Object-oriented Programming - II

Reference:
Chapter 4 & Chapter 6

Overview
- Single implementation inheritance, multiple interface inheritance and supertypes
- Assigning, casting and passing references
- The instanceof operator
- Polymorphism and dynamic method lookup
- Choosing between inheritance and aggregation
- Encapsulation
- Packages: usage and definition
- 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

Interfaces

Interfaces: Programming by Contract
- Java provides interfaces which allow new type names to be introduced and used polymorphically, and also permit multiple interface inheritance.
- Interfaces support programming by contract.
Defining Interfaces

• An interface defines a contract by specifying prototypes of methods, and not their implementation.

```
@interface header
{
@interface body
}
```

• An interface is abstract by definition and therefore cannot be instantiated. It should also not be declared abstract.

• Reference variables of the interface type can be declared.

• Pure Design: Java allows contracts to be defined by using interfaces.

```
interface ISwitch {
    // method prototypes
    void switchOn();
    void switchOff();
    boolean isOn();
}
```

Design Inheritance: Implementing Interfaces

• A class specifies the interfaces it implements as a comma-separated list using the implements clause in the class header.

```
interface ISwitch {
    // method prototypes
    void switchOn();
    void switchOff();
    boolean isOn();
}
```

```
class Light implements ISwitch {
    // Method implementations
    public void switchOn() {...}
    public void switchOff() {...}
    public boolean isOn()   {...}
    //...}
```

```
class TubeLight extends Light {...}
```

A Word on Typing

Type rules determine which type of values (objects) can be manipulated by which type of variables (names).

• A variable of type \( T \) can denote any object of type \( T \) or a subtype of \( T \).

```
Light spotlight = new Light();
TubeLight ceilingLight = new TubeLight();
ISwitch toggle = ceilingLight;
```

Supertypes and Subtypes

• Interfaces define new types.

• Although interfaces cannot be instantiated, variables of an interface type can be declared.

• If a class implements an interface, then references to objects of this class and its subclasses can be assigned to a variable of this interface type.

• The interfaces that a class implements and the classes it extends, directly or indirectly, are called its supertypes.

• The class is then a subtype of its supertypes.

• A supertype is thus a reference type.

• Interfaces with empty bodies are often used as markers to “tag” classes as having a certain property or behavior (java.io.Serializable).

• Note that a class inherits only one implementation of a method, regardless of how many supertypes it has.

• All interfaces are also subtypes of object class.

A supertype reference can denote objects of its type and of its subtypes.
Example: Interface Types

```java
interface ISwitch {
    public void switchOn();
    public void switchOff();
    public boolean isOn();
}
class Light implements ISwitch {
    // Method implementations
    public void switchOn() {...}
    public void switchOff() {...}
    public boolean isOn() {...}
}
class TubeLight extends Light {...}
```

Remarks on Interfaces

- Classes which `implement` interfaces, guarantee that they will honor the contract.
- A class can choose to implement only some of the methods of its interfaces, i.e. give a partial implementation of its interfaces.
- The class must then be declared as `abstract`.
- Note that interface methods cannot be declared `static`, because they comprise the contract fulfilled by the `objects` of the class implementing the interface and are therefore instance methods.
- The interface methods will all have `public` accessibility when implemented in the class (or its subclasses).

Extending Interfaces

- An interface can extend other interfaces, using the `extends` clause.
- Unlike extending classes, an interface can `extend` several interfaces.
- Multiple inheritance of interfaces can result in an inheritance hierarchy which has multiple roots designated by different interfaces.
- Note that there are three different inheritance relations at work when defining inheritance between classes and interfaces:
  - Linear implementation inheritance hierarchy between classes: a class extends another class.
  - Multiple inheritance hierarchy between interfaces: an interface extends other interfaces.
  - Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.
- There is only one single `implementation` inheritance into a class, which avoids many problems associated with general multiple inheritance.

Kinds of Inheritance

- Linear implementation inheritance hierarchy between classes.
- Multiple interface inheritance hierarchy between interfaces and classes.
**Example 1 Interfaces**

```java
interface IStack { // (1)
    void push(Object item);
    Object pop();
}

class StackImpl implements IStack { // (2)
    protected Object[] stackArray;
    protected int tos;
    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        tos = -1;
    }
    public void push(Object item) { // (3)
        stackArray[++tos] = item;
    }
    public Object pop() { // (4)
        Object objRef = stackArray[tos];
        stackArray[tos] = null;
        tos--;
        return objRef;
    }
    public Object peek() { return stackArray[tos]; }
}
```

**interface ISafeStack extends IStack { // (5)**

```java
    boolean isEmpty();
    boolean isFull();
}

class SafeStackImpl extends StackImpl implements ISafeStack { // (6)
    public SafeStackImpl(int capacity) { super(capacity); }
    public boolean isEmpty() { return tos < 0; } // (7)
    public boolean isFull() { return tos == stackArray.length-1; } // (8)
}
```

```java
public class StackUser {    public static void main(String args[]) {                      // (9)
    SafeStackImpl safeStackRef = new SafeStackImpl(10);        StackImpl stackRef = safeStackRef;
    ISafeStack isafeStackRef = safeStackRef;IStack istackRef = safeStackRef;Object objRef = safeStackRef;
    safeStackRef.push("Dollars");                             // (10)
    stackRef.push("Kroner");
    System.out.println(isafeStackRef.pop());System.out.println(istackRef.pop());System.out.println(objRef.getClass());
    }
```
Constants in Interfaces

- An interface can also define constants.
- Such constants are considered to be public, static and final.
- An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface.
- A class that implements this interface or an interface that extends this interface, can also access such constants directly without using the dot (.) notation.
- Extending an interface which has constants is analogous to extending a class having static variables.
  - In particular, these constants can be shadowed by the subinterfaces.
  - In the case of multiple inheritance, any name conflicts can be resolved using fully qualified names for the constants involved.
  - The compiler will flag such conflicts.

Example 2 Variables in Interfaces

```java
interface Constants {
    double PI = 3.14;
    String AREA_UNITS = " sq.cm.;";
    String LENGTH_UNITS = " cm.;";
}

class Client implements Constants {
    public static void main(String args[]) {
        double radius = 1.5;
        System.out.println("Area of circle is " + (PI*radius*radius) + AREA_UNITS); // (1) Direct access.
        System.out.println("Circumference of circle is " + (2*PI*radius) + Constants.LENGTH_UNITS); // (2) Fully qualified name.
    }
}
```

Output from the program:
Area of circle is 7.0649999999999995 sq.cm.
Circumference of circle is 9.42 cm.

Using References

- Only primitive data and reference values can be stored in variables.
- Arrays are object in Java.
- Array types (boolean[], Object[], StackImpl[]) implicitly augment the inheritance hierarchy.
- All array types implicitly extend the Object class.
- Note the difference between arrays of primitive data types and reference types.
  - Arrays of reference types also extend the array type Object[].
- Variables of array reference types can be declared and arrays of reference types instantiated.
- An array reference exhibits the same polymorphic behavior as any other reference, subject to its location in the extended inheritance hierarchy.
Assigning, Passing and Casting References

- Reference values, like primitive values, can be assigned, cast and passed as arguments.
- For values of the primitive data types and reference types, conversions occur during:
  - Assignment
  - Parameter passing
  - Explicit casting
- The rule of thumb for the primitive data types is that widening conversions are permitted, but narrowing conversions require an explicit cast.
- The rule of thumb for reference values is that conversions up the inheritance hierarchy are permitted (called upcasting), but conversions down the hierarchy require explicit casting (called downcasting).
- The parameter passing conversion rules are useful in creating generic data types which can handle objects of arbitrary types.

Example 3 Assigning and Passing Reference Values

```java
interface IStack { /* See Example 1 for definition */ }
class StackImpl implements IStack { /* See Example 1 for definition */ }
interface ISafeStack extends IStack { /* See Example 1 for definition */ }
class SafeStackImpl extends StackImpl implements ISafeStack {
    /* See Example 1 for definition */
}
public class ReferenceConversion {
    public static void main(String args[]) {
        Object objRef;
        StackImpl stackRef;
        SafeStackImpl safeStackRef = new SafeStackImpl(10);
        IStack iStackRef;
        ISafeStack iSafeStackRef;
        // SourceType is a class type
        objRef = safeStackRef;       // (1) Always possible
        stackRef = safeStackRef;     // (2) Subclass to superclass assignment
        iStackRef = stackRef;        // (3) StackImpl implements IStack
        iSafeStackRef = safeStackRef; // (4) Always possible
        // SourceType is an array type.
        objRef = objArray;           // (7) Always possible
        objRef = stackArray;         // (8) Always possible
        objArray = stackArray;       // (9) Always possible
        stackArray = safeStackArray; // (13) Subclass array to superclass array
        iSafeStackArray = safeStackArray;      // (14) Always possible
        // Parameter Conversion
        System.out.println("First call: ");
        sendParams(stackRef, safeStackRef, iStackRef,
                    safeStackArray, iSafeStackArray);                     // (15)
        System.out.println("Second call: ");
        sendParams(iSafeStackArray, stackRef, iSafeStackRef,
                    StackImpl[], ISafeStack[]);                         // (16)
    }
}
```

Figure 2 Array Types in Inheritance Hierarchy
public static void sendParams(Object objRefParam, StackImpl stackRefParam, IStack iStackRefParam, StackImpl[] stackArrayParam, IStack[] iStackArrayParam) {
    // Signature: sendParams(Object, StackImpl, IStack, StackImpl[], IStack[])
    System.out.println(objRefParam.getClass());
    System.out.println(stackRefParam.getClass());
    System.out.println(iStackRefParam.getClass());
    System.out.println(stackArrayParam.getClass());
    System.out.println(iStackArrayParam.getClass());
}

Output from the program:
First call:
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;
Second call:
class [LSafeStackImpl;
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;

Reference Casting and instanceof Operator

- The expression to cast <reference> of <source type> to <destination type> has the following syntax:
  <destination type> <reference>

- The binary instanceof operator has the following syntax:
  <reference> instanceof <destination type>

- The instanceof operator (note that the keyword is composed of only lowercase letters) returns the value true if the left-hand operand (any reference) can be cast to the right-hand operand (a class, interface or array type).

- A compile time check determines whether a reference of <source type> and a reference of <destination type> can denote objects of a class (or its subclasses) where this class is a common subtype of both <source type> and <destination type> in the inheritance hierarchy.

- At runtime, it is the actual object denoted by the reference that is compared with the type specified on the right-hand side.

- Typical usage of the instanceof operator is to determine what object a reference is denoting.

Example: Boys and Girls are Human

- We use names to denote humans.
- We can have boy names.
  Boy tom, dick, harry;
- We can have girl names:
  Girl pandora, venus, medusa;
- Some human names can be for a boy or a girl.
  Human pat, sandy, alex;
  A Boy is a Human.
  A Girl is a Human.
- We can create people and give them names:
  tom = new Boy();
  pandora = new Girl();
  sandy = new Girl(); // A Girl is a Human
  alex = tom; // Alias of the same boy. A Boy is a Human.
  pandora = sandy; // Danger!
  // It is not always the case that a Human is a Girl.
  // What if sandy was a Boy?
  pandora = (Girl) sandy; // Compiler requires a cast.

- Names can be used to manipulate people.
- We cannot write
  sandy.giveBirth();
  as we do not know if sandy is a Girl, and only a Girl can give birth.
- We might try:
  ((Girl) sandy).giveBirth(); // Compiler is happy.
  or
  pandora = (Girl) sandy; // Compiler is happy.
  pandora.giveBirth();
- What if at runtime sandy was really a Boy?
  In that case we get an execution error, ClassCastException.
  Obviously we cannot have a Girl name denote a Boy.
- We should only cast if we are sure that sandy is a Girl.
  if(sandy instanceof Girl) { // Full proof
    pandora = (Girl) sandy; // Compiler is happy.
    pandora.giveBirth();
    // Alternative
    (Girl) sandy).giveBirth();
  }
A cast does not change the type of the object, and only the reference value is cast.

- Note that both variables and objects have a type.
- The type of a variable or an object cannot be changed.

**Objects** are people, i.e. either boys and girls.
**Variables** are names of people.
  i.e. use names to denote people.
  In Java, you use variables to denote objects.

- *Human* can be an abstract class.
  - It provides partial implementation for a human.
  - For example, a *getGender()* method must be defined for all humans.
- *Boy* and *Girl* can be final classes.
  - We do not want boys and girls to be further specialized.

---

**Upcasting and Downcasting**

- **Upcasting**: References of supertypes can be declared, and these can denote objects of its type or its subtypes.
- **Downcasting**: Converting a supertype reference to a subtype reference requires explicit casting.
  - Use the *instanceof* operator to make sure the cast is legal.

---

**Special case: Converting References of Class and Interface Types**

- **Upcasting**: References of interface type can be declared, and these can denote objects of classes that implement this interface.
- **Downcasting**: Converting a reference of interface type to the type of the class implementing the interface requires explicit casting.

```java
IStack istackOne = new StackImpl(5); // Upcasting
StackImpl stackTwo = (StackImpl) istackOne; // Downcasting
Object obj1 = istackOne.pop(); // OK. Method in IStack interface.
Object obj2 = istackOne.peek(); // Not OK. Method not in IStack interface.
```

---

**Sub is Sup**

- Upcasting
- Sub is Sup
  ```java
  Sup a = b; // there b is Sub ref
  ```
- Downcasting
  ```java
  if (a instanceof Sub) // there a is Sup ref
     Sub b = (Sub) a;
  ```
Example: Upcasting and Downcasting

interface ISwitch { // method prototypes
    public void switchOn();
    public void switchOff();
    public boolean isOn();
}

class Light implements ISwitch {...}
class LightBulb extends Light {...}
class TubeLight extends Light {...}
class NeonLight extends TubeLight {...}

// Some client
boolean result1, result2, result3, result4, result5;
ISwitch light1 = new LightBulb(); // (1)
    // String str = (String) light1; // (2) Compile error.
    // result1 = light1 instanceof String; // (3) Compile error.
    // result2 = light1 instanceof TubeLight; // (4) false. Peer class.
    // TubeLight tubelight1 = (TubeLight) light1; // (5) ClassCastException.
    // light1 = new NeonLight(); // (6)
    if (light1 instanceof TubeLight) { // (7) true
        Tubelight tubelight2 = (Tubelight) light1; // (10) OK
        // Can now use tubelight2 to access object of class NeonLight.
    }
Example 4  Polymorphism and Dynamic Method Lookup

```java
interface IDrawable {
    void draw();
}
class Shape implements IDrawable {
    public void draw() { System.out.println("Drawing a Shape."); }
}
class Circle extends Shape {
    public void draw() { System.out.println("Drawing a Circle."); }
}
class Rectangle extends Shape {
    public void draw() { System.out.println("Drawing a Rectangle."); }
}
class Square extends Rectangle {
    public void draw() { System.out.println("Drawing a Square."); }
}
class Map implements IDrawable {
    public void draw() { System.out.println("Drawing a Map."); }
}
```

```java
public class PolymorphRefs {
    public static void main(String args[]) {
        Shape[] shapes = {new Circle(), new Rectangle(), new Square()};   // (1)  
        IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};   // (2)  
        System.out.println("Draw shapes:");
        for (int i = 0; i < shapes.length; i++)  // (3)
            shapes[i].draw();
        System.out.println("Draw drawables:");
        for (int i = 0; i < drawables.length; i++)  // (4)
            drawables[i].draw();
    }
}
```

Output from the program:

Draw shapes:
Drawing a Circle.
Drawing a Rectangle.
Drawing a Square.

Draw drawables:
Drawing a Shape.
Drawing a Rectangle.
Drawing a Map.

Static and Dynamic Binding

- Compile time: Static binding
  - the compiler verifies that the type of the element `drawables[i]` has a method called `draw()`.
- Runtime: Dynamic Binding
  - the runtime system invokes the `draw()` method of the actual object that is denoted by `drawables[i]`.
  - method `draw()` represents different methods depending on the type of the object denoted by `drawables[i]`.

Programming by Contract

```java
public void drawAll(IDrawable[] figures) {
    for (int i = 0; i < figures.length; i++)
        figures[i].draw();
}
```

Note that elements of array `IDrawable` can be objects of `Shape`, `Circle`, `Square`, `Rectangle` or `Map`.

Choosing between Inheritance and Aggregation

- Choosing between inheritance and aggregation to model relationships can be a crucial design decision.
- A good design strategy advocates that inheritance should be used only if the relationship `is-a` is unequivocally maintained throughout the lifetime of the objects involved, otherwise aggregation is the best choice.
- A role is often confused with an `is-a` relationship.
  - Changing roles would involve a new object to represent the new role every time this happened.
- Code reuse is also best achieved by aggregation when there is no `is-a` relationship.
- Aggregation with `method delegating` can result in robust abstractions.
- Both inheritance and aggregation promote encapsulation of `implementation`, as changes to the implementation are localized to the class.
- Changing the `contract` of a superclass can have consequences for the subclasses (called the ripple effect) and also for clients who are dependent on a particular behavior of the subclasses.
Achieving Polymorphism

- Polymorphism is achieved through inheritance and interface implementation.
- Code relying on polymorphic behavior will still work without any change if new subclasses or new classes implementing the interface are added.
- If no obvious is-a relationship is present, then polymorphism is best achieved by using aggregation with interface implementation.

![Figure 4 Inheritance and Aggregation](image)

**Example 5 Inheritance and Aggregation**

```java
class Node { // (1)
    private Object data; // Data
    private Node next; // Next node
    // Constructor for initializing data and reference to the next node.
    public Node(Object obj, Node link) {
        data = obj;
        next = link;
    }
    // Accessor methods
    public void setData(Object obj) { data = obj; }
    public Object getData() { return data; }
    public void setNext(Node node) { next = node; }
    public Node getNext() { return next; }
}

class LinkedList { // (2)
    protected Node head = null;
    protected Node tail = null;
    // Modifier methods
    public void insertInFront(Object dataObj) {
        if (isEmpty()) head = tail = new Node(dataObj, null);
        else head = new Node(dataObj, head);
    }
    public void insertAtBack(Object dataObj) {
        if (isEmpty()) head = tail = new Node(dataObj, null);
        else tail.setNext(new Node(dataObj, null));
        tail = tail.getNext();
    }
    public Object deleteFromFront() {
        if (isEmpty()) return null;
        Node removed = head;
        if (head == tail) head = tail = null;
        else head = head.getNext();
        return removed.getData();
    }
    // Selector method
    public boolean isEmpty() { return head == null; }
}

class QueueByAggregation { // (3)
    private LinkedList qList;
    // Constructor
    public QueueByAggregation() {
        qList = new LinkedList();
    }
    // Methods
    public void enqueue(Object item) { qList.insertAtBack(item); }
    public void insertAtBack(Object dataObj) {
        if (isEmpty())
            head = tail = new Node(dataObj, null);
        else {
            tail.setNext(new Node(dataObj, null));
            tail = tail.getNext();
        }
    }
    public Object dequeueFromFront() {
        if (isEmpty()) return null;
        Node removed = head;
        head = tail = new Node(dataObj, null);
        if (head == tail) head = tail = null;
        else head = head.getNext();
        return removed.getData();
    }
    // Selector method
    public boolean isEmpty() { return head == null; }
}
```
public Object dequeue() {
    if (empty()) return null;
    else return qList.deleteFromFront();
}

public Object peek() {
    Object obj = dequeue();
    if (obj != null) qList.insertInFront(obj);
    return obj;
}

public boolean empty() { return qList.isEmpty(); }
}

class StackByInheritance extends LinkedList {                  // (4)
    public void push(Object item) { insertInFront(item); }
    public Object pop() {
        if (empty()) return null;
        else return deleteFromFront();
    }
    public Object peek() { return (isEmpty() ? null : head.getData()); }
    public boolean empty() { return isEmpty(); }
}

public class Client {                                          // (5)
    public static void main(String args[]) {
        String string1 = "Queues are boring to stand in!";
        int length1 = string1.length();
        QueueByAggregation queue = new QueueByAggregation();
        for (int i = 0; i<length1; i++)
            queue.enqueue(new Character(string1.charAt(i)));
        while (!queue.empty())
            System.out.print((Character) queue.dequeue());
        System.out.println();
        String string2 = "!no tis ot nuf era skcatS";
        int length2 = string2.length();
        StackByInheritance stack = new StackByInheritance();
        for (int i = 0; i<length2; i++)
            stack.push(new Character(string2.charAt(i)));
        stack.insertAtBack(new Character('!'));                // (6)
        while (!stack.empty())
            System.out.print((Character) stack.pop());
        System.out.println();
    }
}

Output from the program:
Queues are boring to stand in!
Stacks are fun to sit on!!

Encapsulation

Abstraction and Encapsulation

- In OOSD, abstraction for an object comes before its implementation.
- Abstraction focuses on visible behavior of an object (called the contract or interface), whereas encapsulation focuses on implementation which give this behavior.
- Encapsulation helps to make clear the distinction between an object’s contract and implementation.
  - Objects are regarded as “black boxes” whose internals are hidden.
A class has 2 views:
- Contract which corresponds to our abstraction of behaviors common to all instances of the class.
  - specification/documentation on how clients (i.e. other objects) can use instances of the class, in particular, how each method should be called.
  - WHAT the class offers.
- Implementation which corresponds to representation of the abstraction and the mechanisms which give provide the desired behaviors.
  - comprises of the instance variables and Java instructions which, when executed, give the desired behavior.
  - HOW the class implements its behaviors - not a concern of the clients.

This separation of concerns has major implications in system design.
Encapsulation: Consequences for Program Development

- Results in programs that are "black boxes".
- Implementation of an object can change without implications for the clients.
- Reduces dependency between program modules (hence complexity), as the internals of an object are hidden from the clients, who cannot influence its implementation.
- Encourages code-reuse.

Encapsulation Levels

Packages

Classes

Data

Methods

Blocks

Data

Packages in Java

What is a Package?

- 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.
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:
  ```java
  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.

```java
// File: Clown.java
package wizard.pandorasBox;               // Package declaration
public class Clown implements Magic {
    LovePotion tlc;
    public static void main(String[] args) {
        System.out.println("Mamma mia!");
    }
}
class LovePotion {/*...*/}
interface Magic {/*...*/}
```

Compiling Packages

- In file Clown.java, the statement
  ```java
  package wizard.pandorasBox;               // Package declaration
  ```
  instructs the compiler to place the Java byte code for all the classes (and interface) specified in the source file in the package wizard.pandorasBox.
- Think of a package name as a path in the file system.
  In this case the package name wizard.pandorasBox corresponds to the path name wizard/pandorasBox.
  So the Java byte code for all the classes (and interface) specified in the source file Clown.java will be placed under the catalog wizard/pandorasBox.

Question: where is wizard/pandorasBox?

- You can choose the location of wizard/pandorasBox by using the -d option (d for destination) when compiling with javac.
- Assume that the current directory is called work, and the source code file Clown.java is to be found here.
  - The command
    ```bash
    javac -d . Clown.java
    ```
    (Note the dot after the -d option to indicate the current directory)
    will create wizard/pandorasBox under the current directory, and place the Java byte code for all the classes (and interface) in it.
    - Without the -d option, the file Clown.class will be placed in the current directory.
Question: How do we run the program?

- Since the current directory is work and we want to run Clown.class, the *full name* of the Clown class must be specified in the java command:
  
  ```java wizard.pandorasBox.Clown```

  This will now result in the main() method from the wizard.pandorasBox.Clown class being executed.

---

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.

---

Modifiers for Classes and Interfaces

- The simple name of the type (i.e. its identifier) can be used to access this particular type.
- This allows any types from the specified package to be accessed by its simple name.

```java
package wizard.spells; // Package declaration
import wizard.pandorasBox.*; // Import on demand
import wizard.pandorasBox.artifacts.Ailment; // Individual type import
class Baldness extends Ailment {
    Magic hairyTrick;
    wizard.pandorasBox.LovePotion tlcOne;
    LovePotion tlcTwo; // From wizard.pandorasBox package
}
class LovePotion { /*...*/ } // From wizard.spells package
```

- The classpath or sourcepath options of the javac compiler can be used to indicate the location of the imported classes and interfaces.
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.
- However, if the class is not accessible, its members will not be accessible, regardless of member accessibility.

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Classes and Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>default (No modifier)</td>
<td>Accessible in its package (package accessibility)</td>
</tr>
<tr>
<td>public</td>
<td>Accessible anywhere</td>
</tr>
</tbody>
</table>

Table 1 Summary of Accessibility Modifiers for Classes and Interfaces

Example 6 Accessibility Modifiers for Classes and Interfaces

```java
// File: Clown.java
package wizard.pandorasBox;               // Package declaration
public class Clown implements Magic {
    LovePotion tlc; /*...*/
} 
class LovePotion {/*...*/}
interface Magic {/*...*/}
```

```java
// File: Ailment.java
package wizard.pandorasBox.artifacts;     // Package declaration
public class Ailment {/*...*/}
```

```java
// 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.

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

- A class that has an abstract method 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 7 Abstract Classes

```java
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.

```java
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

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>Class may contain abstract methods, and thus cannot be instantiated.</td>
<td>Implied.</td>
</tr>
<tr>
<td>final</td>
<td>The class cannot be extended, i.e. it cannot be subclassed.</td>
<td>Not possible.</td>
</tr>
</tbody>
</table>

Modifiers for Members
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.
- Static methods can only access static members.
- Within a class definition, reference variables of the class’s type can access all members regardless of their accessibility modifiers.
  
i.e. if obj1 and obj2 are objects of the same class, instance methods of obj1 can access all members of obj2, and vice versa.

Example 8 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();               // (2)
        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.
                //      int i;        // (3) Already declared in enclosing block 1.
            } // 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
        int index;                // (7) OK
    } // for
} // main

Figure 6 Block Scope

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.

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 (in the order of decreasing accessibility):
  - 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.

Example 9 Public Accessibility of Members

```java
// 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 anyMethodA() {
        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.
  }
}

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.

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

- Client 4 in package packageB cannot access these members, as shown in Figure 8.

![Figure 7 Public Accessibility](image1)

![Figure 8 Protected Accessibility](image2)
A subclass in another package can only access protected members in the superclass via references of its own type or a subtype.

```java
//...
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.

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

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

---

**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.

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

- None of the clients in Figure 10 can access these members.
**Figure 10 Private Accessibility**

**Table 3 Summary of Accessibility Modifiers for Members**

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Accessible everywhere.</td>
</tr>
<tr>
<td>protected</td>
<td>Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.</td>
</tr>
<tr>
<td>default (no modifier)</td>
<td>Only accessible by classes, including subclasses, in the same package as its class (package accessibility).</td>
</tr>
<tr>
<td>private</td>
<td>Only accessible in its own class and not anywhere else.</td>
</tr>
</tbody>
</table>

**Other Modifiers for Members**

**Table 4 Summary of Other Modifiers for Members**

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Variables</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>Defines a class variable.</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>Defines a constant.</td>
<td></td>
</tr>
<tr>
<td>abstract</td>
<td>Not relevant.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines a class method.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The method cannot be overridden.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>No method body is defined. Its class is then implicitly abstract.</td>
<td></td>
</tr>
</tbody>
</table>

**Static Variables**
- They only exist in the class they are defined in.
- They are not instantiated when an instance of the class is created.
- When the class is loaded, static variables are initialized to their default values if no other explicit initialization is provided.

**Static 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 10  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;        // (5)
        indicator = true;      // (6)
        location = new String("X");
        ++counter;          // Increment counter.          // (7)
    }

    // 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 11 Accessing Final Members

class Light {
    // Final static variable
    final public static double KWH_PRICE = 3.25;
    int noOfWatts;

    // Final instance methods
    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 overridden.
    // 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.
    }
}

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.
- 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.

OOP (Object-oriented Programming)

OOP is an implementation methodology,
where a program is organized as a collection of co-operating objects,
where each object is an instance of a class, and
where all classes are members in a hierarchy of classes based on inheritance relationships.
- If the last criteria is not fulfilled, the implementation methodology is called OBP (Object-based Programming).
- One major consequence of OOP is that it promotes code-reuse, i.e. that classes can be “recycled”.

Data + Methods = Objects