

What can it be used for?

- Static variables can be used as data sharing amongst objects of the same class. For example to implement a counter that stores the number of objects created at a given time can be defined as so:

```
public AClass
{
    static private int counter;
    ...
    public AClass()
    {
        ...
        counter += 1;
    }
    ...
    public int getNumberOfObjectsCreated()
    {
        return counter;
    }
}
```

The counter variable is incremented each time an object is created.

Public static variable should not be used, as these become GLOBAL variables that can be accessed from everywhere in the program. Global constants can be used, however. See below:

```
static public final String CONSTANT_VAR = "Const";
```

- Static methods can be used for utility functions or for functions that do not belong to any particular object. For example:

```
public Match
{
    ...
    public static addTwoNumbers( int par1, int par2 )
    {
        return par1 + par2;
    }
}
```

Danger of static variables

Use static variables only for:

- data sharing (be careful)
- defining global constants

Use static methods for:

- utility functions

Using static variables and/or method for other purposes goes against object orientation, and will make your code either harder to read or maintain.

External links

- [Static, interactive Java Lesson](#) ^[32]

Destroying Objects

Navigate **Classes and Objects** topic:

- Defining classes
- Creating objects
- Interfaces
- Using static members
- Destroying objects
- Overloading methods and constructors
- Class Loading

Unlike in many other object-oriented programming languages, Java performs automatic garbage collection - any unreferenced objects are automatically erased from memory - and prohibits the user from manually destroying objects.

finalize()

When an object is garbage-collected, the programmer may want to manually perform cleanup, such as closing any open input/output streams. To accomplish this, the `finalize()` method is used. Note that `finalize()` should never be manually called, except to call a super class' `finalize` method from a derived class' `finalize` method. Also, we can not rely on when the `finalize()` method will be called. If the java application exits before the object is garbage-collected, the `finalize()` method may never be called.

```
protected void finalize() throws Throwable
{
    try {
        doCleanup();           // Perform some cleanup.  If it fails for
some reason, it is ignored.
    } finally {
        super.finalize(); //Call finalize on the parent object
    }
}
```

The garbage-collector thread runs in a lower priority than the other threads. If the application creates objects faster than the garbage-collector can claim back memory, the program can run out of memory.

The `finalize` method is required only if there are resources beyond the direct control of the Java Virtual Machine that need to be cleaned up. In particular, there is no need to explicitly close an `OutputStream`, since the `OutputStream` will close itself when it gets finalized. Instead, the `finalize` method is used to release either native or remote resources controlled by the class.

Overloading Methods and Constructors

Navigate **Classes and Objects** topic:

- Defining classes
- Creating objects
- Interfaces
- Using static members
- Destroying objects
- Overloading methods and constructors
- Class Loading

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be overloaded. This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the throws clauses of two methods with the same name but different signatures.

Methods are overridden on a signature-by-signature basis.

If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method. In this respect, the Java programming language differs from C++.

When a method is invoked, the number of actual arguments and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked. If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup.

Arrays

Navigate **Language Fundamentals** topic:

Intro to Arrays

An **array** is similar to a table of data, keyed by number. In Java an array is an object like all other objects. Look at the following program:

```
import java.util.*;

public class ArrayExample {
    static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int numNames = getInt("Number of names?");
        String[] names = new String[numNames];
        for (int i = 0; i < names.length; i++) {
            names[i] = getString("Enter name #" + i);
        }
        for (int i = 0; i < names.length; ++i) {
            System.out.println(names[i]);
        }
    }
}
```