
Role	To store, retrieve and process months, days and years as a single mutable object
Attributes	month, day and year
Behaviors	create a Date, retrieve and modify the attributes, perform some basic Date math (adding Dates together, subtracting Dates, . . . etc

The Unified Modeling Language has a standardized set of graphical objects to represent class design. Here is the beginning of our basic Date class design in UML.

Class Name: Object data type

Attributes: Instance variables

Behaviors: Methods
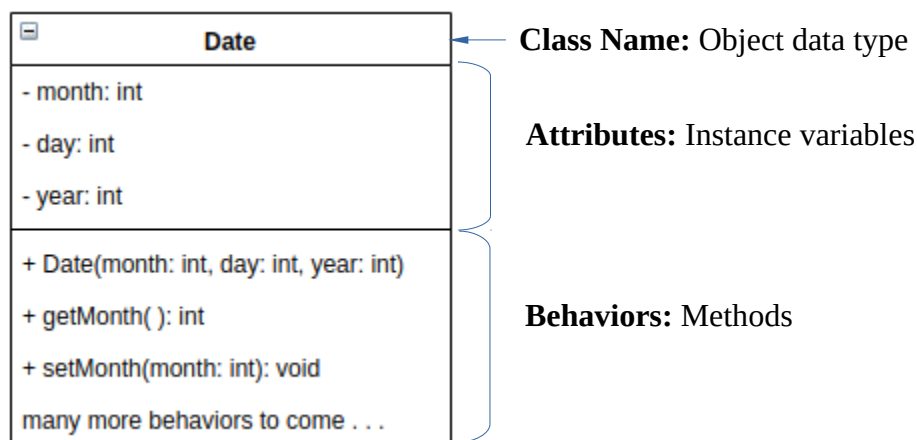
Access Modifiers: The + and – notation in UML defines access: public (+) or private (-). These access modifiers allow us to expose features available to class consumers and hide features that are for internal usage only, or important memory allocation that we wish to protect from potential modification.

Instance Variables:

We will define storage for the attributes **month, day and year** in special variables called **instance variables**. These are also know as **member variables** . . . as in **member of the class.** An instance variable is a variable for which each instantiated object of the class has a separate copy, or instance. Whenever a new object is created from the class definition, new copies of these variables will be created. The variables will be accessible to all object methods.

Instance methods are defined at **class level scope,** that is, inside the class but outside of any method definition. This scope allows access to all methods in the class.

Object Oriented Principle: Encapsulation and Information Hiding

In object-oriented programming (OOP), encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Put simply, encapsulation is about **hiding complexity**. In the real world, objects frequently hide their information and how they work. You don't need to know the internal details in order to use an object. For example, you don't need to understand how a gasoline-powered internal combustion engine works in order to drive a car. By extension, you should be able to drive a car that uses another fuel source or an electric motor without knowing how each of those work either. When you create an object in an object-oriented language, you can hide the complexity of the internal workings of the object. As a developer, there are two main reasons why you would choose to hide complexity. Another example is the Java 2D Graphics object. That class hides all of the details of the actual painting of pixels on the screen. There is no need for use to care about those details if we are simply trying to draw something.
⠀⠀⠀⠀The first goal of encapsulation is to provide a simplified and understandable way to use your object without the need to understand the complexity inside. As mentioned above, a driver doesn't need to know how an internal combustion engine works. It is sufficient to know how to start the car, how to engage the transmission if you want to move, how to provide fuel, how to stop the car, and how to turn off the engine. You know to use the key, the shifter (and possibly clutch), the gas pedal and the brake pedal to accomplish each of these operations. These basic operations form an interface for the car. Think of an interface as the collection of things you can do to the car without knowing how each of those things works. Another good example is a microwave oven . . . you do not need to understand the physics of radiation in order to heat up your cheeses dip. You just program the time by pressing buttons and then press start.
⠀⠀⠀⠀Hiding the complexity of the car from the user allows anyone, not just a mechanic, to drive a car. In the same way, hiding the complex functionality of your object from the user allows anyone to use it and to find ways to reuse it in the future regardless of their knowledge of the internal workings. This concept of keeping implementation details hidden from the rest of the system is key to object-oriented design.
⠀⠀⠀⠀The second reason for hiding complexity is to manage change. Today most of us who drive, use a vehicle with a gasoline-powered internal combustion engine. However, there a gas-electric hybrids, pure electric motors, and a variety of internal combustion engines that use alternative fuels. Each of those engine types has a different internal mechanism yet we are able to drive each of them because that complexity has been hidden. This means that, even though the mechanism which propels the car changes, the system itself functions the same way from the user's perspective.

Imagine a relatively complex object that parses an audio file and yet allows you only to play, seek or stop the playback. Over time, the author of this object could change the internal algorithm of how the object works completely—new optimization for speed could be added, memory handling could be improved, additional file formats could be supported, and the like. However, since the rest of the source code in your application uses only the *play, seek and stop methods*, this object can change significantly internally while the remainder of your application can stay exactly the same. This technique of providing encapsulated code is critical in team environments.

Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. Publicly accessible methods are generally provided in the class (so-called "getters" and "setters") to access the values, and other client classes call these methods to retrieve and modify the values within the object. In object oriented programming languages, encapsulation refers to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting direct access to some of the object's components. Java provides the access modifiers: **public, private and protected**
- A language construct that facilitates the bundling of data with the methods operating on that data. Java provides the class structure
- Many languages provide these features.

To properly restrict access to encapsulated variables we define the access as **private.** A feature marked private is only accessible **within** the class in which it is defined. This allows programmers to strictly define the ways in which a member feature can be modified.

In the Date example we would want to restrict access to the instance variables: **day, month and year** because there are strict domains that govern the values that should be allowed for each variable.

Domain: a set of allowable values for a variable

Date Domains:
- day   => the set of integers $\{1 - 31\}$
- month => the set of integers $\{1 - 12\}$
- year   => the set of integers $\{1000 - 9999\}$ or whatever your context requires.

We need to ensure that these domains are enforced at all times. The first step in enforcing this requirement is to *turn off public access* to the features.

```
1   public class Date{
2
3       // Class Level Instance Variables
4       // each new Date instance will get unique copies of these variables
5       private int month;
6       private int day;
7       private int year;
8   } // end class
```

By applying the **private** access modifier to a variable we restrict its access to **inside the class.** This will prevent any unauthorized external modification of these variables. Let's see what this looks like by creating an instance of this Date class. I will do this in a separate class called a **Driver.** This driver class will have the main method (notice that there is no main method in Date). To obviate any further configuration it is important that the files: **Date.java and DateDriver.java** be saved into the same directory

```
1   public class DateDriver{
2
3       public static void main(String[] args){
4
5           // create a Date instance using the default constructor
6           Date d = new Date();
7
8           // attempt to modify the month variable
9           d.month = 27;
10      }
11  }
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**

```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
DateDriver.java:9: error: month has private access in Date
        d.month = 27;
          ^
1 error
```

Notice that the compiler complains about our attempt to access a private variable. Hopefully you see the utility of this as I tried to set the month to the invalid value of 27. So how do we set a value of a month? Obviously we need someway to tell the instance what it's month, day and year values should be.

Instance Methods
A method in OOP is a procedure associated with a message and an object. An object consists of data and behavior. The data and behavior comprise an interface, which specifies how the object may be utilized by any of various consumers of the object. Methods also provide the **interface** that other classes use to access and modify the data properties of an object. This is an aspect of encapsulation. Unlike instance variables, there exists only a single definition of a method. The method is then associated with the correct object by the object's reference. The JVM handles this association and we need not concern ourselves with the details at this point.

Getters and Setters
One convention of object-oriented programming is to provide public methods to set and get the values of some of its private instance variables. Methods that set or modify an object's instance variables are called **mutator** methods (**setter methods**). Methods that get or retrieve the value of an instance variable are called **accessor** method (**getter methods**). These are the companion methods that allow consumers the ability to interact with the instance variables. Getters allow for the value of an instance

variable to be gotten, or retrieved. Setters allow for an external value to passed into the class. Setter methods allow us to provide some domain validation before setting the value.

It is up to the designer of the class to determine which private variables require getter and setter methods. If you were designing a BankAccount class, you might want a public getAccountNumber() method, so that clients could retrieve information about their bank accounts, but you would probably not want a public getAccountPassword() method or a public setAccountBalance() method. Context always dictates the methods that need to be defined. You need to thoroughly understand the problem before beginning any coding.

In general (though not ALWAYS)
- a setter method will be void and accept an argument. If the argument passes inspection, its value will be applied to the private instance variable
- a getter method will be non-void and not accept an argument. Getters simply return the values of instance variables.

```
1   public class Date{
2
3       // Class Level Instance Variables
4       // each new Date instance will get unique copies of these variables
5       private int month;
6       private int day;
7       private int year;
8
9       // instance methods
10      // these methods form the class' forward facing interface
11      // and are generally public
12      public void setMonth(int m){
13          // perform some domain validation
14          if(m >= 1 && m <= 12)
15              month = m;
16          else month = 1;
17      } // end setMonth
18  } // end class
```

Method definition. Public access exposes it to the outside world

Note: This method mutates the value of the private instance variable **month.** This type of action is called a **side effect . . .** the method has the side effect of changing the value of variable of a different scope. The local instance variable *month* is only assigned if the parameter *m* passes our domain validation. What happens if the domain validation fails is up to your team. For simplicity purposes I will simply set it to the minimum value.

Java Methods: Method definition consists of a method signature and a method body. The signature defines its access, ownership, return type, name and formal parameter list.

public static int methodName(int a, int b) {
    // body
}

Here
- **public:** access modifier. Public exposes it to the outside world
- **static:** class vs instance ownership (more on this later). Static defines class ownership, non-static defines instance ownership
- **int:** return type. Method will give you this when it returns
- **methodName:** name of the method
- **int a, int b:** list of typed parameters

Now that we have a setter method associated with an instance variable we can send our Date object a message to set this value, along with the value we wish to be set. Because the data is now passed through a method, we force the external data through our domain validation. In this example our domain of valid months $\{1 - 12\}$ is being tested. If the argument is in the domain set it is allowed. If it falls outside the domain set, the value will be set to some default value; in this case 1.

```
1    public class DateDriver{
2
3        public static void main(String[] args){
4
5            // create a Date instance using the default constructor
6            Date d = new Date();
7
8            // tell d to set it's month value          Method call
9            d.setMonth(12);
10       }
11   }
```

Let's add a getter method to retrieve the value of month. The getMonth() method is required if we wish to allow access to the month value. You do not have to define getters and setters for a value if your context does not need this behavior.

```
9        // instance methods
10       // these methods form the class' forward facing interface
11       // and are generally public
12       public void setMonth(int m){
13           // perform some domain validation
14           if(m >= 1 && m <= 12)
15               month = m;
16           else month = 1;
17       } // end setMonth
18
19       public void getMonth(){
20           return month;
21       } // end getMonth
```

Let's test these methods in our DateDriver class.

```
 1   public class DateDriver{
 2
 3       public static void main(String[] args){
 4
 5           // create a Date instance using the default constructor
 6           Date d = new Date();
 7
 8           // tell d to set it's month value
 9           d.setMonth(12);
10
11           // tell d to give use its month
12           int m = d.getMonth();
13
14           System.out.println("The month is: " + m);
15       }
16   }
```

Method calls . . . setters are void and getters return values

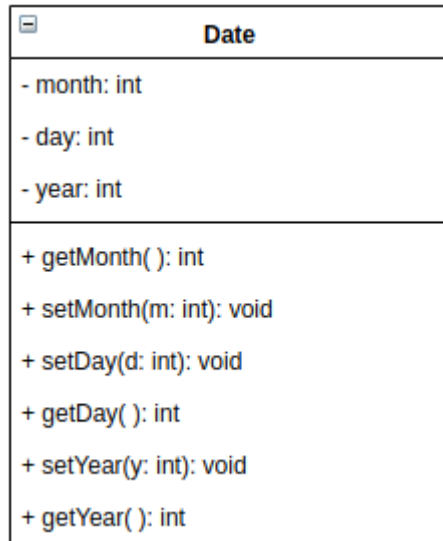PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ java DateDriver
The month is: 12

The full code listing for our class follows. Pay close attention to the public and private features. The public features form the object's interface and the private instance variables maintain the object's state. The principle of *information hiding* dictates this.

```java
public class Date{

    // Class Level Instance Variables
    private int month;
    private int day;
    private int year;

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

    public int getMonth(){
        return month;
    } // end getMonth

    public void setDay(int d){
        if(d >= 1 && d <= 31)
            day = d;
        else day = 1;
    } // end setDay

    public int getDay(){
        return day;
    } // end getDay

    public void setYear(int y){
        if(y >= 1000 && y <= 9999)
            year = y;
        else year = 1000;
    } // end setYear

    public int getYear(){
        return year;
    } // end getYear

} // end class
```

The UML class diagram for our Date class now looks like this.


```
┌─────────────────────────────┐
│ ⊟         Date              │
├─────────────────────────────┤
│ - month: int                │
│ - day: int                  │
│ - year: int                 │
├─────────────────────────────┤
│ + getMonth( ): int          │
│ + setMonth(m: int): void    │
│ + setDay(d: int): void      │
│ + getDay( ): int            │
│ + setYear(y: int): void     │
│ + getYear( ): int           │
└─────────────────────────────┘
```

The toString Method

There are a variety of methods that are expected to exist in our class designs. As we proceed through the course these methods will present themselves. The most basic method is **toString**. This method exists to create a nicely formatted String representation of our object's **state.**

State: An object's state is the current value of all relevant instance variables.

This method allows programmers to dictate how their objects are treated by output routines like **System.out.println.** Let's see what we can do so far

```
1    public class DateDriver{
2
3        public static void main(String[] args){
4
5            // create a Date instance using the default constructor
6            Date d = new Date();
7
8            // set d to 1/2/2020
9            d.setMonth(1);
10           d.setDay(2);
11           d.setYear(2020);
12
13           System.out.println("The date is: " + d);
14       }
15   }
```

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 date is: Date@4b1210ee
```

You should recognize this output as being the *Class type of object "d" followed by its hash code*. This doesn't do us much good. We could format the the date ourselves by calling the get methods and applying formatting but this would be tedious to perform every time and would not be convenient for class consumers. It would be better to define this behavior as part of the Date class. This is what the **toString** method is for. Here is the toString for the Date class.

```
40      public String toString(){
41          return month + "/" + day + "/" + year;
42      }
```

Adding this to our Date class changes the Driver behavior without us having to change the Driver code at all (anther benefit of information hiding). This is because the JVM automatically calls the **toString** on an object anytime it is placed in a String context. **Due to this it is important that your toString signature be exact!**

toString cannot be

- ToString
- to_string
- or any other spelling

```
1   public class DateDriver{
2
3       public static void main(String[] args){
4
5           // create a Date instance using the default constructor
6           Date d = new Date();
7
8           // set d to 1/2/2020
9           d.setMonth(1);
10          d.setDay(2);
11          d.setYear(2020);
12
13          System.out.println("The date is: " + d);
14      }
15  }
```

```
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 date is: 1/2/2020
```

The equals method
Another required method is the equals method. Saying "required" is a bit of a stretch here because it is not always necessary to be able to compare two objects. For the Date class it is necessary because comparing two dates is a common operation.

For instance . . . if you were writing an application to track rentals you would need to be able to ask questions like

- How many days until this item is due?
- Was this item returned before the due date.
- Did the shipment arrive by the advertised shipping date.
- We'll learn how to order dates when we study the comparable pattern

The default equality operator (==) only works on primitive values. Object references are primitive integers. Our two Date objects have the **same state** but are physically distinct objects residing at different addresses.

Remember: Comparing objects with == is a **shallow comparison** and only tests the references and will return true if the object is aliased by multiple references.

The purpose of the equals method is to provide a routine to perform a **deep comparison**. This type of comparison will cycle through the instance variables performing appropriate comparisons on each. This usually takes the form of a compound boolean "and" expression.

Examine the following Driver output to see why we need an equals method.

```java
public class DateDriver{

    public static void main(String[] args){

        // create a Date instance using the default constructor
        Date d = new Date();
        Date d2 = new Date();

        // set d to 1/2/2020
        d.setMonth(1);
        d.setDay(2);
        d.setYear(2020);

        // set d2 to 1/2/2020
        d2.setMonth(1);
        d2.setDay(2);
        d2.setYear(2020);

        if(d == d2) System.out.println(d + " equals " + d2);
        else System.out.println(d + " does not equal " + d2);
    }
}
```

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
1/2/2020 does not equal 1/2/2020
```

Equals method syntax

Access: public
Return Type: boolean
Name: equals
Argument: Instance of the class you are comparing

Note: Custom objects can be passed to methods just like any other data. When you do this you are passing the **reference** into the method. The object then becomes aliased by the local argument identifier. You can call methods on this object through the alias. If you are in the **same class** as the one that defines the object, you can also manipulate private members.

```java
public boolean equals(Date d){
    return  this.day    == d.day    &&
            this.month  == d.month  &&
            this.year   == d.year;
}
```

Notice that we can directly access d's instance variables **because this code is in the same class**, therefore we are not violating privacy as we are in direct control of the source.

Adding this method definition to our class gives us the ability to perform **deep comparisons.** Let's see this in action

```java
9           // set d to 1/2/2020
10          d.setMonth(1);
11          d.setDay(2);
12          d.setYear(2020);
13
14          // set d2 to 1/2/2020
15          d2.setMonth(1);
16          d2.setDay(2);
17          d2.setYear(2020);
18
19          // test using ==
20          System.out.print("Using == ");
21          if(d == d2) System.out.println(d + " does equal " + d2);
22          else System.out.println(d + " does not equal " + d2);
23
24          // test using equals method
25          System.out.print("Using .equals ");
26          if(d.equals(d2)) System.out.println(d + " does equal " + d2);
27          else System.out.println(d + " does not equal " + d2);
28      }
29  }
```

```
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
Using == 1/2/2020 does not equal 1/2/2020
Using .equals 1/2/2020 does equal 1/2/2020
```

The "this" keyword

If you were paying close attention you noticed the **this** keyword in the equals method. "This" is colloquially known as a reference **to the current object.** In this case, the current object is the one through which we called the equals method . . . **the object the method was invoked upon**. We can calling this the **calling object.** I used this to draw attention to the different instance variables being compared. Remember, each instance has its own copies of these variables and we need to be specific with our references. One "**day**" variable belongs to the argument **d**, the other day variable belongs to the instance through which we called equals . . . **this**.

```java
public boolean equals(Date d){
    return  this.day    == d.day    &&
            this.month  == d.month  &&
            this.year   == d.year;
}
```

In our example **this** refers to instance **d in the Driver** while Date d in the equals method refers to instance **d2 in the Driver**. This association is created by

- The object you use to call the method (d)
- The object you pass as an argument to the method (d2)
- This ordering and aliasing is crucial to understand

Constructors

As it stands we only have one way to pass values into our Data class: the setters. This requires 4 lines of code to create and initialize a Date object

```java
// create a Date instance using the default constructor
Date d = new Date();
Date d2 = new Date();

// set d to 1/2/2020
d.setMonth(1);
d.setDay(2);
d.setYear(2020);

// set d2 to 1/2/2020
d2.setMonth(1);
d2.setDay(2);
d2.setYear(2020);
```

This is tedious and cumbersome. Luckily OO languages provide a special method for building instances: **the Constructor**. A class' constructors are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. We have already seen a constructor for our Date class . . . the default constructor.

Default Constructor: If no constructor is explicitly defined in a class, the java compiler will provide one that accepts no arguments and performs no initialization.

Notice that we did not define a constructor but were still able to construct instances. Thank you default constructor! But as we have seen, we need to provide convenient methods of instantiation. Let's define our own constructor . . . one that will allow us to provide initialization code. One that will allow us to pass in the values of month, day and year on the same line as our object creation. Something like this

```
// create a Date instance using custom constructor
Date d = new Date(1, 2, 2020);
```

The signature of this constructor in our Date class will be:

```
public Date(int month, int day, int year){
```

Constructor Rules:

1. They have the same name as the class
2. They do not have a return type
3. They can accept arguments like any other method
4. They should be public, unless you want to prohibit object creation. This is a thing. Try to create an instance of the Math class . . . I dare you.

Constructors can set instance variables like regular methods. So we could define our Date constructor like this (**this.month** refers to the class level instance variable and allows us the ability to distinguish between the argument and the instance variable)

```
public Date(int month, int day, int year){

    this.month  = month;
    this.day    = day;
    this.year   = year;

}
```

But this would bypass all the domain validation we performed in our setter methods. Given the rule of programming

NEVER REPEAT YOURSELF

How can we validate a constructor's input? We can simply have the constructor call our domain validation methods.

```java
public Date(int month, int day, int year){
    // call setter methods to route
    // constructor args through domain validation
    setMonth(month);
    setDay(day);
    setYear(year);
}
```

Let's try this out with our existing driver

```java
public class DateDriver{

    public static void main(String[] args){

        // create a Date instance using custom constructor
        Date date = new Date(1, 2, 2020);


        Date d = new Date();
        Date d2 = new Date();

```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
DateDriver.java:9: error: constructor Date in class Date cannot be applied to given types;
        Date d = new Date();
                 ^
  required: int,int,int
  found:    no arguments
  reason: actual and formal argument lists differ in length
DateDriver.java:10: error: constructor Date in class Date cannot be applied to given types;
        Date d2 = new Date();
                  ^
  required: int,int,int
  found:    no arguments
  reason: actual and formal argument lists differ in length
2 errors
```

You'll notice that we get 2 errors when compiling this code. These errors have to do with code that we used before . . . with no errors. Huh?

Note about default constructors: The default constructor is only supplied when you do not write a constructor yourself. When you do write your own constructor the **java compiler does not provide one.** Closely read the error messages.

If you don't want your earlier code to break you can provide an additional constructor that has that signature. This constructor is called the **no argument constructor** and generally looks like this.

```java
public Date(){} // no argument constructor
```

You can have multiple constructors in the same class as long as the signature is different. This is called **overloading** and also applies to instance methods. This will be covered in more detail in a subsequent document.

```java
1    public class Date{
2
3        // Class Level Instance Variables
4        private int month;
5        private int day;
6        private int year;
7
8        public Date(){} // no argument constructor
9
10       public Date(int month, int day, int year){
11           // call setter methods to route
12           // constructor args through domain validation
13           setMonth(month);
14           setDay(day);
15           setYear(year);
16       }
```

The previous driver code now compiles error free. These multiple constructors are for convenience to any consumer who may interact with our class.

```java
1    public class DateDriver{
2
3        public static void main(String[] args){
4
5            // create a Date instance using custom constructor
6            Date date = new Date(1, 2, 2020);
7
8            Date d = new Date();
9            Date d2 = new Date();
10
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ javac DateDriver.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/Date_class$ []

Our updated UML class **interaction diagram**:
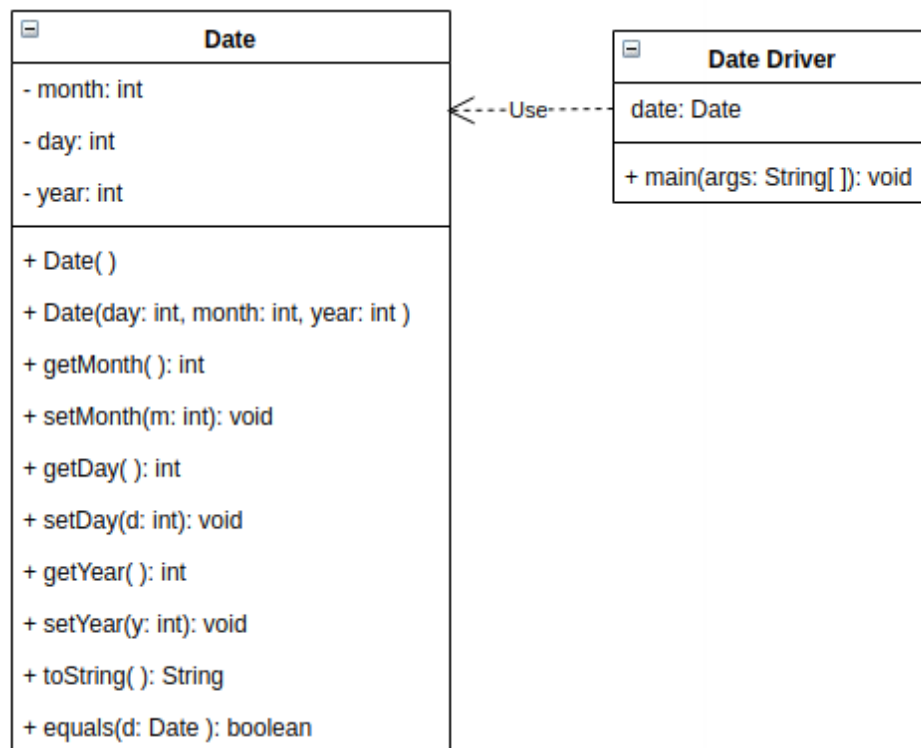
Additions to this class will be left as an exercise. Think about useful features that could be added.

- Different date formats: long, short
- Months as Strings
- Comparing two dates
- Ordering two dates
- Compute the difference between two dates
- Add days to a date
- Add two dates together.
- Determine the day of the week
- Is the year a leap year