Access Control

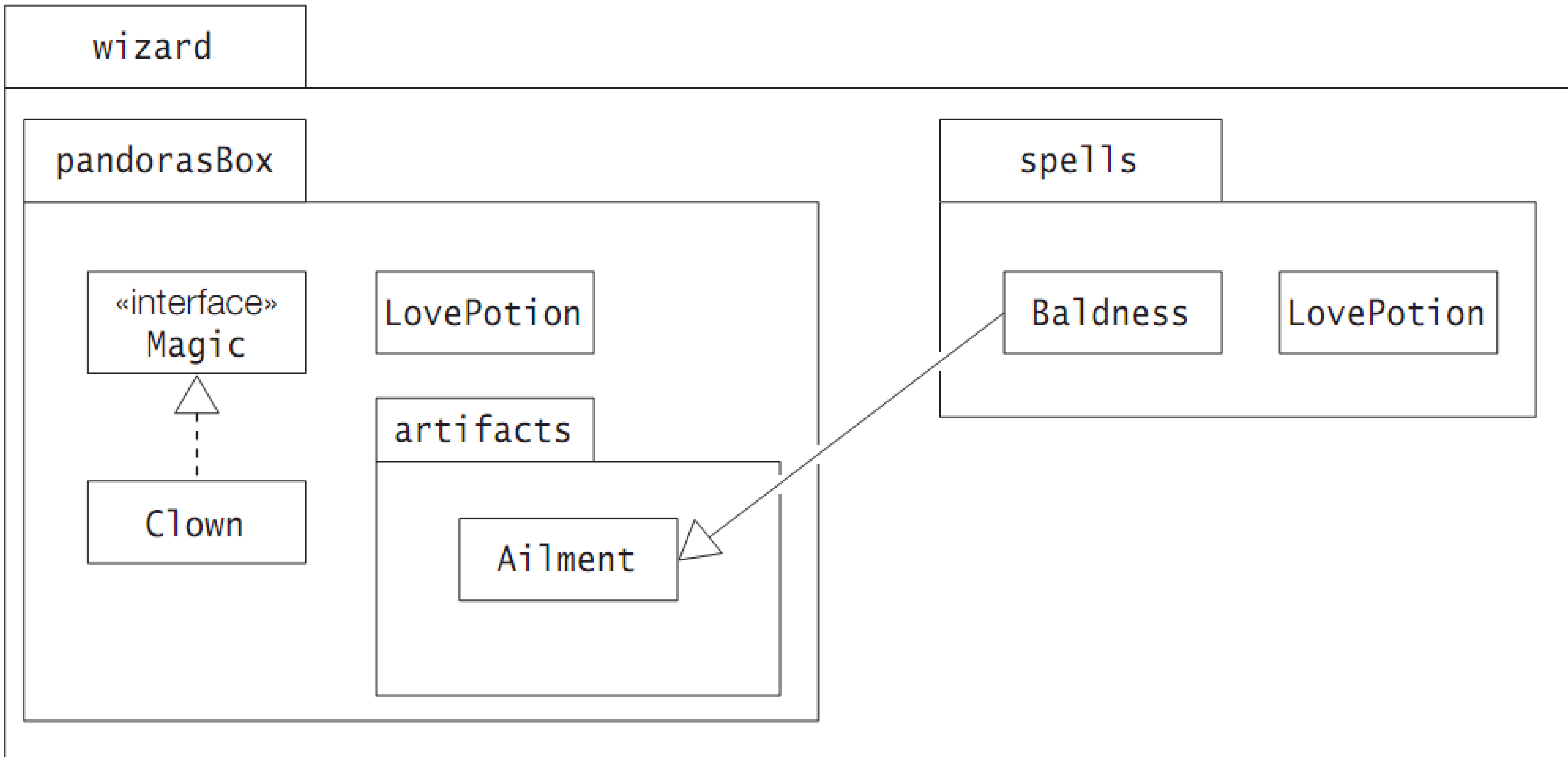
A Java source file can have the following elements that, if present, must be specified in the following order:

1. An optional package declaration to specify a package name.
2. Zero or more import declarations. Since import declarations introduce type or static member names in the source code, they must be placed before any type declarations.
3. Any number of top-level type declarations.

Class, **enum**, and **interface** declarations are collectively known as type declarations. Since these declarations belong to the same package, they are said to be defined at the top level, which is the package level.

```java
// Filename: NewApp.java

// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilePackage;

// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;
import static java.lang.Math.*;

// PART 3: (ZERO OR MORE) top-level class and interface declarations
public class NewApp { }

class A { }

interface IX { }

class B { }

interface IY { }

enum C { FIRST, SECOND, THIRD }

// end of file
```

Packages



Defining Packages

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 byte code for the types contained in the package.

Defining Packages

- If a package declaration is omitted in a compilation unit, the Java byte code for the declarations in the compilation unit will belong to an unnamed package (also called the default package), which is typically synonymous with the current working directory on the host system.

Defining Packages

```
//File: Clown.java
package wizard.pandorasBox;    // (1) Package declaration
import wizard.pandorasBox.artifacts.Ailment; // (2) Importing class
public class Clown implements Magic { /* ... */ }
interface Magic { /* ... */ }
//File: LovePotion.java
package wizard.pandorasBox;    // (1) Package declaration
public class LovePotion { /* ... */ }
//File: Ailment.java
package wizard.pandorasBox.artifacts; // (1) Package declaration
public class Ailment { /* ... */ }
//File: Baldness.java
package wizard.spells;      // (1)Package declaration
import wizard.pandorasBox.*;   // (2) Type-import-on-demand
import wizard.pandorasBox.artifacts.*;  // (3) Import from subpackage
public class Baldness extends Ailment { // (4) Abbreviated name for Ailment
  wizard.pandorasBox.LovePotion tlcOne; // (5) Fully qualified name
  LovePotion tlcTwo;   // (6) Class in same package
  // ...
}
class LovePotion { /* ... */ }
```

Using Packages

Single-type-import:

```
import <fully qualified type name>;
```

Type-import-on-demand:

```
import <fully qualified package name>.*;
```

Using Packages

- An import declaration does not recursively import subpackages.
- The declaration also does not result in inclusion of the source code of the types.
- The declaration only imports type names.
- All compilation units implicitly import the `java.lang` package

Importing Static Members of Reference Types

- Single-static-import: imports a specific static member from the designated type

```
import static <fully qualified type name>.<static member name>;
```

- Static-import-on-demand: imports all static members in the designated type

```
import static <fully qualified type name>.*;
```

Example:

```
import static java.lang.Math.*;
double hypotenuse = hypot(x, y);
```

- Both forms require the use of the keyword static.
- In both cases, the fully qualified name of the reference type we are importing from is required.

import static Example

```
import static java.lang.Math.PI;   // (1) Static field
import static java.lang.Math.sqrt; // (2) Static method
// Only specified static members are imported.
public class CalculateI {
  public static void main(String[] args) {
    double x = 3.0, y = 4.0;
    double squareroot = sqrt(y); // Simple name of static method
    double hypotenuse = Math.hypot(x, y);  // (3) Requires type name.
    double area = PI * y * y;        // Simple name of static field
    System.out.printf("Square root: %.2f, hypotenuse: %.2f, area: 
%.2f%n", squareroot, hypotenuse, area);
  }
}
```

Avoiding the Interface Constant Antipattern

```java
package mypkg;
public interface IMachineState {
  // Fields are public, static and final.
  int BUSY = 1;
  int IDLE = 0;
  int BLOCKED = -1;
}
import static mypkg.IMachineState.*;
// (1) Static import interface constants
public class MyFactory {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED };
    for (int s : states)
      System.out.print(s + " ");
  }
}
```

Importing Enum Constants

```
package mypkg;
public enum State { BUSY, IDLE, BLOCKED }
----------------------------------------------------------------------------------
import mypkg.State; // (1) Single type import
import static mypkg.State.*;          // (2) Static import on demand
import static java.lang.System.out;   // (3) Single static import
public class Factory {
  public static void main(String[] args) {
    State[] states = {
      IDLE, BUSY, IDLE, BLOCKED  // (4) Using static import implied by (2).
    };
    for (State s : states)       // (5) Using type import implied by (1).
      out.print(s + " ");        // (6) Using static import implied by (3).
  }
}
```

Shadowing by Importing

```
import static java.lang.System.out;       // (1) Static import
import java.io.FileNotFoundException;
import java.io.PrintWriter;               // (2) Single type import
public class ShadowingByImporting {
  public static void main(String[] args) throws FileNotFoundException {
    out.println("Calling println() in java.lang.System.out");
    PrintWriter pw = new PrintWriter("log.txt");
    writeInfo(pw);
    pw.flush();
    pw.close();
  }
  public static void writeInfo(PrintWriter out) {
    // Shadows java.lang.System.out
    out.println("Calling println() in the parameter out");
    System.out.println("Calling println() in java.lang.System.out");
    // Qualify
  }
}
```

Conflict in Importing Static Method with the Same Signature

```
package mypkg;
public class Auxiliary {
  public static int binarySearch(int[] a, int key) {// As in java.util.Arrays.
    // Implementation is omitted.
    return -1;
  }
}
-----------------------------------------------------------------------------
import static java.util.Collections.binarySearch;  //  2 overloaded methods
import static java.util.Arrays.binarySearch;       // + 18 overloaded methods
import static mypkg.Auxiliary.binarySearch; // (1) Causes signature conflict.
class MultipleStaticImport {
  public static void main(String[] args) {
    int index = binarySearch(new int[] {10, 50, 100}, 50); // (2) Not ok!
    System.out.println(index);
  }
//  public static int binarySearch(int[] a, int key) {     // (3)
//    return -1;
//  }
}
```

Running Code from Packages

If the current directory has the absolute pathname /pgj

work and we want to run Clown.class in the directory with the pathname ./wizard/pandorasBox, the fully qualified name of the Clown class must be specified in the java command

```
>java wizard.pandorasBox.Clown
```

This will load the class Clown from the byte code in the file with the pathname wizard/pandorasBox/Clown.class, and start the execution of its main() method.

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.

Accessing Members within a Class

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()`

Block Scope

```
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.
      } // 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.
      } //if

    } // for
    int index;             // (7) OK

} // main
```

Accessibility Modifiers for Top-Level Type Declarations

- The accessibility modifier public can be used to declare top-level types
- If the accessibility modifier is omitted, they are only accessible in their own package and not in any other packages or subpackages.

This is called *package* or *default* accessibility.

Other Modifiers for Classes

The modifiers **`abstract`** and **`final`** can be applied to top-level and nested classes.

`abstract` Classes

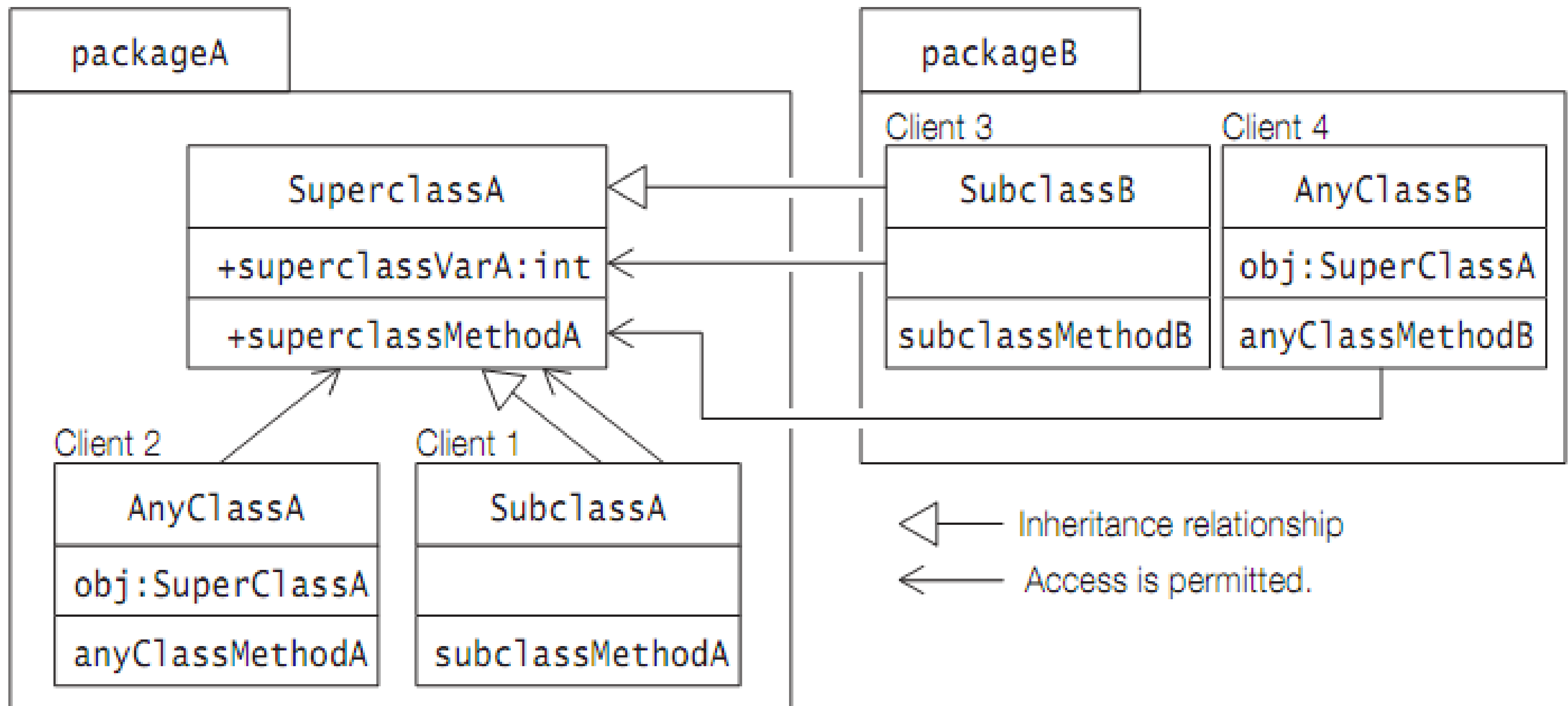
`final` Classes

`abstract` and `final`

Modifiers	Classes	Interfaces	Enum types
`abstract`	A non-`final` class can be declared `abstract`. A class with an `abstract` method must be declared `abstract`. An `abstract` class cannot be instantiated.	Permitted, but redundant.	Not permitted.
`final`	A non-`abstract` class can be declared `final`. A class with a `final` method need not be declared `final`. A `final` class cannot be extended.	Not permitted.	Not permitted.

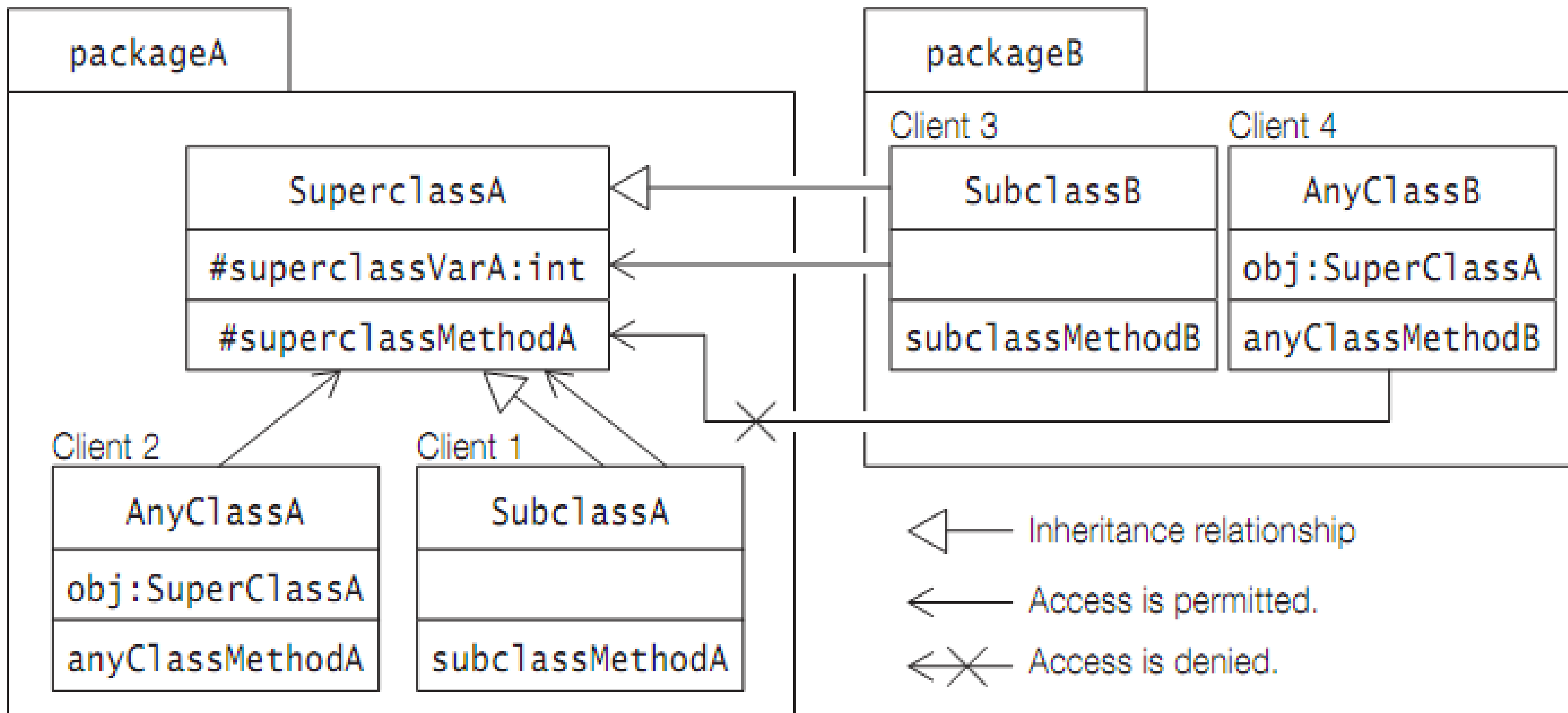
Member Accessibility Modifiers

public Members



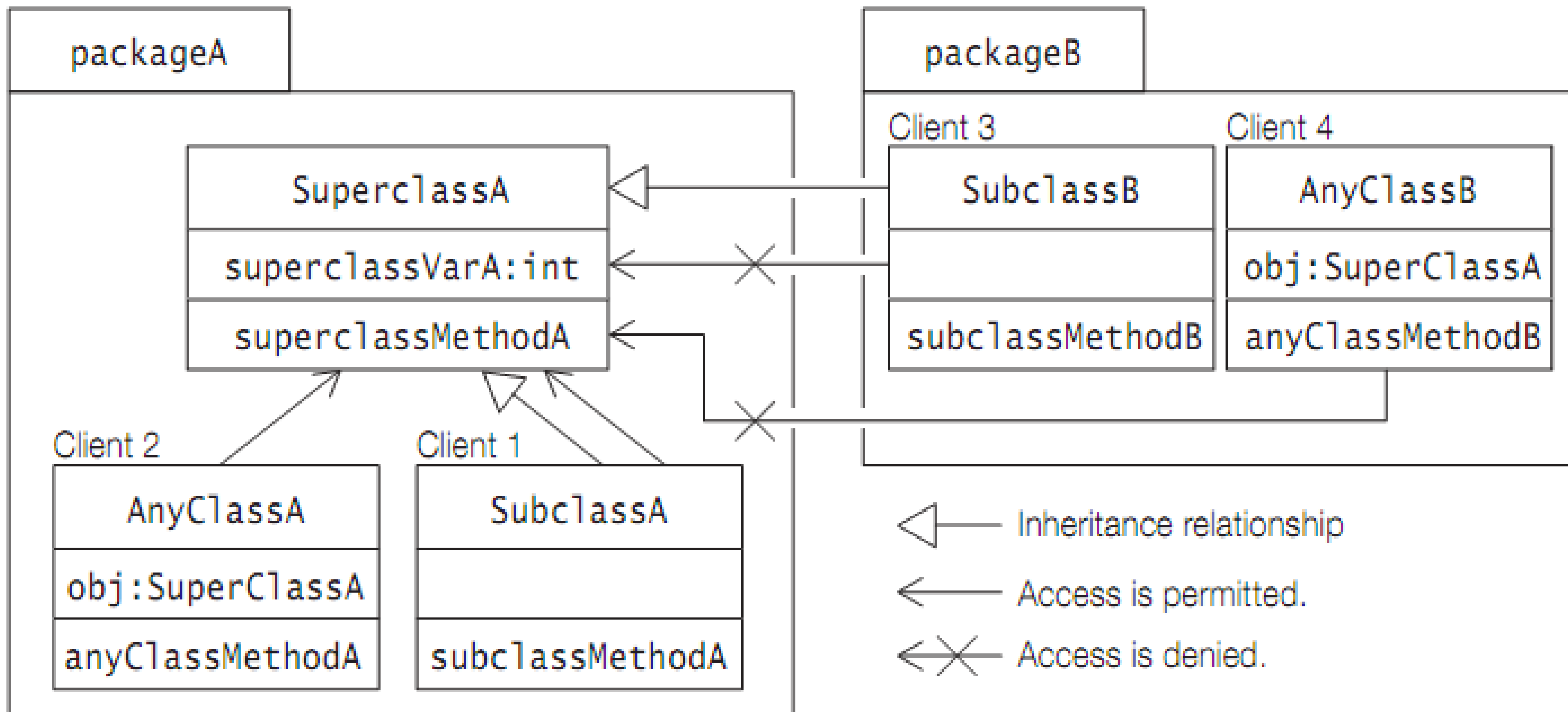
Member Accessibility Modifiers

protected Members



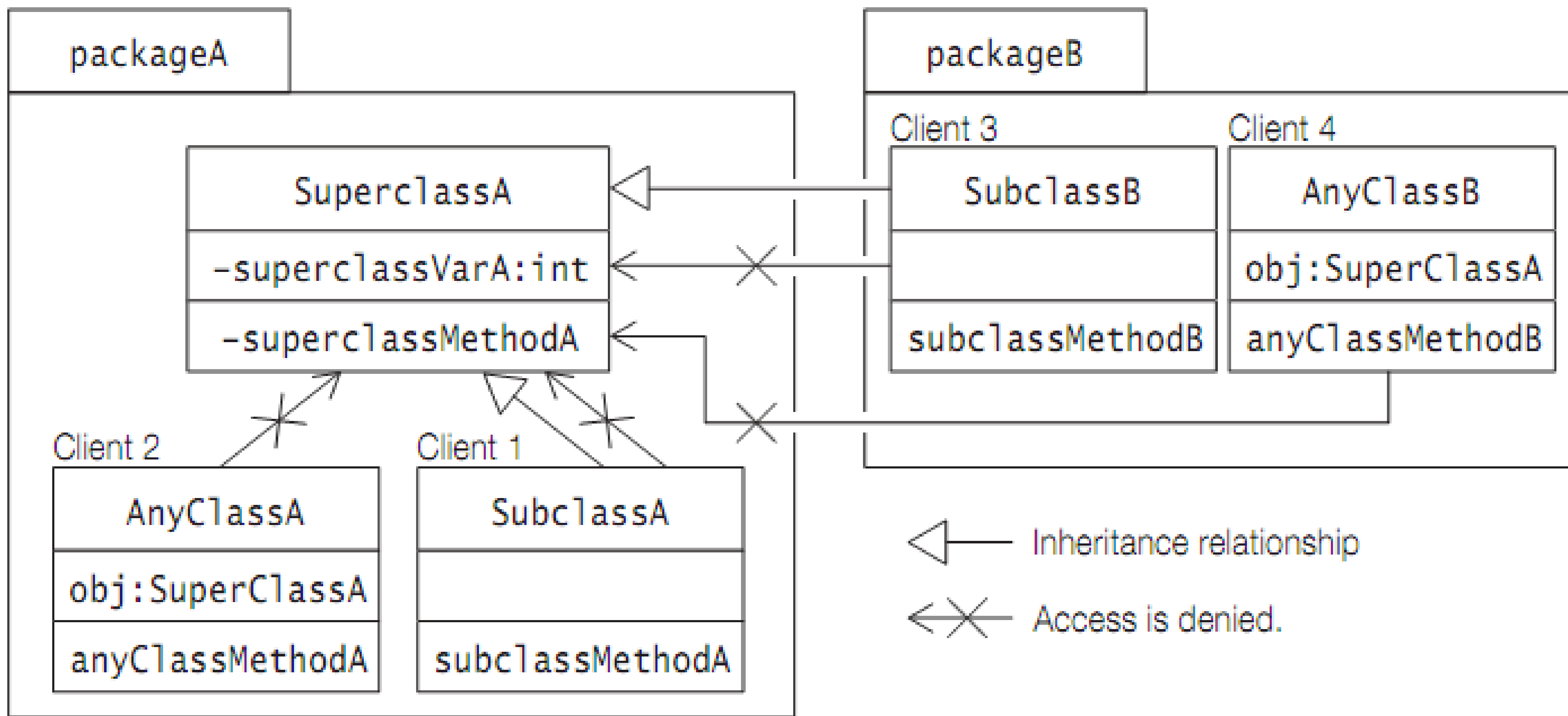
Member Accessibility Modifiers

Default Accessibility for Members



Member Accessibility Modifiers

private Members



Member Accessibility Modifiers

Summary

Modifiers	Members
`public`	Accessible everywhere.
`protected`	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
`private`	Only accessible in its own class and not anywhere else.

Other Modifiers for Members

- **`static`**
- **`final`**
- **`abstract`**
- **`synchronized`**
- **`native`**
- **`transient`**
- **`volatile`**

static Members

- **Static variables** (also called class variables) exist in the class they are defined in only.
- **Static methods** are also known as class methods. A static method in a class can directly access other static members in the class.

final Members

- A final variable of a primitive data type cannot change its value once it has been initialized.
- A final variable of a reference type cannot change its reference value once it has been initialized. This effectively means that a final reference will always refer to the same object.
- *However, the keyword final has no bearing on whether the state of the object denoted by the reference can be changed or not.*

final Members

- Final static variables are commonly used to define manifest constants (also called named constants)
- Note that a final variable need not be initialized in its declaration, but it must be initialized in the code once before it is used.
- These variables are also known as blank final variables.

abstract Methods

An abstract method has the following syntax:

```
abstract <accessibility modifier> <return
type> <method name> (<parameter list>)
<throws clause>;
```

- The keyword abstract is mandatory in the header of an abstract method declared in a class. Its class is then incomplete and must be explicitly declared abstract
- Only an instance method can be declared abstract. Since static methods cannot be overridden, declaring an abstract static method makes no sense.

synchronized Methods

- Methods can be declared synchronized if it is desirable that only one thread at a time can execute a method of the object. Their execution is then mutually exclusive among all threads.
- At any given time, at most one thread can be executing a synchronized method on an object. This discussion also applies to static synchronized methods of a class.

native Methods

- Native methods are methods whose implementation is not defined in Java but in another programming language, for example, C or C++.
- Such a method can be declared as a member in a Java class declaration. Since its implementation appears elsewhere, only the method header is specified in the class declaration.
- The keyword **`native`** is mandatory in the method header.

transient Fields

- Often it is desirable to save the state of an object. Such objects are said to be persistent. In Java, the state of an object can be stored using serialization
- Sometimes the value of a field in an object should not be saved, in which case, the field can be specified as **transient** in the class declaration.

volatile Fields

The **volatile** modifier can be used to inform the compiler that it should not attempt to perform optimizations on the field, which could cause unpredictable results when the field is accessed by multiple threads

Summary of Other Modifiers for Members

Modifiers	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 `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.