Interfaces

- Using the keyword interface, you can **fully abstract** a class' interface from its implementation.

- That is, using interface, you can **specify what a class must do, but not how it does it.**
- Interfaces are **syntactically similar to classes**, but they **lack instance variables**, and their **methods are declared without any body.**

- In practice, this means that you can define interfaces which **don't make assumptions** about **how they are implemented**.

- Once it is defined, **any number of classes can implement an interface.**
- Also, **one class can implement any number of interfaces**.

- To implement an interface, **a class must create the complete set of methods defined by the interface**.

- However, **each class is free to determine the details** of its own implementation.

- By providing the interface keyword, Java allows you to fully utilize the "**one interface, multiple methods**" aspect of polymorphism.

- Interfaces are designed to support **dynamic method resolution at run time**.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.
- This requirement by itself makes for a static and nonextensible classing environment.
- Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.
- Interfaces are designed to avoid this problem.
- They **disconnect the definition of a method or set of methods from the inheritance hierarchy.**
- Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.
- This is where the real power of interfaces is realized.

Defining an Interface

- This is the general form of an interface:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);

    type final-varname1 = value;
    type final-varname2 = value;

    // ...

    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- Here, **access** is **either public or not used**.
- When **no access specifier** is included, then the interface is **only available to other members of the package** in which it is declared.
- When it is declared as **public**, the interface **can be used by any other code.**

- ***name*** is the name of the interface, and can be any valid identifier.

- Notice that the **methods** which are declared **have no bodies**.
- They **end with a semicolon** after the parameter list.
- They are, essentially, **abstract methods**; **there can be no default implementation of any method specified within an interface**.

- Each **class that includes an interface must implement all of the methods.**

- **Variables** can be declared inside of interface declarations.

- They are **implicitly final and static**, meaning **they cannot be changed by the implementing class**.

- They **must** also **be initialized with a constant value**.

- All **methods and variables** are **implicitly public if the interface**, itself, is declared as **public**.

Implementing Interfaces

- Once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the implements clause looks like this:

access* class *classname [*extends *superclass]

[implements *interface [,interface...]] {*

// class-body

}

- Here, **access** is either **public or not used**.
- **If a class implements more than one interface**, the interfaces are **separated with a comma**.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The **methods** that implement an interface **must be** declared **public**.
- Also, the **type signature** of the implementing method **must match exactly** the type signature **specified in the interface** definition.

- **Note:**

  When you implement an interface method, it must be declared as **public.**

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

Accessing Implementations Through Interface References

- You can **declare variables** as **object references that uses an interface rather than a class type**.
- **Any instance of any class that implements the declared interface can be referred** to by such a variable.
- When you call a method through one of these references, **the correct version** will be **called based on the actual instance of the interface being referred to**.
- This is one of the key features of interfaces.

- The **method** to be executed is **looked up dynamically at run time**, allowing **classes to be created later than the code which calls methods on them**.

- The **calling code** can **dispatch through an interface without having to know anything about the “callee.**”

- This process is **similar to using a superclass reference to access a subclass object.**

Partial Implementations

- If a **class includes an interface but does not fully implement the methods** defined by that interface, then that class **must be declared as abstract**.
- For example:

```
abstract class Incomplete implements Callback {
   int a, b;
   void show() {
         System.out.println(a + " " + b);
   }
   // ...
}
```

- Here, the class Incomplete does not implement callback( ) and must be declared as abstract.
- Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Interfaces Can Be Extended

- One **interface can inherit another** by the use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.