Object-based Programming

Reference:

Chapter 4 of A Programmer's Guide to Java Certification: A Comprehensive Primer.

Overviews

- Defining classes
- Instance and static members
- Method signatures
- Overloading methods
- The `this` reference
- Constructors: default and non-default
- Overloading constructors
- Packages
- Accessibility modifiers for classes and interfaces: default and `private`
- Member accessibility modifiers: `public`, `protected`, default and `private`
- Abstract classes and methods
- Final classes, members and parameters

Defining Classes

- A class definition specifies a new type and its implementation.

  <*class header*> {
    <*class body*>
  }

- In the *class header,* the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:
  - Scope or accessibility modifier.
  - Additional class modifiers.
  - Any class it extends.
  - Any interfaces it implements.
- The *class body* can contain declarations of the following *members*:
  - Instance variables and methods (*instance members*)
  - Static variables and methods (*static members*)
  - *Constructors*
  - Nested classes *(inner classes)*
  - *Static and instance initializers*

Defining Methods

- The behavior of objects is specified by the instance methods of the class.
- The general syntax of a method definition is:

  <*method header*> (<*formal parameter list*>) <*throws clause*> {
    <*method body*>
  }

- Like member variables, member methods can be characterized as:
  - *Instance methods* belonging to the object of the class.
  - *Static methods* belonging only to the class.
- the *method header* must specify:
  - the *name* of the method
  - the *type* of the *return value*

  and, in addition, can specify
  - Scope or *accessibility modifier.*
  - Additional *method modifiers*.

- The *formal parameter list* is a comma-separated list of parameters, passing information to the method when the method is invoked by a *method call*.
  - Each parameter is a simple *variable declaration* consisting of its type and name.
- *Exceptions* thrown by the method can be specified using the `throws` clause.
- The *method body* specifies the *local declarations* and the *statements* of the method.

Statements

- A statement in Java is terminated by a semicolon (;).
- Variable declarations with explicit initialization of the variables are called *declaration statements*.
- An *expression statement* is an *expression* terminated by a semicolon:
  - Assignments
  - Increment and decrement operators
  - Method calls
  - Object creation with the `new` operator
- *Flow control statements* include the following: if-else, for, while, do-while, switch, break, continue, return, throw, try-catch-finally.
- Statements can also be labelled. `label:` *<statement>*
- A solitary semicolon (;) denotes the *empty statement* that does nothing.
- A block, `{}`, is a *compound* statement which can be used to group zero or more local declarations and statements.
  - It can be used in any context where a simple statement is permitted.

Instance Methods and Object Reference `this`

- Instance methods belong to every object of the class, and can only be invoked on objects of the class.
- The body of an instance method can access all members, including static members, defined in the class.
  - All instance methods are passed an implicit parameter, the `this` reference, which is a reference to the object on which the method is being invoked.
  - In the body of the method, the `this` reference can be used like any other object reference to access members of the object.
  - Note that the `this` reference cannot be modified.
- The simple name `member` is considered a short-hand notation for `this.member`.
- If, for some reason, a method needs to pass the object on which it is being invoked to another method, it can do so using the `this` reference.
- Note that no implicit `this` reference is passed to static methods, as these are not invoked on behalf of any object.

Local Variables and Instance Variables

- A local variable can *shadow* (also called *hiding*) a member variable that has the same name.
- The reference `this` can be used to distinguish the instance variables from the local variables inside the method.

Example 1 Using `this` *Reference*

```
class Light {
    // Instance variables
    int noOfWatts;          // wattage
    boolean indicator;      // on or off
    String location;        // placement
    // Constructor
    public Light(int noOfWatts, boolean indicator, String site) {
        String location;
        this.noOfWatts = noOfWatts;   // (1) Assignment to instance variable.
        indicator = indicator;        // (2) Assignment to parameter.
        location = site;              // (3) Assignment to local variable.
        this.someAuxilliaryMethod();  // (4)
        someAuxilliaryMethod();       // equivalent to call at (4)
    }
    void someAuxilliaryMethod() { System.out.println(this); }  // (5)
}
```

Method Overloading

- Each method has a *signature*, which is comprised of the name of the method and the types and order of the parameters in the parameter list.
- *Method overloading* allows a method with the same name but different parameters, thus with different signatures, to have different implementations and return values of different types.
- Rather than invent new method names all the time, method overloading can be used when the same operation has different implementations.
- The JDK APIs make heavy use of method overloading.

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```


- In the examples below, five implementations of the method `methodA` are shown:

```
public void methodA(int a, double b) {/* ... */}        // (1)
public int methodA(int a) { return a; }                 // (2)
public int methodA() { return 1; }                      // (3)
public long methodA(double a, int b) { return b; }      // (4)
public long methodA(int x, double y) { return x; }      // (5) Not OK.
```

- The corresponding signatures of the methods are as follows:

```
methodA(int, double)                        // (1')
methodA(int)                                // (2') Number of parameters.
methodA()                                   // (3') Number of parameters.
methodA(double, int)                        // (4') Order of parameters.
methodA(int, double)                        // (5') Same as (1').
```

- Changing just the return type (as shown at (3) and (4) below), or the exceptions thrown, in the implementation is not enough to overload a method, and will be flagged as a compile time error.
  - The *parameter list* in the definitions must be different.

```
void bake(Cake k) { /* ... */  }             // (1)
void bake(Pizza p) { /* ... */ }             // (2)
int     halfIt(int a) { return a/2; }        // (3)
double  halfIt(int a) { return a/2.0; }      // (4) Not OK. Same signature.
```

- At compile time, the right implementation is chosen based on the signature of the method call.

Constructors

- *The main purpose of constructors is to set the initial state of an object when the object is created using the* `new` *operator.*
- A constructor has the following general syntax:

```
<constructor header> (<parameter list>) {
   <constructor body>
}
```

- Constructors are like member methods, but the constructor header can only contain the following information:
  - Scope or accessibility modifier. Accessibility modifiers for methods also apply to constructors.
  - Constructor name, which must be the same as the class name.
- The following restrictions should be noted:
  - Modifiers other than accessibility modifiers are not permitted.
  - Constructors cannot return a value.
  - Constructors cannot specify exceptions in the header.

Default Constructor

- A *default constructor* is a constructor without any parameters.
- If a class does not specify *any* constructors, then an *implicit* default constructor is supplied for the class.
  - The implicit default constructor is equivalent to the following implementation:

```
<class name> () {}                  // No parameters. Empty constructor body.
```

- In the code below, the class `Light` does not specify any constructors.

```
class Light {
    // Instance variables
    int noOfWatts;          // wattage
    boolean indicator;      // on or off
    String location;        // placement

    // No constructors
    //...
}

class Greenhouse {
    // ...
    Light oneLight = new Light();     // (1) Call of implicit default constructor.
}
```

- The following implicit default constructor is generated and employed when a `Light` object is created at (1):

```
Light () { }
```

  - The instance variables of the object are initialized to their *default values*.
- A class can choose to provide an implementation of the default constructor.

```
class Light {
    // ...
    // Explicit Default Constructor
    Light() {                       // (1)
        noOfWatts = 50;
        indicator = true;
        location = new String("X");
    }
    //...
}

class Greenhouse {
    // ...
    Light extraLight = new Light();   // (2) Call of explicit default constructor.
}
```

What is different about the state of the objects created in the two examples above?

- If a class defines one or more constructors, it cannot rely on the implicit default constructor being generated.
  - If the class requires a default constructor, its implementation must be provided.

```
class Light {
    // ...
    // Only non-default Constructor
    Light(int watts, boolean state, String place) {          // (1)
        noOfWatts = watts;
        indicator = state;
        location = place;
    }
    //...
}

class Greenhouse {
    // ...
    Light moreLight = new Light(100,true,"Greenhouse");      // (2) OK.
//  Light firstLight = new Light();                          // (3) Error.
}
```


Overloaded Constructors

- Like methods, constructors can also be overloaded.
- Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked.

```
class Light {
    // ...
    // Explicit Default Constructor
    Light() {                                                // (1)
        noOfWatts = 50;
        indicator = true;
        location = new String("X");
    }
    // Non-default Constructor
    Light(int watts, boolean ind, String loc) {              // (2)
        noOfWatts = watts;
        indicator = ind;
        location = loc;
    }
    //...
}
class Greenhouse {
    // ...
    Light moreLight = new Light(100,true,"Greenhouse");      // (3) OK.
    Light firstLight = new Light();                          // (4) OK.
}
```


Packages

- A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces and subpackages.
- The *fully qualified name* of a package member is the "containment path" from the root package to the package member, specified using the dot (.) operator.

```
wizard.pandorasBox.artifacts.Ailment.
```

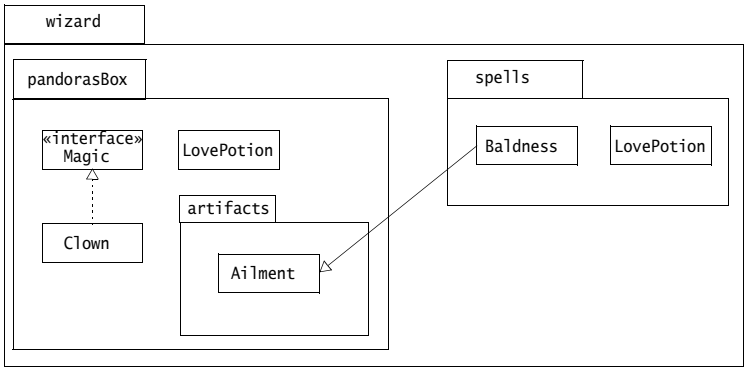

Figure 1 Package Hierarchy

Defining Packages

- A package hierarchy represents the organization of the *Java byte code* of classes and interfaces.
- Each Java source file (also called *compilation unit*) can contain zero or more definitions of classes and interfaces, but the compiler produces a separate *class* file containing the Java byte code for each of them.
- A class or interface can indicate that its Java byte code be placed in a particular package, using a `package` declaration.

  `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 compilation unit.
- The package name is saved in the Java byte code for the types contained in the package.

See Example 2.

Using Packages

- Types in a package must have the right accessibility in order to be referenced from outside the package.
- Given that a type is accessible from outside a package, a type can be accessed in two ways.
  - The first method is to use the fully qualified name of the type.
  - The second method is a short-hand for specifying the fully qualified name of the type, and involves using the `import` declaration.
- The `import` declarations must be the first statement after any `package` declaration in a source file.
- An `import` declaration does not recursively import subpackages.
- The `import` declaration does not result in inclusion of the source code of the types.
- The `import` declaration only imports type names.
- Name conflicts can be resolved using variations of the `import` declaration together with fully qualified names.

Individual Type Import:

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

- The *simple* name of the type (i.e. its identifier) can be used to access this particular type.

Import on Demand:

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

- This allows any types from the specified package to be accessed by its simple name.

```
package wizard.spells;                          // Package declaration
import wizard.pandorasBox.*;                    // (1)
import wizard.pandorasBox.artifacts.*;          // (2)

class Baldness extends Ailment {                // (3)
      wizard.pandorasBox.LovePotion tlcOne;     // (4)
      LovePotion tlcTwo;                        // (5)
}

class LovePotion { /*...*/ }
```

See Example 2.

Accessibility Modifiers for Classes and Interfaces

- Top-level classes and interfaces within a package can be specified as `public`.
  - This means that they are accessible from everywhere, both inside and outside of this package.
- The access modifier can be omitted (called *package* or *default accessibility*).
  - In this case they are only accessible in the package, but not in any subpackages.
- If a class is accessible, it does not automatically mean that members of the class are also accessible.
  - Member accessibility is governed separately from class accessibility, as explained in Section .
  - However, if the class is not accessible, its members will not be accessible, regardless of member accessibility.

Table 1 Summary of Accessibility Modifiers for Classes and Interfaces

Modifiers	Classes and Interfaces
default (No modifier)	Accessible in its package (package accessibility)
`public`	Accessible anywhere

Example 2 Accessibility Modifiers for Classes and Interfaces

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

public class Clown implements Magic {
     LovePotion tlc;
     /*...*/
}

class LovePotion {/*...*/}

interface Magic {/*...*/}
```

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

public class Ailment {/*...*/}
```

```
// File: Client.java
import wizard.pandorasBox.*;               // Import of classes.

public class Client {
    Clown performerOne;                    // OK. Abbreviated class name
    wizard.pandorasBox.Clown performerTwo; // OK. Fully qualified class name

//  LovePotion moreTLC;                    // Error. Not accessible
//  Magic magician;                        // Error. Not accessible
}
```


Other Modifiers for Classes

`abstract` Classes

- Any class can be specified with the keyword `abstract` to indicate that it cannot be instantiated.
- A class might choose to do this if the abstraction it represents is so general that it has to be specialized in order to be of practical use.

```
abstract class Vehicle { /* ... */ }    // too general
class Car extends Vehicle { /* ... */ } // more specific
class Bus extends Vehicle { /* ... */ }
```

- A class that has an `abstract` method (p. 54) must be declared `abstract`.
  - Such classes cannot be instantiated, as their implementation is only partial.
  - A class might choose this strategy to dictate certain behavior, but allow its subclasses the freedom to provide the relevant implementation.
  - A subclass which does not provide an implementation of its inherited methods is also abstract.
- Reference variables of an abstract class can be declared, but an abstract class cannot be instantiated.

Example 3 Abstract Classes

```
abstract class Light {
    // Instance variables
    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
    abstract public double KWHprice();              // (1) No method-body
}
class TubeLight extends Light {
    // Instance variables
    int tubeLength;
    int color;

    // Implementation of inherited abstract method.
    public double KWHprice() { return 2.75; }       // (2)
}
class Factory {
    TubeLight cellarLight = new TubeLight();        // (3) OK.
    Light spotlight;                                // (4) OK.
//  Light tableLight = new Light();                 // (5) Compile time error.
}
```

Interfaces are abstract.

- Interfaces just specify the method prototypes and not the implementation; they are, by their nature, implicitly abstract, i.e. they cannot be instantiated.
  - Thus specifying an interface with the keyword `abstract` is not appropriate, and should be omitted.

`final` Classes

- A `final` class cannot be extended.
  - Its behavior cannot be changed by subclassing.
  - It marks the lower boundary of its *implementation inheritance hierarchy.*
  - Only a class whose definition is complete (i.e. has implementations of all the methods) can be specified to be `final`.
- A final class must be *complete*, whereas an abstract class is considered incomplete.
  - Classes therefore cannot be both abstract and final at the same time.

```
final class TubeLight extends Light {
    // Instance variables
    int tubeLength;
    int color;

    // Implementation of inherited abstract method.
    public double KWHprice() { return 2.75; }
}
```

- The Java API includes many `final` classes, for example `java.lang.String` which cannot be specialized any further by subclassing.

Table 2 Summary of Other Modifiers for Classes and Interfaces

Modifiers	Classes	Interfaces
`abstract`	Class may contain abstract methods, and thus cannot be instantiated.	Implied.
`final`	The class cannot be extended, i.e. it cannot be subclassed.	Not possible.

Scope and Accessibility of Members

- Member scope rules govern where in the program a variable or a method is accessible.
- In most cases, Java provides explicit modifiers to control the accessibility of members, but in two areas this is governed by specific scope rules:
  - Class scope for members
  - Block scope for local variables

Class Scope for Members

- Class scope concerns accessing members (including inherited ones) by their simple names from code within a class.
- Accessibility of members from outside the class is primarily governed by the accessibility modifiers (p. 33).
- Static methods can only access static members (p. 46).
- *Within a class definition, reference variables of the class's type can be used to access all members regardless of their accessibility modifiers.*

Example 4 Class Scope

```
class Light {
    // Instance variables
    private int noOfWatts;          // wattage
    private boolean indicator;      // on or off
    private String location;        // placement

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

    public static Light duplicateLight(Light oldLight) {       // (1)
        Light newLight = new Light();
        newLight.noOfWatts = oldLight.noOfWatts;               // (2)
        newLight.indicator = oldLight.indicator;               // (3)
        newLight.location = new String(oldLight.location);     // (4)
        return newLight;
   }
}
```


Block Scope for Local Variables

- Declarations and statements can be grouped into a *block* using braces, {}.
- Note that the body of a method is a block.
- Blocks can be nested and certain scope rules apply to local variable declarations in such blocks.
- A local declaration can appear anywhere in a block.
- *Local variables* of a method are comprised of formal parameters of the method and variables that are declared in the method body.
  - A local variable can exist in different invocations of the same method, with each invocation having its own storage for the local variable.
- The general rule is that 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.
  - Block scope of a declaration begins from where it is declared in the block and ends where this block terminates.
  - It is not possible to declare a new variable if a local variable of the same name is already declared in the current scope.


```
public static void main(String args[]) {                  // Block 1
//  String args = "";         // (1) Cannot redeclare parameters.
    char digit;
    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
```

Figure 2 Block Scope

Member Accessibility Modifiers

- Accessibility modifiers for members help a class to define a *contract* so that clients know exactly what services are offered by the class.
- Accessibility of members can be one of the following:
  - `public`
  - `protected`
  - default (also called *package accessibility*)
  - `private`
- In UML notation, `+` , `#` and `-` as prefix to the member name indicates `public`, `protected` and `private` member access respectively, whereas no prefix indicates default or package access.

`public` Members

- Public access is the least restrictive of all the access modifiers.
- A `public` member is accessible everywhere, both in its class's package and in other packages where its class is visible.
  - This is true for both instance and static members.
- Subclasses can access their inherited public members directly, and all clients can access `public` members through an instance of the class.

Example 5 Public Accessibility of Members

```
// Filename: SuperclassA.java                          (1)
package packageA;

public class SuperclassA {
    public int superclassVarA;                      // (2)
    public void superclassMethodA() {/*...*/}       // (3)
}

class SubclassA extends SuperclassA {
    void subclassMethodA() { superclassVarA = 10; } // (4) OK.
}

class AnyClassA {
    SuperclassA obj = new SuperclassA();
    void anyClassMethodA() {
        obj.superclassMethodA();                    // (5) OK.
    }
}
```



```
// Filename: SubclassB.java                            (6)
package packageB;
import packageA.*;

public class SubclassB extends SuperclassA {
    void subclassMethodB() { superclassMethodA(); } // (7) OK.
}

class AnyClassB {
    SuperclassA obj = new SuperclassA();
    void anyClassMethodB() {
        obj.superclassVarA = 20;                    // (8) OK.
    }
}
```

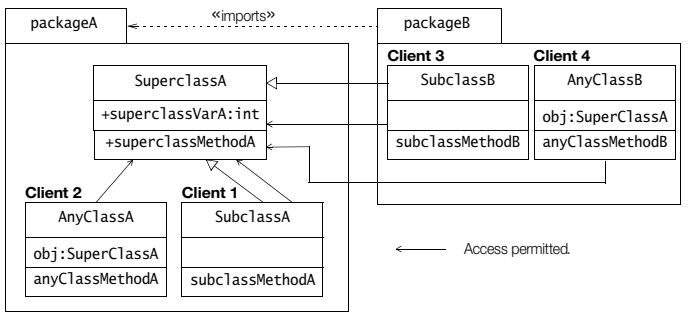

Figure 3 Public Accessibility

`protected` Members

- Protected members are accessible in the package containing this class, and by all subclasses of this class in any package where this class is visible.
  - In other words, non-subclasses in other packages cannot access protected members from other packages.
- *It is less restrictive than the default accessibility.*

```
public class SuperclassA {
    protected int superclassVarA;                    // (2)
    protected void superclassMethodA() {/*...*/}     // (3)
}
```

- Client 4 in package `packageB` cannot access these members, as shown in Figure 4.

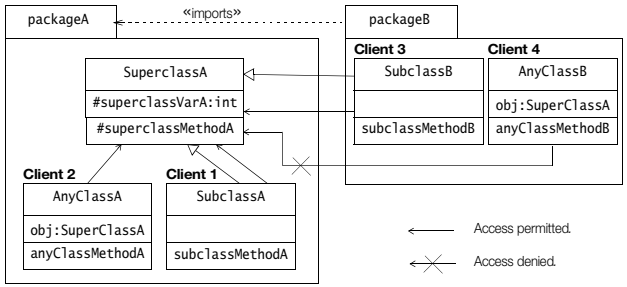

Figure 4 Protected Accessibility

- A subclass in another package can only access protected members in the superclass via references of its own type or a subtype.

```
//...
import packageA.*;
//...
public class SubclassB extends SuperclassA {         // In packageB.

    SuperclassA objRefA = new SubclassB();            // (1)
    SubclassB objRefB = new SubclassB();              // (2)

    void subclassMethodB() {
        objRefB.superclassMethodA();                  // (3) OK.
        objRefA.superclassVarA;                       // (4) Not OK.
    }
}
```

- The above restriction helps to ensure that subclasses in packages different from their superclass can only access protected members of the superclass in their part of the inheritance hierarchy.

Default Accessibility for Members

- When no access modifier is specified for a member, it is only accessible by another class in the package where its class is defined.
- Even if its class is visible in another (possibly nested) package, the member is not accessible there.

```
public class SuperclassA {
    int superclassVarA;                         // (2)
    void superclassMethodA() {/*...*/}          // (3)
}
```

- The clients in package `packageB` (i.e. Clients 3 and 4) cannot access these members.

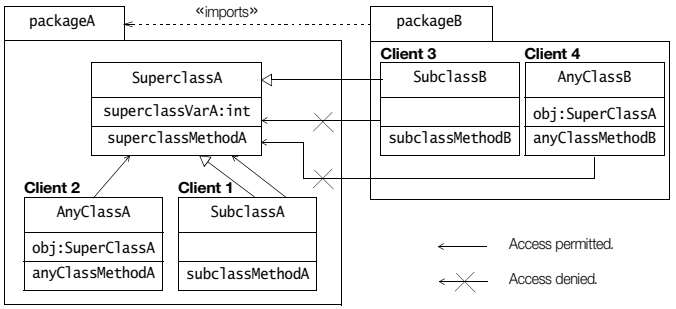

Figure 5 Default Accessibility

`private` Members

- This is the most restrictive of all the access modifiers.
- Private members are not accessible from any other class.
  - This also applies to subclasses, whether they are in the same package or not.
- It is not to be confused with inheritance of members by the subclass.
  - Members are still inherited, but they are not accessible in the subclass.
- It is a good design strategy to make all member variables private, and provide public accessor methods for them.
- Auxiliary methods are often declared private, as they do not concern any client.

```
public class SuperclassA {
    private int superclassVarA;                 // (2)
    private void superclassMethodA() {/*...*/}  // (3)
}
```

- None of the clients in Figure 6 can access these members.

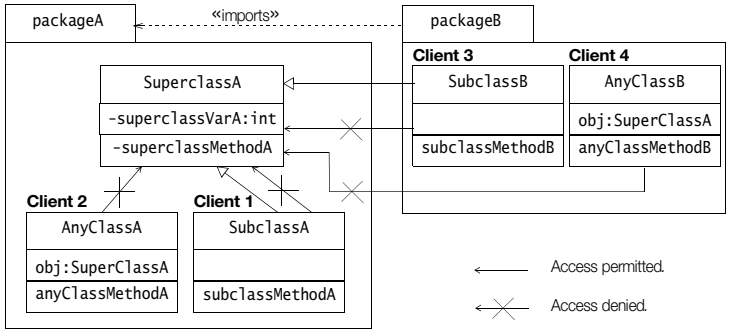

Figure 6 Private Accessibility

Table 3 Summary of Accessibility Modifiers for Members

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.

`static` Members

Static variables

- These are also called *class variables*) and they only exist in the class they are defined in.
- They are not instantiated when an instance of the class is created.
  - In other words, the values of these variables are not a part of the state of any object.
- When the class is loaded, static variables are initialized to their default values if no other explicit initialization is provided.

Static methods

- These are also known as *class methods*.
- A static method in a class can directly access other static members in the class.
- It cannot access instance (i.e. non-static) members of the class, as there is no object being operated on when a static method is invoked.
- Note, however, that a static method in a class can always use a reference of the class's type to access its members, regardless of whether these members are static or not.
- A typical static method might perform some task on behalf of the whole class and/or for objects of the class.

Example 6 Accessing Static Members

```
class Light {
    // Instance variables
    int noOfWatts;          // wattage
    boolean indicator;      // on or off
    String location;        // placement

    // Static variable
    static int counter;     // No. of Light objects created.         (1)

    // Explicit Default Constructor
    Light() {
        noOfWatts = 50;
        indicator = true;
        location = new String("X");
        ++counter;          // Increment counter.
    }

    // Static method
    public static void writeCount() {
         System.out.println("Number of lights: " + counter);      // (2)

         // Error. noOfWatts is not accessible
         // System.out.println("Number of Watts: " + noOfWatts);  // (3)
    }
}
```

```
public class Warehouse {
    public static void main(String args[]) {                    // (4)

        Light.writeCount();                         // Invoked using class name

        Light aLight = new Light();                 // Create an object
        System.out.println(
            "Value of counter: " + Light.counter  // Accessed via class name
        );
        Light bLight = new Light();                 // Create another object
        bLight.writeCount();                        // Invoked using reference
        Light cLight = new Light();                 // Create another object
        System.out.println(
            "Value of counter: " + cLight.counter // Accessed via reference
        );

    }
}
```

Output from the program:

```
Number of lights: 0
Value of counter: 1
Number of lights: 2
Value of counter: 3
```


`final` Members

Final Variables

- A `final` *variable* is a constant, and its value cannot be changed once it's initialized.
- Note that a `final` variable need not be initialized at its declaration, but it must be initialized once before it is used.
- These variables are also known as *blank final variables*.
- *Instance, static and local variables, including parameters* can be declared `final`.
- For `final` variables of primitive datatypes, it means that, once the variable is initialized, its value cannot be changed.
- For a `final` variable of a reference type it means that its reference value cannot be changed, but the state of the object it references may be changed.
- `Final` static variables are commonly used to define *manifest constants*, for example `Integer.MAX_VALUE`, which is the maximum `int` value.
- Variables defined in an interface are implicitly `final`.

Final Methods

- A `final` method in a class is complete (i.e. has an implementation) and cannot be overridden in any subclass.
- Subclasses are restricted in changing the behavior of the method.
- `Final` variables ensure that values cannot be changed, and `final` methods ensure that behavior cannot be changed.
- For `final` members, the compiler is able to perform certain code optimizations because certain assumptions can be made about such members.

Example 7 Accessing Final Members

```
class Light {
    // Final static variable                    (1)
    final public static double KWH_PRICE = 3.25;

    int noOfWatts;

    // Final instance methods                   (2)
    final public void setWatts(int watt) {
        noOfWatts = watt;
    }

    public void setKWH() {
      // KWH_PRICE = 4.10;                 // (3) Not OK. Cannot be changed.
    }
}

class TubeLight extends Light {
    // Final method cannot be overriden.
    // This will not compile.
    /*
    public void setWatts(int watt) {       // (4) Attempt to override.
        noOfWatts = 2*watt;
    }
    */
}
```

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

        final Light aLight = new Light();// (5) Final local variable.
        aLight.noOfWatts = 100;          // (6) OK. Changing object state.
    //  aLight = new Light();            // (7) Not OK. Changing final reference.
    }
}
```


`abstract` Methods

- An abstract method has the following syntax:

  `abstract` ... <*return type*> <*method name*> (<*parameter list*>) <*throws clause*>;

- An abstract method does not have an implementation, i.e. no method body is defined for an `abstract` method, only the method prototype is provided in the class definition.
- Its class is then `abstract` (i.e. incomplete) and must be explicitly declared as such (p. 23).
  - Subclasses of its class are then forced to provide the method implementation.
- A `final` method cannot be `abstract` (i.e. cannot be incomplete), and vice versa.
- Methods specified in an interface are implicitly `abstract`, as only the method prototypes are defined in an interface (Section 6.4, p. 195).