Abstract

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses leaving it to each subclass to fill in the details. Such classes determines the nature of the methods. Java's solution to this problem is the abstract method.

You can require that certain method be overridden by subclasses by specifying the abstract type modifier.

abstract type name (parameter list);

These methods are sometimes referred to as subclass's responsibility because they have no implementation specified in superclass.

Thus a super-class must override them - it cannot simply use the version defined in the superclass.

Any class that contains one or more abstract methods must be declared abstract.

There cannot be a objects of abstract class

You cannot declare abstract constructors or abstract static methods.

You can declare static methods in abstract class.

Because there can be no object for abstract class. If they had allowed to call abstract static methods, it would mean that we are calling an empty method (abstract) through classname because it is static.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to runtime polymorphism is implemented through the use of superclass references.

A Public Constructor on an abstract class doesn't make any sense because you cannot instantiate an abstract class directly.
(Can only instantiate through a derived type that ~~does~~ itself is not marked abstract)

Abstract class vs Interface

Type of methods:
Interfaces can have only abstract methods
Abstract class can have abstract and non abstract methods.
(From Java 8, it can be default and static methods also)

Final Variables

Variables declared in Java interface are by default final.
An abstract class may contain non-final variables.

Types of variables:
Abstract classes can have final, non-final static and non static variables.
Interface has only static and final variables.

Inheritance vs Abstraction
A Java interface can be implemented using keyword "implements"
& abstract class can be extended using keyword 'extends'

Multiple implementation:
An interface can extend another Java interface only.
an abstract class can extend another Java class and implement multiple Java interface

Accessibility of Data Members:
Members of a Java Interface are only public by default
A Java abstract class can have class members like private and protected etc.

Interface

Multiple Inheritance is not available in Java
(same function in two classes, it will skip that hence no multiple inheritance)

Instead we have Java interfaces.
they have abstract functions (no body of functions)

Interface is like class but not completely.
It is like an abstract class.
By default functions are public and abstract in interface. variables are final and static by default in interface.

Interfaces specify only what the class is doing, not how it is doing it.
The problem with Multiple Inheritance is that two classes may define different ways of doing the same thing, and the Subclass can't choose which one to pick.

Key differences between a class and an interface:
a class can maintain state information
(especially through the use of instance variable)
but an interface cannot.

Using interface, you can specify a set of
methods that can be implemented by one or more
classes.
Although they are similar to abstract classes
interfaces have an additional ~~Catchlib~~ capability:
A class can implement more than one interface
By contrast, a class can only inherit a single
superclass (abstract or otherwise).

Using the keyword interface, you can fully
abstract a class' interface from its implementa-
-tion. That is using interface, you can
specify what a class must do but ~~how~~ not
how it does it.

Interfaces are syntactically similar to classes
but they lack instance variable and as a
general rule, their methods are declared
without any body.

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

NOTE: 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 non extensible 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 decouple the definition of a method or a 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 class hierarchy to implement same interface.

Beginning with JDK8, it is possible to add a default implementation to an interface method.
Thus it is possible for interface to specify some behaviour.
However, default methods constitute what is, in essence, a special-use feature, and the original intent behind interface still remains.

Variables can be declared inside of interface declarations.
Note: 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.

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

NOTE: You can declare variables as object interface reference that use an interface rather than a class type.
This process is similar to using a superclass reference to access a subclass object.
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. Called at runtime by the type of object it refers to.

This method to be executable referred is looked up dynamically at run time, allowing classes to be created later than the code which calls method on time.

The calling code can dispatch through an interface without having to know anything about ~~the~~ the "callee".

CAUTION: Because dynamic lookup of a method at runtime incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in ~~performance~~ performance-critical code.

Nested Interface:

A nested interface can be declared a member of a class or another interface
Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private or protected.
This differs from a top level interface, which must either be declared as public or use the default access level.

```
// This class contains a nested interface
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}
// B implements the nested interface
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}
class NestedIFDemo {
  public static void main(String args[]) {
    // use a nested interface reference
    A.NestedIF nif = new B();
    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

Interface can be Extended:
one interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.
Any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Default interface method (aka extension method):
A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. In all the classes that implements that interface.
Ex.

```
default String getString(){
    return "Default String";
}
```

For example you might have a class that implements two interfaces.
If each of ~~these~~ these interfaces provide default methods, then some behaviour is inherited from both.
In all cases, a class implementation takes priority over an interface default implementation
In cases in which a class implements two interfaces that both have the same default method, but the class does not override the method, then an error will result.
In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence.

Note: Static interface methods are not inherited by either an implementing class or sub interface.
So static interface method should have a body. They cannot be abstract.
Note: When overriding methods, the access modifier should be same or better
if Parent class was Protected, overrider should be protected or public