Modifiers:

In Java, there are a set of **access modifiers**.

Types of access modifiers:

1. Default – No keyword required
2. Private
3. Protected
4. Public

- An access modifier defines the accessibility of a class, method, or data member.
- In other words, the access modifier will indicate where a programmer can reference or refer to a particular class, method, or data member.
- For example, the main() method in a Java program is defined as **public**. This means that the main method may be accessed (called, invoked) anywhere in the program's code, and also anywhere in another program's code.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

public Classes, Methods, and Instance Variables

A class that is defined as "**public**" may be instantiated by any other class. Applets must be defined as public classes because they have to be instantiated by a user's browser. A class can be defined with no modifier, such as this one:

```
class MyMessage
{
      // ... class code goes here ...
}
```

- This class is not available to all other classes;
- it is only available to classes in the same package.
- We refer to this as **default**, **friendly**, or **package** access. For our use, this means that this class is available only to classes in the same folder. This is technically slightly incorrect, but we'll talk about this when we learn to create our own packages later on.

A method that is defined as public can be invoked or called by any other code in the class or program where the method is defined, or even from other classes and programs. For example, a class MyMessage has a public method called "displayMessage()":

```
public class MyMessage
{
    public String message;
    public void displayMessage()
    {
        System.out.println(message);
    }
}
```

This displayMessage() method can be accessed by another Java class or program:

```
public class TestMyMessage
{
    public static void main(String[] args)
    {
        MyMessage hello = new MyMessage();
        hello.message = "Hello, World!";
        displayMessage();
    }
}
```

An instance variable defined as public can be read or written to by any other class or program. This is usually an undesirable situation, as we'll learn in the next couple of sessions, however the previous example demonstrates the use of a public variable.

The MyMessage class has a public variable called "message". We can assign a value to this variable (as we did in the statement **hello message = "Hello, World!";**) and we can also read the value in the variable and use it (as we could do in a statement such as **String newMessage = "kaluha! " + hello.message;**).

Note
by declaring them at the top of the program, under the class header, but you would declare them as "private" as we'll see later.

private Methods and Instance Variables

Notice that we did not include classes in this sub-heading! That's because classes are generally not defined as private! In fact, a class can only use the "public" access modifier unless it's an *inner class*. You'll learn about inner classes in the second course.

A method that is defined as private in a class or program is only available inside that class or program. For example, create the following class and program:

```
public class Book {
public String bookTitle;
    public double bookPrice;

    private void displayBook() {
        System.out.println(bookTitle + ": $" + bookPrice);
    }
}
public class TestBook {
    public static void main(String[] args) {

        Book javaText = new Book();
        javaText.bookTitle = "OOP Develpment Using Java";
        javaText.bookPrice = 122.95;
        javaText.displayBook();
    }
}
```

When you compile both programs, you'll get the following error:

```
TestBook.java:17: displayBook() has private access in
Book
                javaText.displayBook();
                         ^
```

This error is telling you that you can't invoke the javaText.displayBook() method because it's defined as private in the Book class. You can't access a private method from outside the class where it is defined.

Why would you want to define a private method in a class? Later we'll talk about **encapsulation**, which is the main reason behind private members of classes and programs. Simply, a private method is usually a "helper" method that performs a task to help out the tasks performed in other methods in the same class or program. The following scenario should help understand this:

Say you are asked to write a program that makes use of an Employee class that another programmer has already written. Your program prints the pay checks for the employees in the company. You read the documentation for the Employee class and see that there are some data members and some methods that are going to be useful to you. One of them is a method that prints a pay check.

This printPayCheck() method accesses the employee database and gathers the necessary information, then calculates all the fields necessary for a pay check, such as federal tax deductions, union dues, pension plan deductions, gross pay, and net pay. This printPayCheck() method is probably made up of many code statements that call other methods, such as the method to calculate the federal tax deduction. As a programmer using the Employee class, you don't need to know about or even care about those methods; you are only interested in the public method that prints a pay check. The little methods that "help" the printPayCheck() method are of no use to other programmers, only to the printPayCheck() method inside the Employee class. Therefore, those "helper" methods will be defined as private, so they can't be accessed outside the Employee class.

As we'll see later with instance variables, members of a class are often defined as private in order to protect instances of the class from being maliciously or accidentally altered in some way.

Private instance variables work the same way as private methods: they are only accessible inside the class and are not available to code in other classes or programs. You might wonder, why on earth would we add a data member to a class if it's not to be accessed by another program or class? In fact, this is a standard practice! In order to protect a piece of data from being given an incorrect value, we often make the data member private, and then we provide an "accessor method" for that data member. The accessor method will accept the value you want to put into the data member, make sure it's valid, and place the valid value in the data member itself. We'll learn about accessor methods in a later session.

Instance Variables

When we define the data members (also called properties or attributes) for a class, we define **instance variables**. These are not the same as the variables we have been defining in a program's main() method; those variables were **local variables**. A few of the significant differences between local variables and instance variables are:

Instance Variables	Local Variables
Represent data members for a class.	Are used to temporarily hold values (such as user inputs, or results of calculations) inside a method.
Can be defined with ***modifiers*** such as **public** and **private**.	Are not defined with ***modifiers*** because they are only accessible inside the the method in which they are defined.
Do not need to be initialized.	Must be initialized or assigned a value before they are used in an expression.
Should be defined all together at the top of the class, under the class header.	Can technically be declared anywhere in the method, although sometimes we prefer to declare them all together at the top of the method under the method header.

Local variables can be thought of as "working" variables; we use them to hold data values temporarily until we are ready to use them later in our code. Instance variables are the pieces of data that contribute to the definition of a particular object; they are part of our class "recipe" or template.

Accessor and Mutator Methods

As previously discussed, in a well-designed class, the programmer should keep the instance variables private, so that access to them is limited. Recall that private instance variables can't be accessed outside the class in which they are defined; in other words, no other class or program can read or change the value of a private instance variable. We need to make instance variables private in order to protect them from receiving invalid data. One problem with this of course is that other classes and programs can't access the variables even for legitimate purposes. For this reason, we provide accessor and mutator methods as gateways or filters to the private instance variables.

Code the following test program for your Room class, then compile them both, and run the program:

```
public class TestRoom {

    public static void main(String[] args) {

        Room office = new Room();
        office.length = 16.5;
        office.width = 20;
        System.out.println("Area: " + office.calcArea());
    }
}
```

When we run the program, we get the following error messages:

```
TestRoom.java:15: length has private access in Room
                classroom.length = 16.5;
                          ^
TestRoom.java:16: width has private access in Room
                classroom.width = 20;
                          ^
```

```
2 errors

Press any key to continue...
```

We've discussed this kind of error before: You can't access the private instance variables length and width ouside of the Room class. So how do we give them values? How do we retrieve those values later if we want to display them? This is where accessor and mutator methods come in.

Mutator Methods

Mutator methods are special methods that put values into private instance variables. Many mutator methods will also validate the data going into a variable, for example when we made the Circle class, we decided that if a radius value was 0 or less, we would use a default value of 1 instead. If you were writing an employee class, you might want to make sure that an employee object's rate of pay was always greater than 0 and never greater than 100.00. In our Room class, we might decide that 0 or negative values are invalid for the length and width instance variables. We can program our mutator methods to check the values first, then either assign them if they're valid, or use a default if they're not.

There are rules you must follow when defining a mutator method for a class. Some of these are stylistic rules but a few are syntax rules:

- There must be one mutator method for each private instance variable in your class if you want to provide write-access to that variable by other classes/programs.
    - To create a read-only instance variable, simply don't define a mutator method! Without a mutator method, no other class or program can change the value of an instance variable.
- A mutator method always accepts one argument, for which there must be a parameter variable in the method's header.
    - The parameter variable's data type must match the data type of the instance variable for this mutator.
- A mutator method always returns type void.
- A mutator method's name always starts with "set" (lower-case) and then the name of the instance variable, starting with a capital letter.
    - This is why mutators are sometimes called "set methods".
    - Examples:

- instance variable: price -- mutator method: setPrice()
- instance variable: radius -- mutator method: setRadius()
- instance variable: firstName -- mutator method: setFirstName()

- Mutators should be public. They must be accessible to other classes and programs.

Note
Some programmers use the terms "accessor methods" or "accessors" refer to **both** mutator methods

and accessor methods! This is because both kinds of methods allow access to private instance variables, regardless of whether that's write access or read access. Be careful if you hear the term "accessor methods" - in many cases, this is referring to both kinds of methods. Make sure you understand the context in which this term is used, so you know whether the reference is to both kinds of methods, or only the "set" methods.

Let's create to very basic mutator methods for the Room class that set the values of length and width, but only if the programmer-specified values are positive. If the parameter is 0 or negative, use a default value of 1:

```
public void setLength(double length)
{
    if (length > 0)
        this.length = length;
    else
        this.length = 1;
}

public void setWidth(double width)
{
    if (width > 0)
        this.width = width;
    else
        this.width = 1;
}
```

To use these methods, you simply call them as you would any other method:

```
Room classroom = new Room();

classroom.setLength(16.5);

classroom.setWidth(20);

classroom.displayDimensions();
```

Try it with invalid values for length and/or width and see what output you get.

Accessor Methods

Accessor methods are similar to mutator methods, except that instead of setting the values of instance variables, accessor methods allow another class or program to read the values of instance variables. An accessor method returns the value of a particular instance variable, so as with mutators, there are a set of rules that need to be followed:

- There must be one accessor method for each private instance variable in your class if you want to provide read access to that variable by other classes/programs.
- An accessor method accepts no arguments, so it does not require any parameter variables.
- An accessor method always returns a data type, never void.
  - The return data type must match the data type of the instance variable for this accessor.
- An accessor method must always return the value of its instance variable using the return statement.
- An accessor method's name always starts with "get" (lower-case) and then the name of the instance variable, starting with a captial letter.
  - This is why accessors are sometimes called "get methods".
  - Examples:
- instance variable: price -- accessor method: getPrice()
- instance variable: radius -- accessor method: getRadius()
- instance variable: firstName -- accessor method: getFirstName()
- Accessors should be public. They must be accessible to other classes and programs.

The accessor methods for our Room class would be defined as:

```
public double getLength()
{
    return length;
}

public double getWidth()
{
    return width;
}
```

Modifiers

Modifiers and their meanings:

Modifier	Used With			Explanation
	class	method	data	
(default)	X	X	X	A class, method, or data field is visible in this package.
public	X	X	X	A class, method, or data field is visible to all the programs in any package.
private		X	X	A method or data field is only visible in this class.

`protected`		X	X	A method or data field is visible in this package and in subclasses of this class in any package.
`static`		X	X	Defines a class method or a class variable.
`final`	X	X	X	A final class can't be subclassed. A final method can not be modified or overridden in a child class. A final data member is a constant.
`abstract`	X	X		An abstract class must be a parent class, you can't instantiate an abstract class. An abstract method must be implemented in a concrete subclass.
`native`		X		A method defined as native is implemented using another language.
`synchronized`		X		A synchronized method cannot be executed by more than one thread at the same time.

Throwing Exceptions

For this section, you'll need your Circle, Room, and Time classes.

Up until now, our mutator methods perform validation by using the conditional operator or if-statement to assign a default value to the instance variable. A more professional way of handling this problem is to inform the programmer using our class that they've tried to place an invalid value into an instance variable. You might think this could be done in the following way:

```
1
2
3
4
5
6
```

```
    public void setRadius(double radius) {
        if (radius <= 0)
            System.out.println("Invalid value for radius!");
        else
            this.radius = radius;
    }
```

The problem with this solution is that we're forcing the programmer to use the console. Similarly, we can't use JOptionPane's input dialog or we'll be forcing the programmer to use a GUI environment. The whole point of designing object-oriented programs is to write classes that can be used in any environment, whether it be command-line, GUI, or applet. If our classes use the console or dialog boxes, we limit the flexibility and usability of our classes.

Another solution might be to have the method return a string with the error message, but not only would that violate the rules regarding the format of mutator methods, but it would also mean the programmer would have to always use an if-statement to check to see if the method returned an error message.

A better solution is to have the method throw a special exception object called an IllegalArgumentException when the instance variable receives an invalid value. The IllegalArgumentException is a child of the RuntimeException class (which in turn is a child of the Exception class). If you check the docs for the IllegalArgumentException, you'll read that this exception class is used when "a method has been passed an illegal or inappropriate argument." We can use this exception class in our own classes by using the **throw** statement:

```
1    if (radius <= 0)
2        throw new IllegalArgumentException("Invalid value for radius.");
3    else
4        this.radius = radius;
```

This new statement says "throw an IllegalArgumentException with the error message I've specified if radius is less than or equal to 0". If you guessed that the new operator indicates that we are instantiating a new IllegalArgumentException object and then throwing that object, then you were correct!

If you prefer, you can do the validation and exception throwing in two statements:

```
    if (radius <= 0) {
1        IllegalArgumentException myEx = new IllegalArgumentException("Invalid
2    value for radius.");
3        throw myEx;
4    } else {
5        this.radius = radius;
    }
```

Either way you do it, your mutator method setRadius() is now throwing an exception if the radius parameter contains an invalid value.

The exception objects in Java all have an instance variable called "message". When you pass a String value into the exception object's constructor method (as we're doing in both examples above by passing the string "Invalid value for radius." into the IllegalArgumentException constructor), the constructor method will place that string into the exception object's **message** instance variable. This will come in handy when a programmmer is using your class in a program:

```
1
2     Scanner keysIn = new Scanner(System.in);
3     try {
4         System.out.print("Enter a radius: ");
5         int radius = keysIn.nextInt();
6         Circle circle = new Circle(radius);
7         System.out.println("Area: " + circle.getArea());
8     } catch (Exception ex) {
9         System.out.println(ex.getMessage());
      }
```

In the sample code above, the programmer decided to use a try-catch block to handle the IllegalArgumentException that might be thrown if the user enters an invalid radius (and that will also catch the InputMismatchException if the user types a value that's not a valid int!) In the catch-block, the value of the exception object's **message** data member is displayed on the console.

Even though this catch-block's parameter is an Exception object, it will still catch both an IllegalArgumentException and an InputMismatchException. This is because of *inheritance*: both exception classes are *child classes* of the Exception class. You'll learn more about inheritance and catching multiple types of exceptions in the term 2 Java course, and in Chapter 18 of your textbook.