FAQ

(Frequently Asked Questions)

Table of Contents

What are the differences between instance methods and class methods?

Instance methods can be invoked only on an instance of a class. Hence to invoke an instance method, you need to first create an instance of a class. Instance can operate on both static and non-static members of the class.
Class methods are methods which are declared as static. These methods can be invoked on an instance of a class as well as the class itself. Class methods can only operate on static members of a class.
Note that instance methods of a class can also not be called from within a class method unless they are being called on an instance of that class.

How do you use the ternary operator?

The ternary operator ?:, is a shorthand of an if-else statement. It can be used to evaluate an expression and return one of two operands depending on the result of the expression.

```
boolean b = true;
String s = ( b == true ? "True" : "False" );
```

This will set the value of the String s according to the value of the boolean b. This could be written using an if else statement as follows:

```
boolean b = true;
String s;
if(b == true)
{
    s = "True";
}
else
{
    s = "False";
}
```

The ternary operator can be very useful in reducing the number of lines of code as you can see, but beware it can make your code difficult to read and due to this is not ideal for nested statements. It can also be very useful in testing for null values quickly.

Can Java handle nested switch statements?

Yes. Nested switch statements are allowed. Just make sure you remember to have all the break statements in the correct places.
E.g.:

```
int i;
int j;
switch(i)
{
    case 0: System.out.println("i is 0"); break;
    case 1: System.out.println("i is 1");
      switch(j)
      {
          case 0: System.out.println("j is 0"); break;
          case 1: System.out.println("j is 1"); break;
          default: System.out.println("j is not known");
      }
            break; //note this break after 2nd switch
    default: System.out.println("i is not known");
}
```

What are the differences between String.concat(), StringBuffer.append(), and the '+' operator (applied to Strings)?

They all do basically the same thing, except that they do it in a different way and with different performances. Basically String.concat() and "+" concatenates two strings, while StringBuffer.append() appends a string to a StringBuffer.
The great difference is that String is an immutable object, while StringBuffer is not. This means that every time you assign a value to a String concatenating many Strings, you are recreating a new String every time, while, using a StringBuffer you are appending the additional String to the buffer.
Try these two pieces of code and see the result:

1)

```
String s = new String();
long start = System.currentTimeMillis();
for (int i=0; i<10000; i++)
{
    s += "a"; // or s = s.concat("a");
}
long stop = System.currentTimeMillis();
System.out.println("String = " + (stop-start) + "ms.");
```

2)

```
StringBuffer sb = new StringBuffer();
start = System.currentTimeMillis();
for (int i=0; i<10000; i++)
{
    sb.append("a");
}
stop = System.currentTimeMillis();
System.out.println("StringBuffer = " + (stop-start) + "ms.");
```

What are the differences between typecasting and data conversion?

Typecasting is a feature of the language whereby you are telling the compiler that you explicitly want to treat the object as if it were really of the type that you specified rather than its actual type.

Data conversion is physically changing one chunk of data into another, often with interpretation.

The difference can be illustrated thus:

Typecasting:

```
Object o = new String("1234");
String myString = (String)o;
```

This allows you to cast the type of object o to its proper type, String.

Data conversion:

```
String o = new String("1234");
Integer i = Integer.parseInt(o);
```

This converts the String "1234" into the Integer which represents the value 1234 (one thousand two hundred twenty four).

Why can a `return` or `throw` statement inside a finally block override a thrown exception or a return statement executed in the associated try or catch blocks?

By definition of the try/catch/finally statement the finally block is executed no matter what happens in the try/catch blocks.

Next, also by definition, both the return and throw statements complete abruptly. Lastly, by definition, abrupt completion of a finally block causes the try/catch/finally statement to complete abruptly.

Therefore, executing a return or throw in the finally clause abruptly completes the finally clause which, in turn, abruptly completes the try statement which therefore has the effect of masking/overriding any previous return or throw statement executed in the associated try/catch blocks.

So, just take a look at the try/catch/finally statements and check to see that you are actually handling exceptions and returns the way that you expected.

How can I prohibit methods from being overridden?

You can prevent a method from being overridden by declaring it `final`.

Why does Java not support global variables?

The concept of global variables has not been introduced in Java because the designers of the language wanted a more pure object-oriented language. Global variables introduce a number of problems such as having more layers of scoping and life-cycle timing issues.

Unfortunately, although Java does not have globals, it does have something similar to global variables, called statics (or class variables). Statics avoid the namespace collision problem, since they are tied to a particular class name, but still have the other problems of globals.

Should I declare the constants in my interface as `public final static` and my methods as `public abstract`?

The use of these modifiers in interfaces is completely optional because of the following reasons:

- All interface members are implicitly `public`.
- Every field declaration in the body of an interface is implicitly `public`, `static`, and `final`.
- Every method declaration in the body of an interface is implicitly `public` as well as `abstract`.

What does it mean to override a method?

If a subclass shares a method name with identical arguments and return type, it overrides the method of its superclass. Overridden methods may differ in exceptions thrown by throwing a more specific exception type or none at all.

Overriding methods are instance (non-static) methods that match a non-private (i.e. inherited) and non-final (i.e. overridable) instance method in the superclass. The names of the methods, the type and order of their parameter lists and their return types must match exactly. The overriding method may not declare any checked exceptions that are not declared by the overridden method but may declare fewer exceptions, more specific exceptions (i.e. an exception that is a subclass of an exception declared by the overridden method) and any unchecked (i.e. Error or RuntimeException) exceptions. The overriding method may not have an access modifier that is more restrictive than the access modifier of the overridden method.

Can a Java class be static?

Yes and no. The static modifier is only applicable to so-called member classes – that is, classes which are directly enclosed within another class.
In order to understand the use of the static keyword in class declaration, we need to understand the class declaration itself. You can declare two kinds of classes: top-level classes and inner classes.

Top-level classes
A Top-level class is declared at the top level as a member of a package. Each top-level class corresponds to its own java file sporting the same name as the class name.

A Top-level class is by definition already top-level, so there is no point in declaring it static; it is an error to do so. The compiler will detect and report this error.
Inner classes

You define an inner class within a top-level class. Depending on how it is defined, an inner class can be one of the following four types:

1) **Anonymous**. Anonymous classes are declared and instantiated within the same statement. They do not have names, and they can be instantiated only once.
The following is an example of an anonymous class:

```
okButton.addActionListener( new ActionListener()
{
   public void actionPerformed(ActionEvent e)
   {
       dispose();
   }
});
```

Because an anonymous class does not have a normal class declaration where it is possible to use static, it cannot be declared static.

2) **Local**. Local classes are the same as local variables, in the sense that they are created and used inside a block. Once a class is declared within a block, it can be instantiated as many times as you wish within that block. Like local variables, local classes are not allowed to be declared `public`, `protected`, `private`, or `static`.

Here's a code example:

```
   //some code block.......
  {
   class ListListener implements ItemListener
   {
      List list;
      public ListListener(List l)
      {
          list = l;
      }
```

```
        public void itemStateChanged(ItemEvent e)
        {
            String s = l.getItemSelected();
            doSomething(s);
        }
    }
    List list1 = new List();
    list list2 = new List();
    list1.addItemListener(new ListListener(list1));
    list2.addItemListener(new ListListener(list2));
}
```

3) **Member**. Member classes are defined within the body of a class. You can use member classes anywhere within the body of the containing class. You declare member classes when you want to use variables and methods of the containing class without explicit delegation.

The member class is the only class that you can declare `static`. When you declare a member class, you can instantiate that member class only within the context of an object of the outer class in which this member class is declared. If you want to remove this restriction, you declare the member class a `static` class.

When you declare a member class with a `static` modifier, it becomes a nested top-level class and can be used as a normal top-level class as explained above.

4) **Nested top-level**. A nested top-level class is a member classes with a `static` modifier. A nested top-level class is just like any other top-level class except that it is declared within another class or interface. Nested top-level classes are typically used as a convenient way to group related classes without creating a new package. If your main class has a few smaller helper classes that can be used outside the class and make sense only with your main class, it is a good idea to make them nested top-level classes. To use the nested top-level class, write: `TopLevelClass.NestedClass`

How are Signed number represented?

So far, we have treated binary numbers as unsigned values. The binary number …00000 represents zero, …00001 represents one, …00010 represents two, and so on toward infinity. What about negative numbers? In mathematics, negative numbers in any base are represented in the usual way, by prefixing them with a "−" sign. However, on a computer, there are various ways of representing a number's sign. One of the approaches to this problem is representing a number's sign by allocating one sign bit to represent the sign: set that bit (often the Most Significant Bit) to 0 for a positive number, and set to 1 for a negative number. The remaining bits in the number indicate the magnitude (or absolute value). Hence in a byte with only 7 bits (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127). Thus, you can represent numbers from −12710 to +12710. A consequence of this representation is that there are two ways to represent 0, 00000000 (0) and 10000000 (−0). Decimal −43 encoded in an eight-bit byte this way is 10101011.

This approach is directly comparable to the common way of showing a sign (placing a "+" or "−" next to the number's magnitude).

What's the difference between the ">>" and ">>>" operators?

">>" is a signed right shift, while ">>>" is an unsigned right shift. What this means is that if you use a signed shift on a negative number, the result will still be negative. Signed right shifting by one is equivalent to dividing by two, even if the number is negative.

Unsigned shifting, on the other hand, ignores the sign of the number, and unsigned right shifting a negative number will result in a positive number -- which is why you generally should not use the unsigned shift unless you are considering your number as unsigned.

The reason for the difference is that bytes, shorts, ints, and longs are all stored in two's-complement form (i.e. as signed numbers). This means, among other things, that the most significant bit is the sign bit, so that if the sign bit is 0 the number is positive, and if the sign bit is 1 the number is negative. If you want to divide a signed number by two by shifting it right, you have to retain the sign bit.

Another way of thinking about the difference between the operators is that the signed shift fills from the left with the sign bit (0 or 1), and the unsigned shift zero-fills from the left.

If you are treating a number as unsigned, then you should use the unsigned shift operator. Otherwise you should use the signed shift operator, which retains the sign bit.

Can a `private` variable be accessed by other instances of the same class?

Yes. The `private` keyword restricts which classes can access a variable or method, not which instances. Any instance of a class can access the private instance variables of any other instance.

If a `continue` statement is used within a for loop, is the test or iteration statement executed next?

The third statement in the `for` loop, or the iteration statement is executed after a `continue` statement is used within a `for` loop. After the iteration statement is executed, the test condition is executed to see if the loop body should be repeated.

If the same exception can be thrown from multiple places, does it make sense to create one instance and throw it from both?

When you create the `Throwable/Exception` instance, stack trace information is placed into the object. This is used to report the origin of the exception for the stack dump. If you create the exception at a place other than where the exception actually occurred, the stack dump will actually be incorrect. It is best to combine the creation of the exception object with the use of the throw keyword to make sure the information is reported correctly:

```
throw new Exception("Please enter a positive non-zero value.");
```

Is there a way for a class in a package to access a class that is not in a package (i.e., those that are in the unnamed, default package)?

Yes, you can do that by importing the class:

```
package MyPackage;
import RootClass;
public class MyClass
{
      public static void main(String[] argv)
      {
            RootClass obj = new RootClass();
            System.out.println(obj.toString());
      }
}
```

Just make sure that the class with no package is a public class so classes outside of the empty package can access it.

What are the differences between PATH and CLASSPATH environment variables?

The `PATH` environment variable is typically something that the operating system uses to find executable files.

The `CLASSPATH` environment variable is typically something that implementations of the Java Virtual Machine (JVM) use to find Java class files. The `CLASSPATH` can be changed by setting the `CLASSPATH` in the command prompt or in the application which is to be executed.

For example, assume that Java is installed in `C:\jdk1.5.0\` and your code is in `C:\Sample\example.java`, then your `PATH` and `CLASSPATH` should be set to:

```
PATH=C:\jdk1.5.0\bin;%PATH%
CLASSPATH=C:\Sample
```

In short,

`PATH` = only for executables
`CLASSPATH` = for directories, JAR files & ZIP files.

What is the difference between a Java compiler and a Java interpreter?

Typically, when used in that generic manner, the term Java compiler refers to a program which translates Java language source code into the Java Virtual Machine (JVM) bytecodes. The term Java interpreter refers to a program which implements the JVM specification and actually executes the bytecodes (and thereby running your program).

How do I create a constructor for an anonymous class?

Since anonymous classes don't have names, you cannot create a constructor by just using the name of the class. What you can do instead is to use an "instance initializer". Instance initializers basically look like methods with no names or return values. The code in bold is an instance initializer:

```
ActionListener listener = new ActionListener {
  {
    System.out.println("Instance Initializer");
 }
  public void actionPerformed(ActionEvent event) {
    ...
  }
};
```

When I create anonymous classes, what access modifier do they get?

Anonymous classes get the default, unnamed access modifier. You cannot make them `private`, `protected`, `public`, or `static`.

Can an anonymous class have static members?

With one exception, anonymous classes cannot contain static fields, methods, or classes. The only static things they can contain are `static final` constants.

What are the differences between instance and class variables?

Instance variables have separate values for each instance of a class. Class variables maintain a single shared value for all instances of the class, even if no instance object of that class exists.

You would use the static keyword to change an instance variable into a class variable.

Both instance and class variables are declared at the class level, not within methods:

```
public class Foo
{
  static private int count; // class variable
  String name; // instance variable
  private void Bar()
  {
      int halfCount; // local variable
  }
}
```

When should I use an interface instead of an abstract class?

In object-oriented languages, like Java, "Inheritance" can be implemented in two ways. "Implementation" inheritance is where the sub-class inherits the actual code implementation from the parent. "Interface" inheritance is where the "sub-class" adheres to the public interface of the "parent".

Java actually mixes the two notions together a bit. Java interfaces are good to use, as when you "implement" an interface, you are stipulating that your class adheres to the "contract" of the interface that you specified. Whereas, Java class inheritance isn't as good, since when you sub-class in Java you are getting both the code inheritance but you are also stipulating that your subclass adheres to the "contract" of the interface of the parent class.

abstract classes in Java are just like regular Java classes but with the added constraint that you cannot instantiate them directly. In terms of that added constraint, they are basically classes which do not actually implement all of the code specified by their "contract".

So, it is generally considered good OO practice to specify the "contract" which you want to adhere to via Java interfaces. Then use normal Java class inheritance primarily for code reuse purposes. Use abstract Java classes when you want to provide some standard base code but want or need to force the users of your class to complete the implementation that is you create a skeleton implementation and the sub-classes must flesh it out.

Are `try` blocks expensive in time and space?

Exceptions are essential to writing robust code and, fortunately, are extremely cheap until you actually throw an exception; the try costs you very little (usually the cost of marking the stack pointer). Exceptions are for exceptional behavior and so an expensive throw operation is usually ok as they are not executed very often. Do not use exceptions for control-flow in your program.

On the other hand, one legitimate use of exceptions for control-flow is backtracking algorithms in very large complicated programs. They do not affect parsing speed much when backtracking over large input streams if I'm smart about it (only backtracking when totally necessary).

What are Order of precedence and Associativity, and how are they used?

Order of precedence determines the order in which operators are evaluated in expressions. Associativity determines whether an expression is evaluated left-to-right or right-to-left.

How do you set class path for class files located in several locations?

If you have your `.class` files located in different locations on your system, include the path of each location in the `CLASSPATH` environment variable. The following table lists two different `.class` files located in different locations on a system.

.class File Name	Path of .class file
Welcome.class	C:\Source Code
Pyramid.class	D:\NetBeans

To set the `CLASSPATH` environment variable to execute `Welcome.class` and `Pyramid.class`, you will have to set the `CLASSPATH` environment variable in the following manner:

```
CLASSPATH = %CLASSPATH%;C:\Source Code;D:\NetBeans;
```

From where can you download NetBeans IDE?

The various versions NetBeans IDE can be downloaded from the following URL:

`http://www.netbeans.org`

--- End of FAQ ---