Access Modifiers:

1. **Access Modifiers:** Private, default, protected, public: from most to least restrictive order. They are optional to use in methods and variables.
2. **Optional Specifiers:** static, abstract, final, synchronized, native, strictfp
3. Here are the rules for a method signature
   1. Return type should always come before the method name
   2. Optional specifiers can come before or after the access modifiers, i.e. **static public** or **public static** is the same.
4. Varargs: They look like String… or Int… and they are very cool to use, they have the following rules:
   * 1. You can only declare varargs as method parameters. They cannot be used as reference variables.
     2. While being passed as method parameter, varargs must be the last element to be passed
     3. Compiler will generate error if you have two overloaded methods with the same signature except one is varargs and the other is an array as both methods are same they accept arrays of a type.
     4. When calling a method with varargs parameter, you can pass in an array, you can list the elements of the array like (1,2,3,4) or you can even completely omit the parameters and just send ().
5. Out of the 4 access modifiers, protected is the trickiest one and it has some special rules we need to be careful about. The protected rules apply under two scenarios:
   1. A member is used without referring to a variable. In this case we are taking advantage of inheritance and protected access is allowed.

Ex:

Package pond.shore;

Public class Bird{

Protected String text =”floating”;  
 protected void floatInWater(){

System.out.println(text);

}

}

Package pond.goose;

Import pond.shore.Bird;

Public class Gosling extends Bird{

Public void swim(){

**floatInWater(); 🡪 Protected access to the method of parent Bird**

System.out.println**(text); 🡪 Protected access variable of parent Bird**

}

}

* 1. A member is used through a variable. In this case, the rules for the reference type of the variables are what matters. If it is a subclass, protected access is allowed. This works for references to the same class or a subclass.   
     Ex: Continuing the example above

Package pond.goose;

Import pond.shore.Bird;

Public class Gosling extends Bird{

Public void swim(){

Bird bird = new Bird();

bird.**floatInWater()**; -> **Compiler error, For reference Bird, because Bird is being used outside of pond.shore package and Bird does not extend Bird(Weird)**

bird.**text**;

// But the reference to Gosling works

Gosling gosling = new Gosling();

gosling.**floatInWater**(); -> **This is OK because the reference of Gosling is being used and Gosling extends Bird.**

gosling.**text**;

}

}

**Constructors**

1. Every Java object must have a constructor
2. If an explicit constructor is not present, java will provide a default constructor
3. This() or Super() should be the first thing in the constructor.
4. Calling this() on a parameter less constructor will generate a compile error , recursive call   
   E.x:   
   ConstructorBasic(){  
    **this**(); // GENERATES COMPILER error  
   }
5. Adding a return; statement in constructor is fine
6. If a superclass has a non-default constructor, then every constructor in the subclass needs to either call the super () constructor or call the constructor in the class that is calling the superclass constructor.

As you can see from the example below, ConstructorBasic has a non- default constructor and every constructor in ConstructorComplexity needs to somehow call the non-default constructor of ConstructorBasic. In the first constructor of ConstructorComplexity, this is happening by just calling super (1), but in the second and third constructor, this is happening by the constructors calling the this () and this(1) constructors which are internally calling the super constructor.

E.x:

**public class** ConstructorBasic {  
 ConstructorBasic(**int** number){ }  
}  
**class** ConstructorComplexity **extends** ConstructorBasic {  
ConstructorComplexity(){  
 **super**(1); // IF THIS IS REMOVED , the code will not compile   
 }  
ConstructorComplexity(**int** num){  
 **this**(); // IF THIS IS REMOVED , the code will not compile  
 }  
 ConstructorComplexity(String str){

**super**(1); // IF THIS IS REMOVED , the code will not compile  
 }  
}

General rules for methods and encapsulations:

1. If swing is a static variable in Rope object then the following does not throw a null pointer  
   Rope rope = null;   
   rope.swing;   
   This is because java looks at the type of reference for rope object and calls the static variable  
   swing.
2. Just because you are using the following import: import static java.util.Arrays.asList; , does not mean that you are importing the Arrays class . You are only importing the asList method of the Arrays class. So the following would fail:  
   E.x:  
   import static java.util.Arrays.**asList**:  
   public class BadImport{

Public static void main(String… str){

**Arrays.asList(“one”)** // This line does not compile because we are not importing Arrays class

}  
}

1. Protected and private modifiers can only be applied to inner classes.
2. The compiler performs the following checks when you override a non-private method:
   1. The method in the child must have the same signature as the method in the parent class.
   2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
   3. The method in the class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class
   4. If the method returns a value, it must be same or subclass of the method in the parent class known as covariant return types.

For static

* 1. Method defined in the child class must be marked as static if it is marked as static in parent class. Likewise, method must not be marked as static in the child class if it is not marked as static in the parent class.

1. Unlike regular overriding where a method in the child class replaces the method of the parent class, hidden methods only replace parent methods in the calls defined in the child class.

**Abstract Classes**