TA Session 8

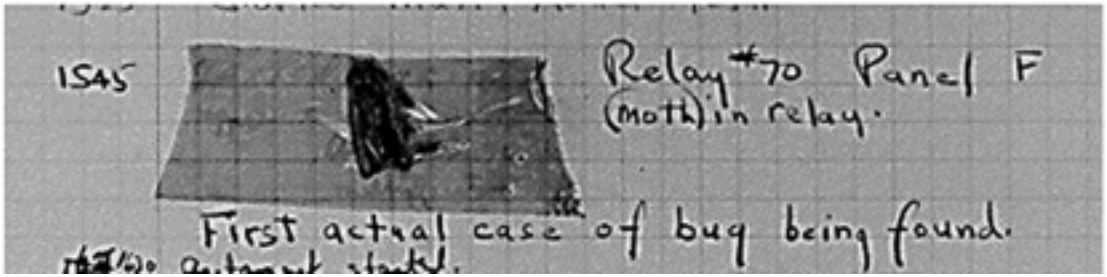
Advanced Eclipse, Debugging and Packages

Debugging, the Naïve Way

```
SomeClass c = new SomeClass();

// ..
// Some Logic
// ..

// Debug Code
System.out.println(c.getSomeField());
// Debug Code

//..

// Debug Code
if (c.getOtherField() == 0) {
    System.out.println(c.getSomeField());
}
```

Using a Debugger

- So far when we wanted to test our code we've just executed it (Using Ctrl+F11 from eclipse, for instance).
- Java code can also be executed using a **Debugger**.
- A Debugger: A piece of software that **executes** our code, and allows us to inspect the different states of our program throughout its execution.



Using a Debugger

- So far when we wanted to test our code we've just executed it (Using Ctrl+F11 from eclipse, for instance).
- Java code can also be executed using a **Debugger**.
- A Debugger: A piece of software that **executes** our code, and allows us to inspect the different states of our program throughout its execution.

Eclipse already has an **integrated** debugger.
To execute your code using a Debugger, press F11 instead of Ctrl+F11.

Debugger: Breakpoints

- Breakpoints allow you to stop the code execution and inspect the program's current state:

```
public static void main(String[] args) {

    for (int i = 0; i < 100; i++) {
        if (i % 43 == 0) {
            // Do something once in every 43
            // cycles.
            processNumber(i);

        }
    }

}
```

Debugger: Breakpoints

- Breakpoints allow you to stop the code execution and inspect the program's current state:

```
public static void main(String[] args) {

    for (int i = 0; i < 100; i++) {
        if (i % 43 == 0) {
            // Do something once in every 43
            // cycles.
            processNumber(i);

        }
    }

}
```

Ctrl+Shift+B

Debugger: Breakpoints

- Breakpoints allow you to stop the code execution and inspect the program's current state:

```
for (int i = 0; i < 100; i++) {
    if (i % 43 == 0) {
        // Do something once in every 43
        // cycles.
        processNumber(i);

    }
}
```

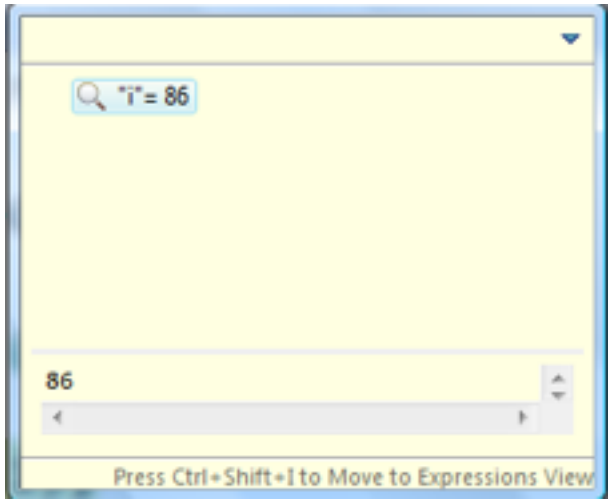
The program has halted when it reached our breakpoint.
But what can we do now ?

Debugger: Inspecting Variables

- It would be very useful to inspect the values inside the different variables.

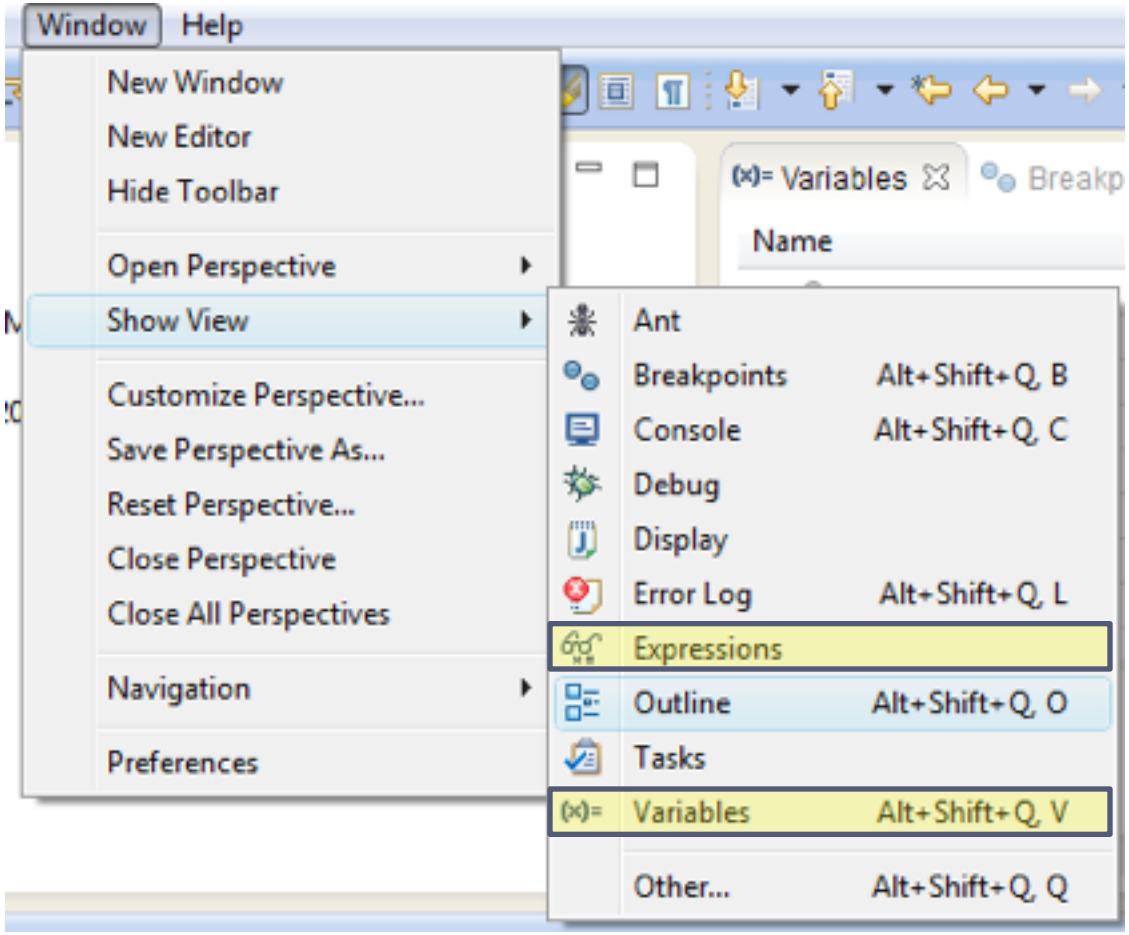
- What is the current value of "i" ?

- If we mark the variable name and press **Ctrl+Shift+I** we'll get the following:



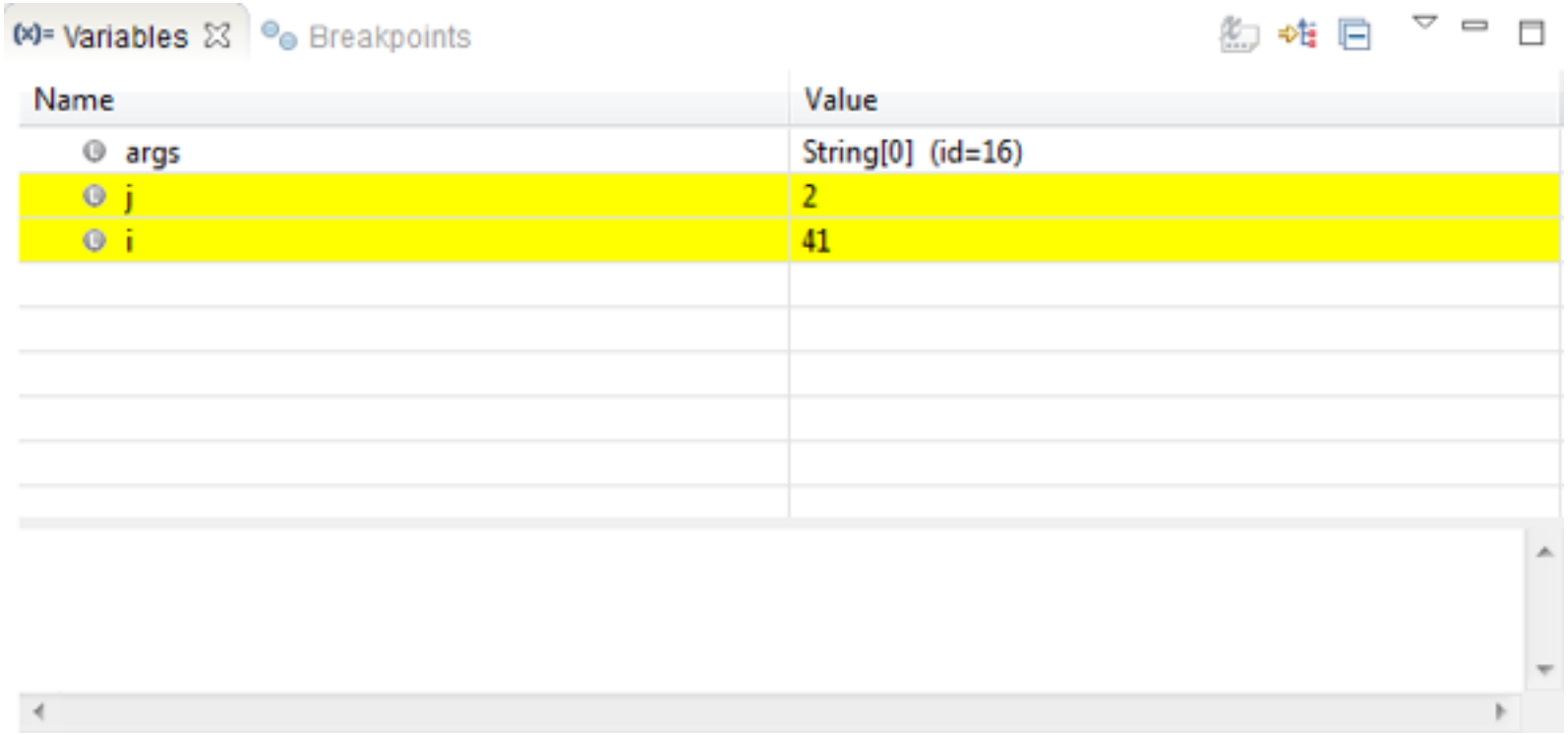
Debugger: Inspecting Variables

- In case we want to inspect multiple variables, we can use the "Variables" or "Expressions" debug views:



Debugger: Inspecting Variables

- In case we want to inspect multiple variables, we can use the "Variables" or "Expressions" debug views:



Debugger: Navigating the code

- Once you have stopped at a breakpoint, you can:
  - Press F5 to step into a function.
  - Press F7 to step out of a function.
  - Press F6 to move to the next line.
  - Press F8 to continue executing the code (until the next breakpoint).

```
for (int i = 0; i < 100; i++) {
    if (i % 43 == 0) {
        // Do something once in every 43
        // cycles.
        processNumber(i);

    }
}
```

Debugger: Navigating the code

- Once you have stopped at a breakpoint, you can:
  - Press F5 to step into a function.
  - Press F7 to step out of a function.
  - Press F6 to move to the next line.
  - Press F8 to continue executing the code (until the next breakpoint).

```
for (int i = 0; i < 100; i++) {
    if (i % 43 == 0) {
        // Do something once in every 43
        // cycles.
        processNumber(i);

    }
}
```

F5 will step into "processNumber" method code.

Debugger: Conditional Breakpoints

```
public static void main(String[] args) {

    for (int j = 0; j < 100; j++) {
        for (int i = 0; i < 100; i++) {
            processNumber(i,j);
        }
    }

}
```

What if something goes wrong only when i == 23 and j == 23.
What can we do to debug our code efficiently ?

Debugger: Conditional Breakpoints

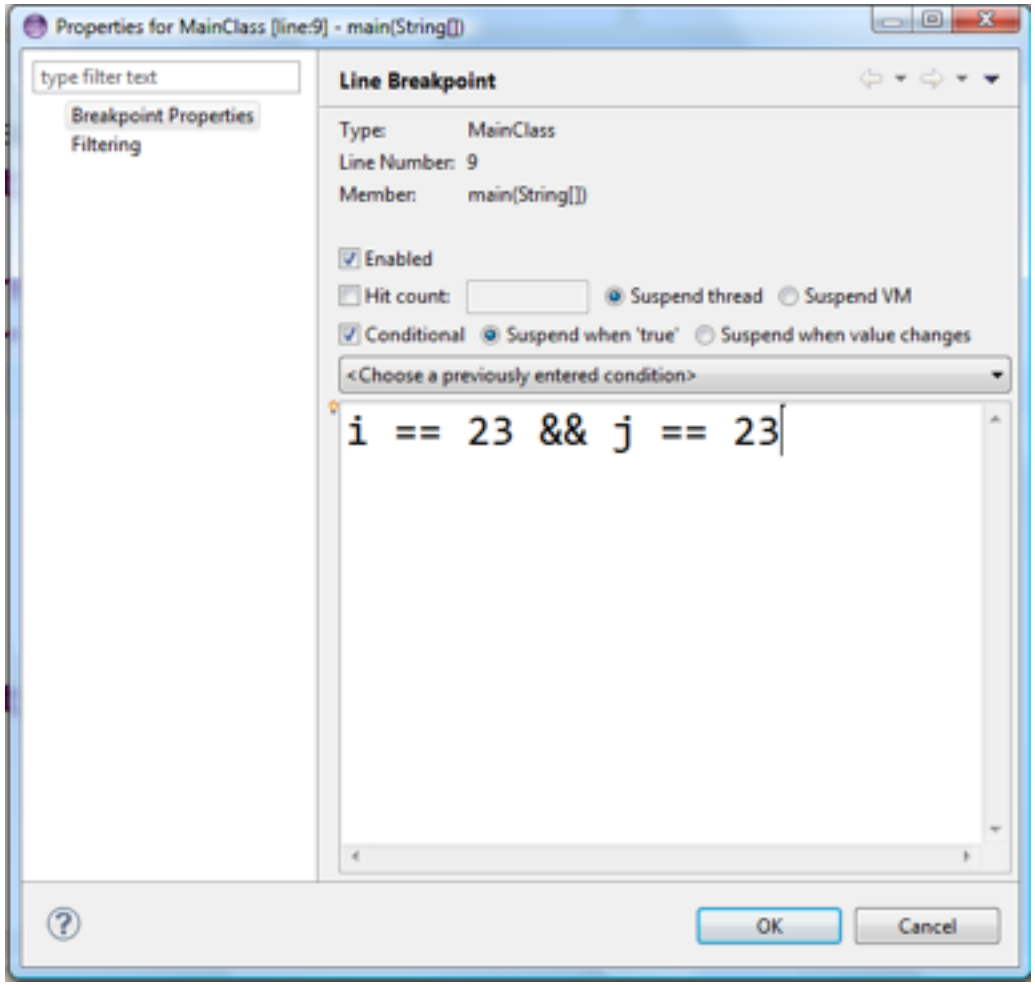
We can add temporary condition to our code and add a breakpoint when it holds. Highly not recommended.

```
for (int j = 0; j < 100; j++) {
    for (int i = 0; i < 100; i++) {
        // Debug ! Remember to remove
        if (i == 23 && j == 23) {
            int stam = 0;
        }
        // Debug

        processNumber(i,j);
    }
}
```

Breakpoint

Debugger: Conditional Breakpoints

We can use a **Conditional Breakpoint.**



Organizing your classes:Packages

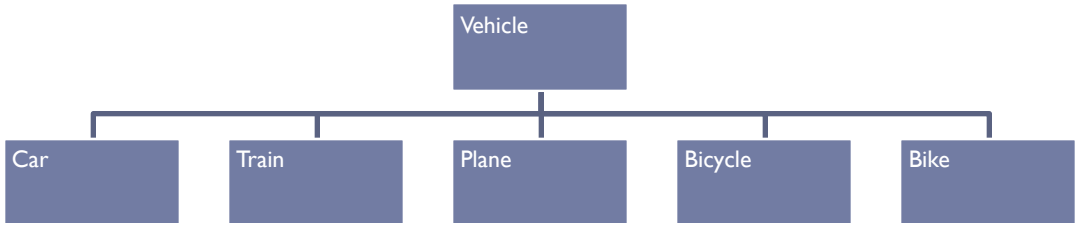
- Packages allows you to group together related classes.
- It can be thought of as "folders" of classes.
- Packages make it easier to:
  - Be aware of all the classes relating to the same purpose.
  - Solve conflicting names.
- Additionally, packages plays a role in access permission between related classes.

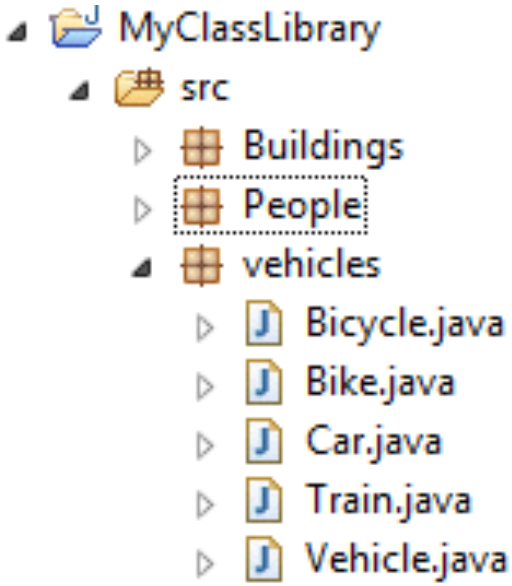
Creating packages:Packages

- The first line of every source file (*.java) should start with a package statement (one statement in each file).
  - - package folder.folder…

- Three ways of using code from other packages:
  - Importing a Package Member – import vehicle.Car
  - Importing an Entire Package – import vehicle.*

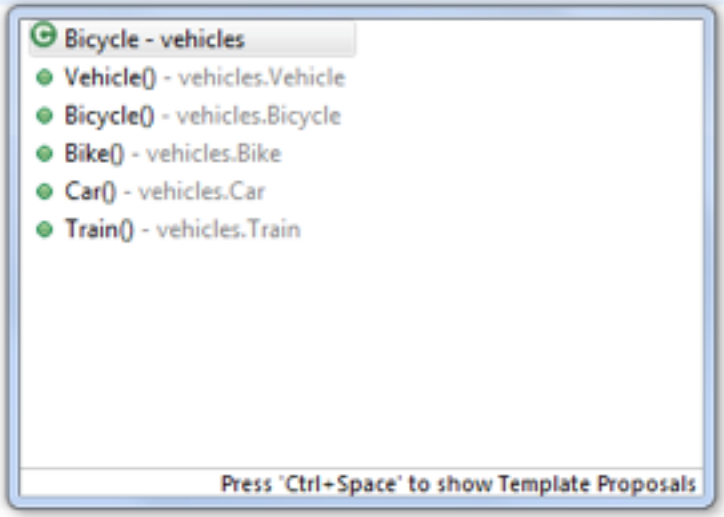
Related classes stored together:Packages





Related classes stored together:Packages

```
public static void main(String[] args) {
    Vehicle v = new vehicles.

}
```



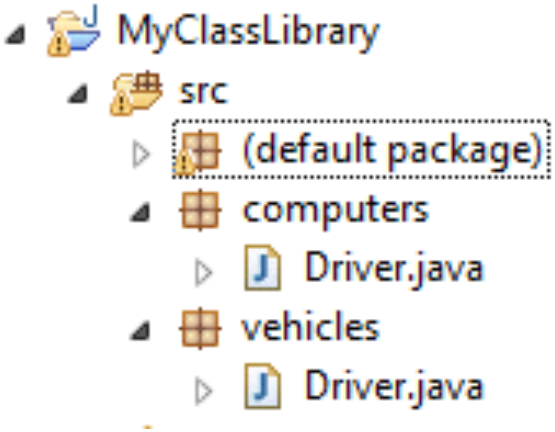
Now the user can easily know which Vehicle implementations are available.

Solving name conflicts:Packages

- Let's say you've written two classes in your code with the same name:
  - class Driver: Represents a vehicle driver.
  - class Driver: Represents a computer driver.

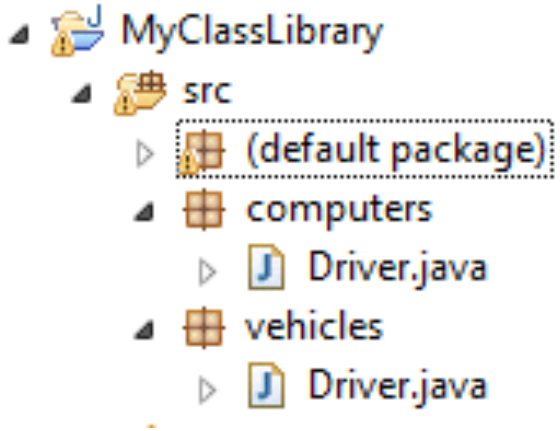
- We'll get a compilation error if we'd try to compile both classes in the "same package".

- However, this will work ->



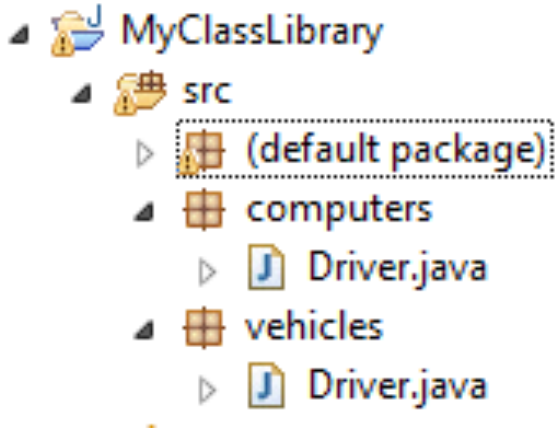
Solving name conflicts:Packages

```
import vehicles.*;

import computers.*;


public class MainClass {


    public static void main(String[] args) {

        Driver carDriver = new Driver();

        Driver compDriver = new Driver();

    }

}
```



Solving name conflicts:Packages

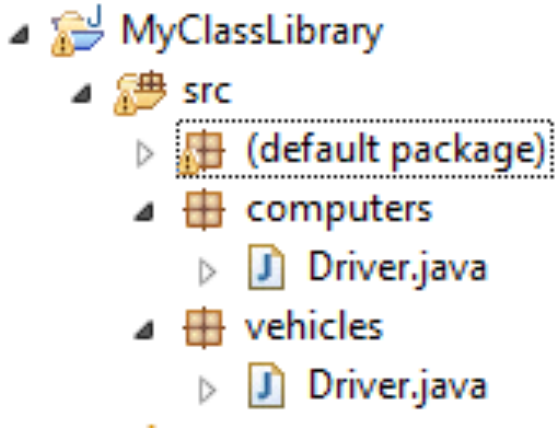
```
import vehicles.*;

import computers.*;


public class MainClass {


    public sta                    String[] args) {

        Driver                    new Driver();

        Dri                       new Driver();

    }

}
```



To distinguish between the two we must use the fully qualified name.

Solving name conflicts:Packages

```
import vehicles.*;

import computers.*;


public class MainClass {


      public static void main(String[] args) {

            vehicles.Driver carDriver = new vehicles.Driver();

            computers.Driver compDriver = new computers.Driver();

            // ..

      }

}
```



Permissions:Packages

- You were not told the whole truth about the *protected* access modifier.

- *Protected* fields can be accessed by:
  - Subclasses.

Permissions:Packages

- You were not told the whole truth about the *protected* access modifier.

- *Protected* fields can be accessed by:
  - Subclasses.
  - All the classes that share the same package.

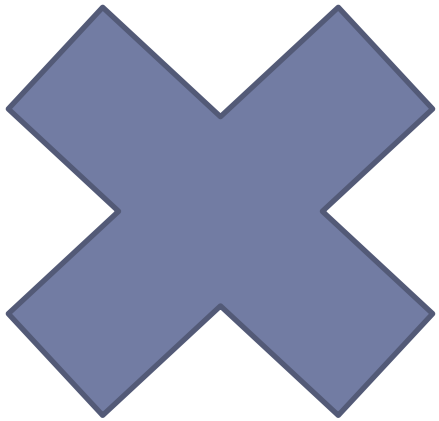
Permissions:Packages

```java
package humans;

public class Doctor {
    public boolean seePatient(Patient patient) {
        if (patient.heartRate > 190) {
            return false;
        }

        return true;
    }
}
```

```java
package humans;

public class Patient {
    protected int heartRate;
}
```

Permissions:Packages

```java
package medical.crew;


public class Doctor {
    public boolean seePatient(Patient patient) {
        if (patient.heartRate > 190) {
            return false;
        }

        return true;
    }
}
```



```java
package humans;

public class Patient {
    protected int heartRate;
}
```

Permissions:Packages

- What if there is no modifier?

```
public class Car {
    public int numOfWheel;
    int numOfWindows;

}
```

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default(=package)	***Y***	***Y***	***N***	***N***
private	Y	N	N	N

EX 5

Implementing AVL-Tree