Lecture - 08

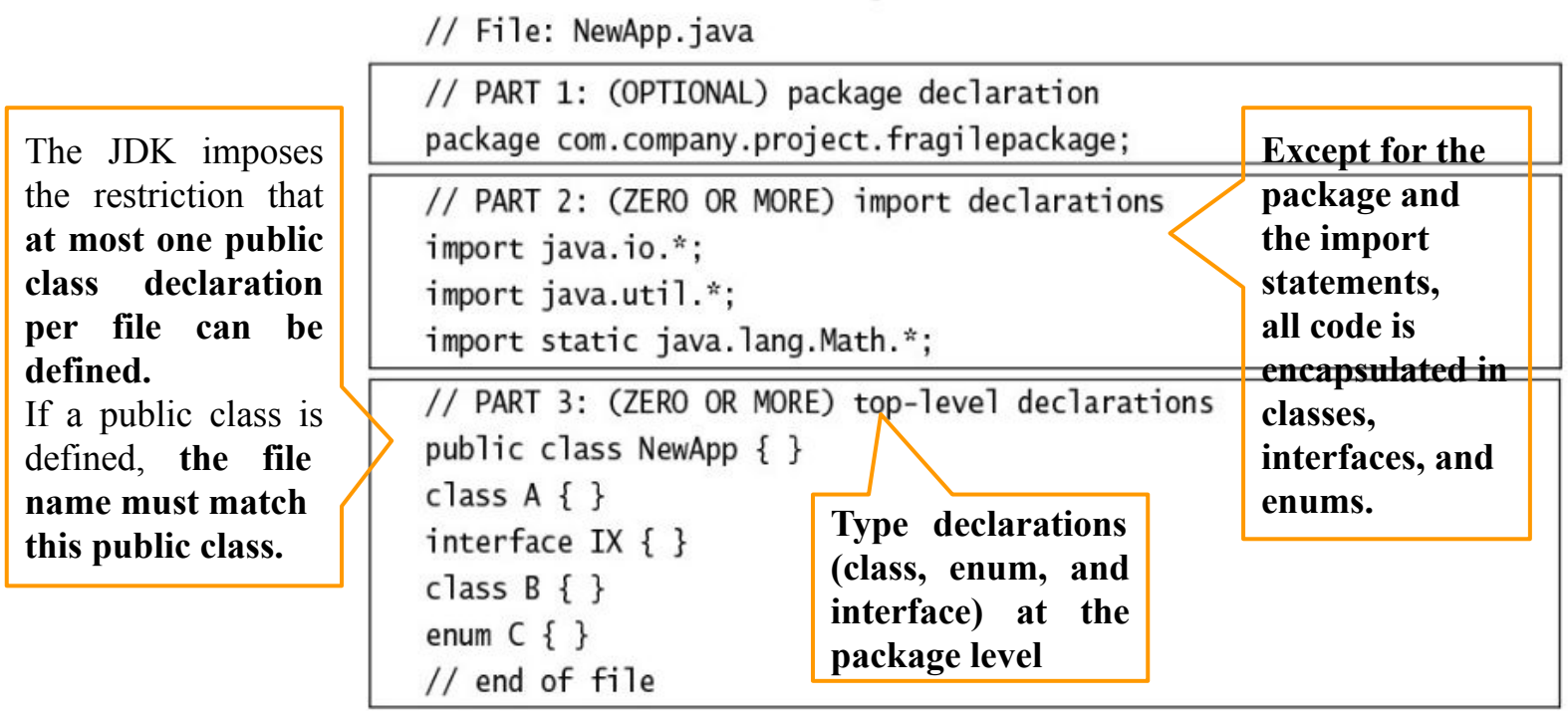
Access Control

Arpit Rana

15th Feb 2022

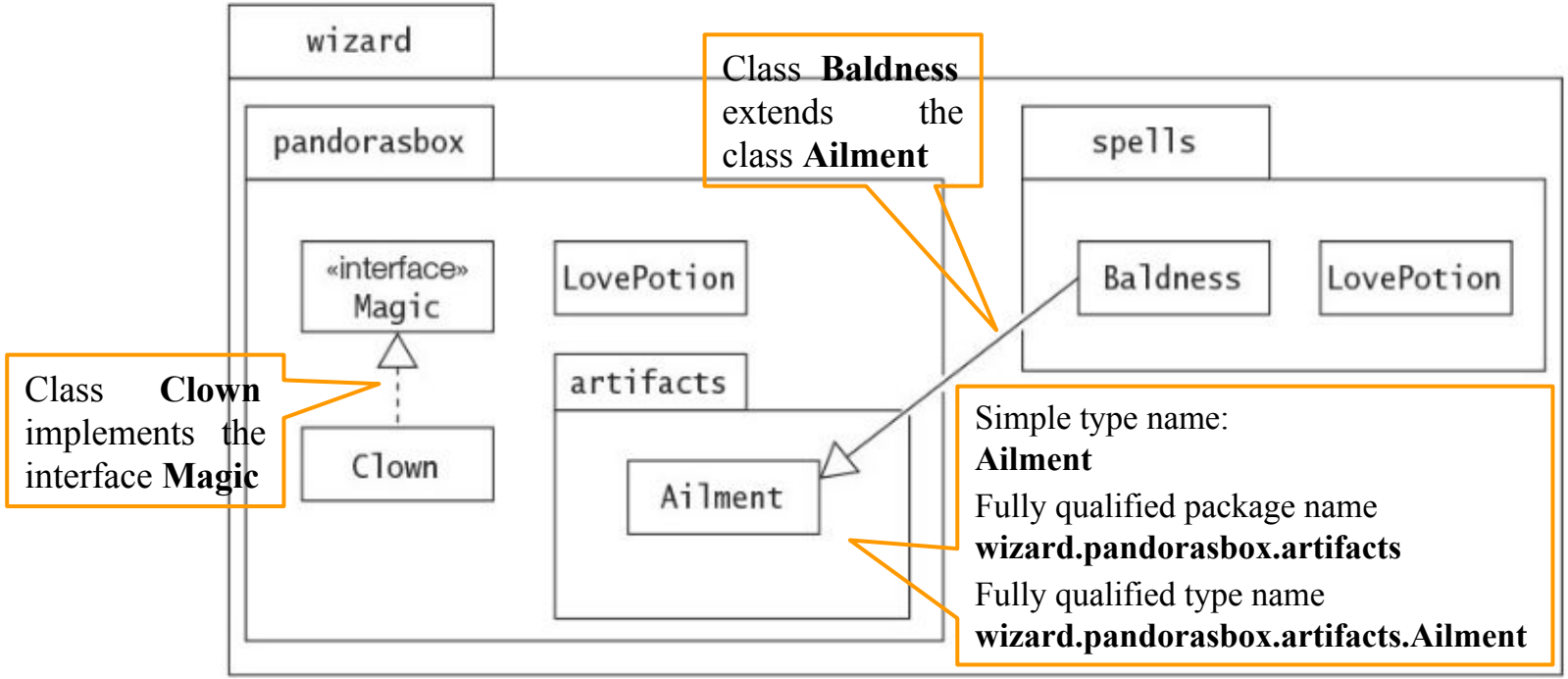
Java Source File Structure

A Java source file can have the following elements that, if present, must be specified in the following order:



Packages

In Java, a package is an encapsulation mechanism that can be used to group related classes, interfaces, enums, and subpackages.



Defining Packages

The package statement has the following syntax:

```
package fully_qualified_package_name;
```

- At most ***one package declaration*** can appear in a source file and it must be ***the first statement*** in the source file.
- The package name is ***saved in the Java bytecode*** for the types contained in the package.
- Java naming conventions recommend writing ***package names in lowercase letters***.
- All the classes and interfaces in a source file will be placed in the same package and several source files can be used to specify the contents of a package.
- By default, the Java bytecode for the declarations in the compilation unit belongs to an unnamed package (also called the *default* package)

```
// File name: Clown.java
package wizard.pandorasbox;                    // Package declaration

import wizard.pandorasbox.artifacts.Ailment; // Importing specific class

public class Clown implements Magic { /* … */ }

interface Magic { /* … */ }
```

```
// File name: LovePotion.java
package wizard.pandorasbox;                    // Package declaration

public class LovePotion { /* … */
```

```
// File name: Ailment.java
package wizard.pandorasbox.artifacts;          // Package declaration

public class Ailment { /* … */ }
```

```
// File name: Baldness.java
package wizard.spells;                         // Package declaration

import wizard.pandorasbox.*;                   // (1) Type-import-on-demand
import wizard.pandorasbox.artifacts.*;         // (2) Import from subpackage

public class Baldness extends Ailment {        // Simple name for Ailment
  wizard.pandorasbox.LovePotion tlcOne;        // (3) Fully qualified class
name
  LovePotion tlcTwo;                           // Class in same package
  // …
}

class LovePotion { /* … */ }
```

Using Packages

The import facility in Java makes it easier to use the contents of packages.

Importing Reference Types

- By the *fully qualified name* of the type.
- By the *import statement*
  - *Single type import:* `import fully_qualified_type_name;`
  - *Type import on demand:* `import fully_qualified_package_name.*;`
- The import declarations must be the first statement after any package declaration in a source file.
- An import declaration does not recursively import subpackages.
- All compilation units implicitly import the `java.lang` package. e.g. we can refer to the class String without using its fully qualified name `java.lang.String` all the time.

Using Packages

The import facility in Java makes it easier to use the contents of packages.

Importing Static Members of Reference Types

- Java allows import of static members of reference types from packages, often called ***static import***.
  - Single static import: `import static fully_qualified_type_name.static_member_name;`
  - Static import on demand: `import static fully_qualified_type_name.*;`
- Compiling code into packages
- Running code from packages

Scope Rules

In two areas access is governed by specific scope rules:

- ***Class scope for members:*** how member declarations are accessed within the class.
- ***Block scope for local variables:*** how local variable declarations are accessed within a block

Class Scope for Members

```
class SuperName {
  int instanceVarInSuper;
  static int staticVarInSuper;

  void instanceMethodInSuper()       { /* … */ }
  static void staticMethodInSuper() { /* … */ }
  // …
}

class ClassName extends SuperName {
  int instanceVar;
  static int staticVar;

  void instanceMethod()       { /* … */ }
  static void staticMethod() { /* … */ }
  // …
}
```

Member declarations	Non-static code in the class ClassName can refer to the member as	Static code in the class ClassName can refer to the member as

Instance variables	`instanceVar`
	`this.instanceVar`
	`instanceVarInSuper`
	`this.instanceVarInSuper`
	`super.instanceVarInSuper`
		Not possible

Instance methods	`instanceMethod()`
	`this.instanceMethod()`
	`instanceMethodInSuper()`
	`this.instanceMethodInSuper()`
	`super.instanceMethodInSuper()`
		Not possible

Static variables	`staticVar`
	`this.staticVar`
	`ClassName.staticVar`
	`staticVarInSuper`
	`this.staticVarInSuper`
	`super.staticVarInSuper`
	`ClassName.staticVarInSuper`
	`SuperName.staticVarInSuper`
		`staticVar`
		
		`ClassName.staticVar`
		`staticVarInSuper`
		
		`ClassName.staticVarInSuper`
		`SuperName.staticVarInSuper`

Static methods	`staticMethod()`
	`this.staticMethod()`
	`ClassName.staticMethod()`
	`staticMethodInSuper()`
	`this.staticMethodInSuper()`
	`super.staticMethodInSuper()`
	`ClassName.staticMethodInSuper()`
	`SuperName.staticMethodInSuper()`
		`staticMethod()`
		
		`ClassName.staticMethod()`
		`staticMethodInSuper()`
		
		`ClassName.staticMethodInSuper()`
		`SuperName.staticMethodInSuper()`

Class Scope for Members

The following factors can influence the scope of a member declaration:

- Shadowing of a field declaration, either by local variables or by declarations in the subclass.
- Overriding an instance method from a superclass.
- Hiding a static method declared in a superclass.

Block Scope for Local Variables

Blocks can be nested, and scope rules apply to local variable declarations in such blocks.

- A variable declared in a block is in scope inside the block in which it is declared, but it is not accessible outside of this block.
- It is not possible to re-declare a variable if a local variable of the same name is already declared in the current scope.
- Local variables of a method include the formal parameters of the method and variables that are declared in the method body.
- The local variables in a method are created each time the method is invoked, and are therefore distinct from local variables in other invocations.

Block Scope for Local Variables

```
public static void main(String args[]) {              // Block 1
//  String args = "";     // (1) Cannot redeclare parameters.
    char digit = 'z';
    for (int index = 0; index < 10; ++index) {        // Block 2
      switch(digit) {                                 // Block 3
          case 'a':
              int i;    // (2)
          default:
          //  int i;    // (3) Already declared in the same block
      } // end switch
      if (true) {                                     // Block 4
          int i;        // (4) OK
      //  int digit;    // (5) Already declared in enclosing Block 1
      //  int index;    // (6) Already declared in enclosing Block 2
      } // end if
    } // end for
    int index;          // (7) OK
} // end main
```

Accessibility Modifiers for Top-Level Type Declarations

Top-level types means classes, enums, and interfaces

Modifiers	Top-level types
No modifier	Accessible in its own package (*package accessibility*)
`public`	Accessible anywhere

Non-Accessibility Modifiers for Classes

The non-accessibility modifiers ***"abstract"*** and ***"final"*** can be applied to top-level classes.

A class can either be ***"abstract"*** or ***"final", but*** not both.

Non-Accessibility Modifiers for Classes

abstract Classes

- A non-final class can be declared as abstract
- An abstract class cannot be instantiated
- A class with an abstract method must be declared as abstract
- Subclasses of the abstract class have to provide implementation of any inherited abstract methods before instances can be created.

```
abstract class Light {
  // Fields:
  int     noOfWatts;       // Wattage
  boolean indicator;       // On or off
  String  location;        // Placement

  // Instance methods:
  public void switchOn()  { indicator = true; }
  public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }

  // Abstract instance method
  public abstract double kwhPrice();                  // (1) No method body
}
//_____________________________________________________________________________
class TubeLight extends Light {
  // Field
  int tubeLength;

  // Implementation of inherited abstract method.
  @Override public double kwhPrice() { return 2.75; } // (2)
}
//_____________________________________________________________________________
public class Factory {
  public static void main(String[] args) {
    TubeLight cellarLight = new TubeLight();          // (3) OK
    Light nightLight;                                 // (4) OK
//  Light tableLight = new Light();                   // (5) Compile-time
error
    nightLight = new TubeLight();                     // (6) OK
    System.out.println("KWH price: $" + nightLight.kwhPrice());
  }
}
```

Non-Accessibility Modifiers for Classes

final Classes

- A non-abstract (a.k.a. concrete) class can be declared as final
- A final class cannot be extended
- A class with a final method need not to be declared as final
- The Java SE platform API includes many final classes—for example, the `java.lang.String` class and the wrapper classes for primitive values.

A final class and an interface represent two extremes when it comes to providing an implementation. An abstract class represents a compromise between these two extremes.

Member Accessibility Modifiers

By specifying member accessibility modifiers, a class can control which information is accessible to clients (that is, other classes).

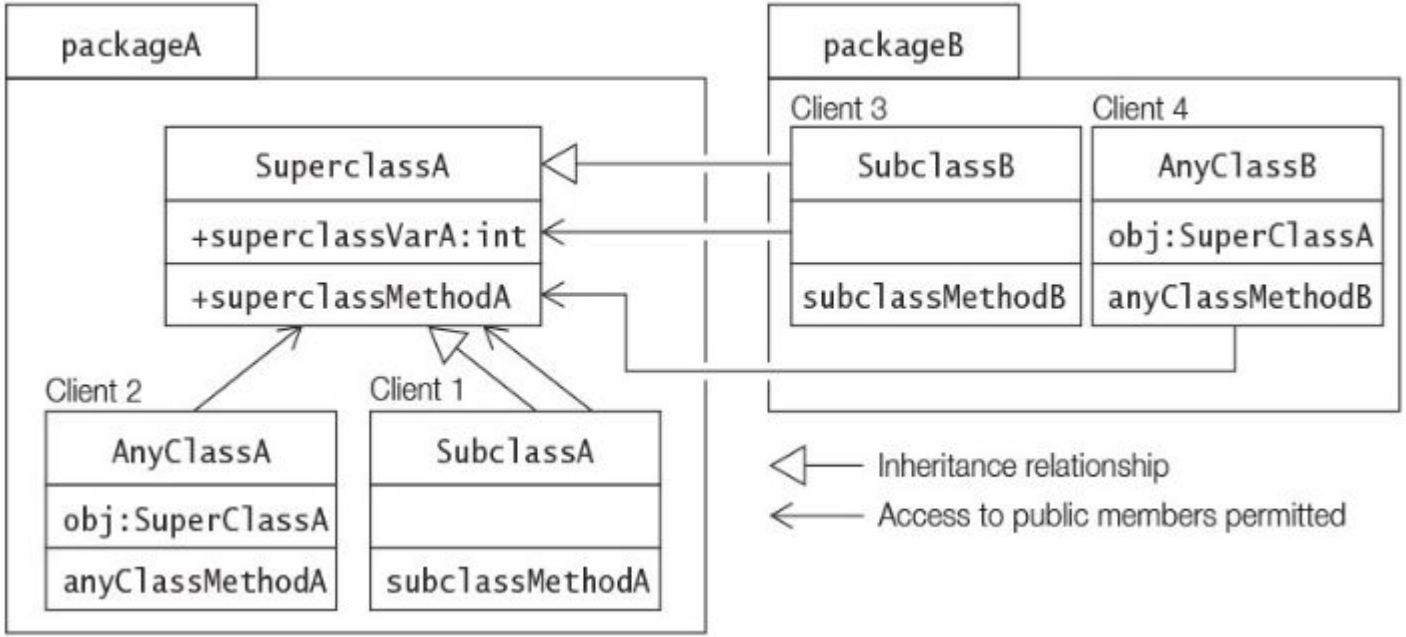
The accessibility of members can be one of the following:

- `public`
- `protected`
- `Default accessibility` (also known as package accessibility), meaning that no accessibility modifier is specified
- `private`

Member Accessibility Modifiers

`public` Members

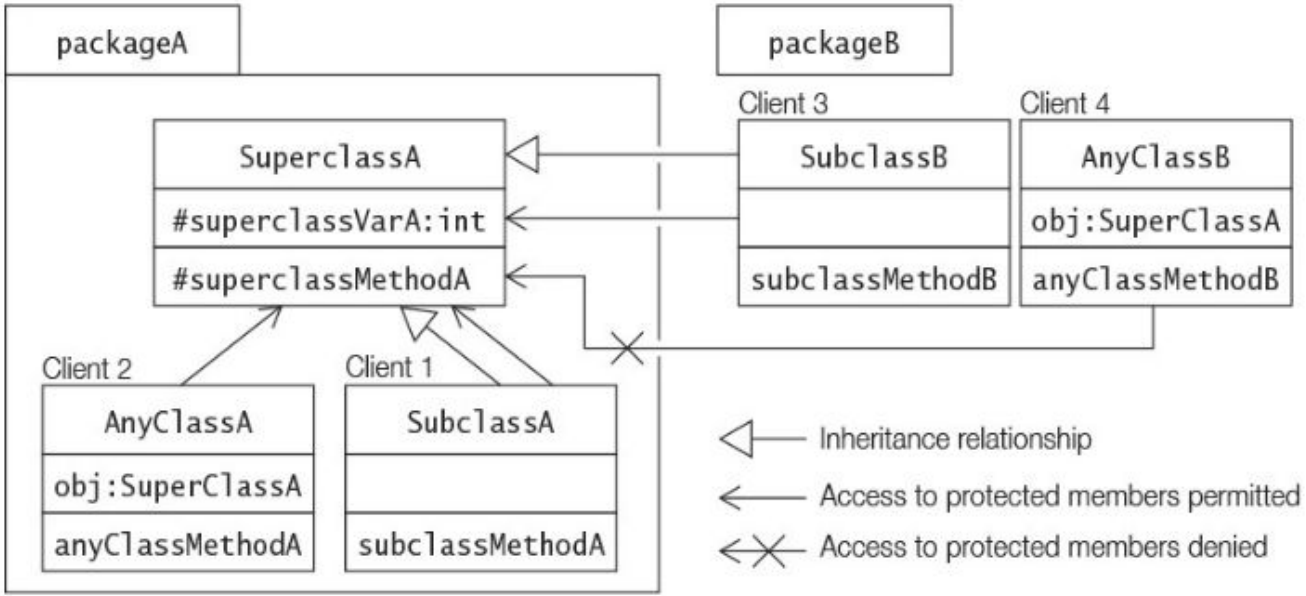
- Accessible anywhere



Member Accessibility Modifiers

`protected` Members

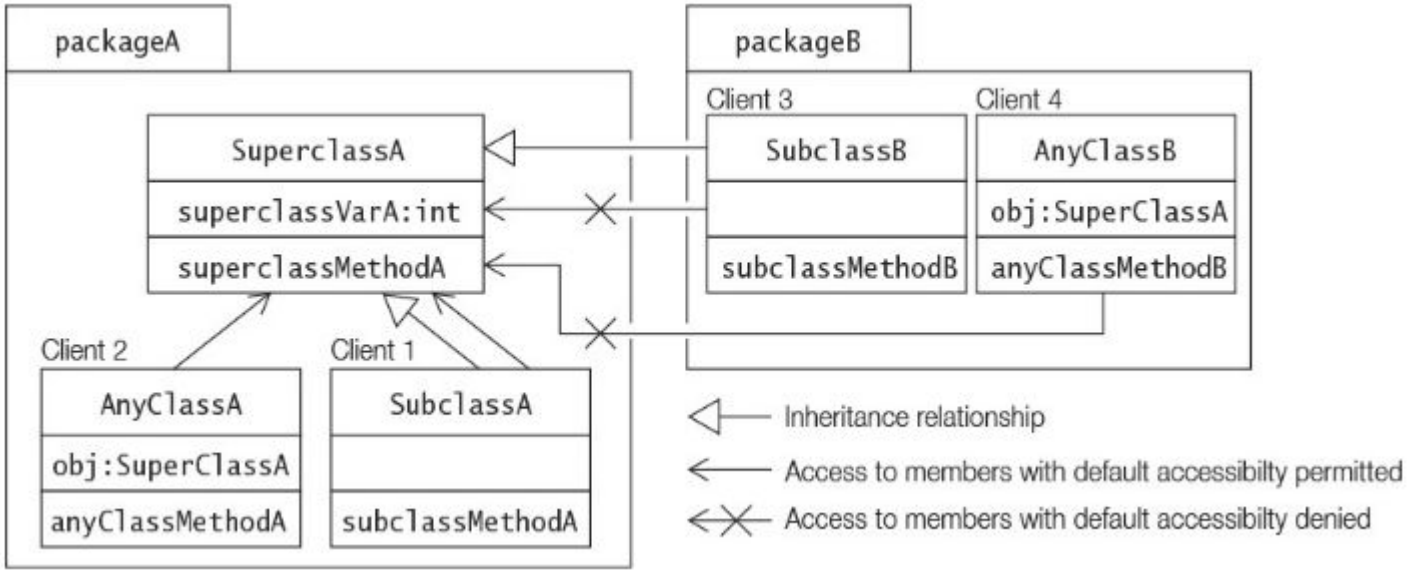
- Accessible by any class in the same package as its class and accessible only by subclass of its class in other packages



Member Accessibility Modifiers

Default accessibility for Members

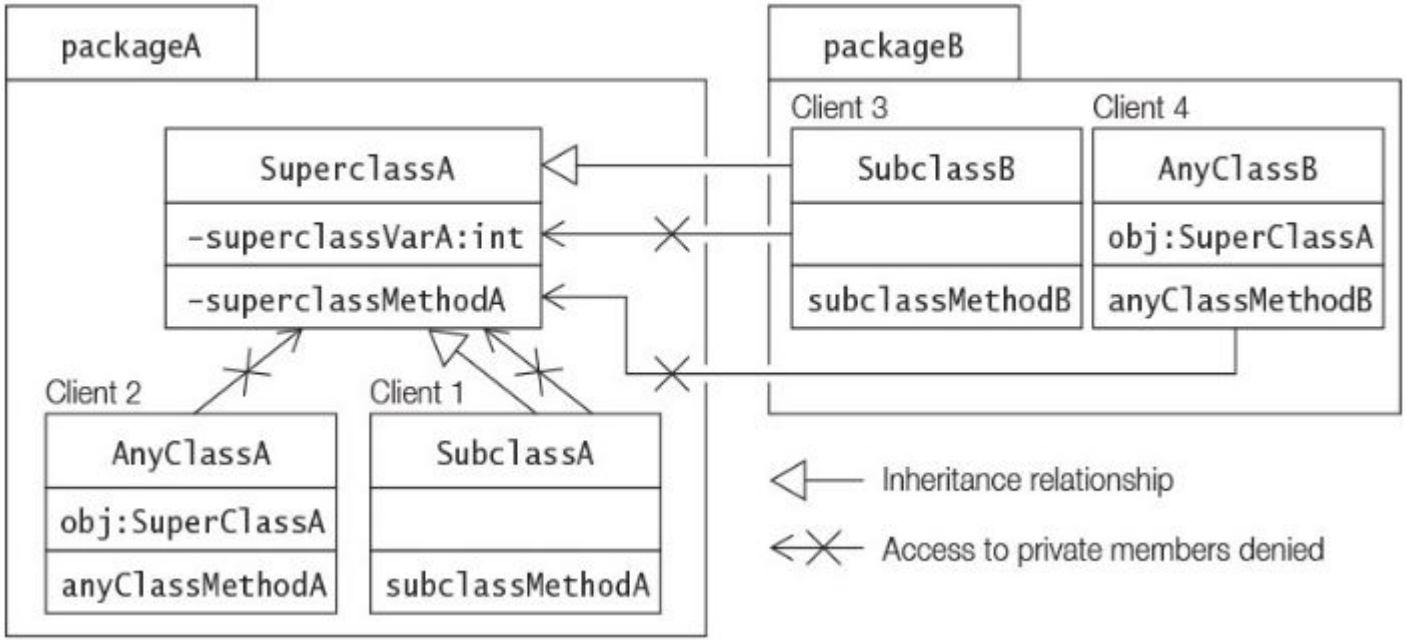
- Only accessible by classes, including subclasses, in the same package as its class



Member Accessibility Modifiers

`private` Members

- Only accessible in its own class and not anywhere else



Non-Accessibility Modifiers for Members

Modifier	Fields	Methods
`static`	Defines a class variable.	Defines a class method.
`final`	Defines a constant.	The method cannot be overridden.
`abstract`	Not applicable.	No method body is defined. Its class must also be designated as `abstract`.
`synchronized`	Not applicable.	Only one thread at a time can execute the method.
`native`	Not applicable.	Declares that the method is implemented in another language.
`transient`	The value in the field will not be included when the object is serialized.	Not applicable.
`volatile`	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

Next lecture -

Exception Types and Handling