

```
{  
}  
}
```

The Constructor is called automatically when an object is created with the **new** keyword. A constructor may also be called from an other constructor, see below:

```
public class MyClass  
{  
    private String memberVar;  
    /**  
     * MyClass Constructor, there is no input parameter  
     */  
    public MyClass()  
    {  
        MyClass("Default Value");  
    }  
    /**  
     * MyClass Constructor, there is one input parameter  
     */  
    public MyClass( String param1 )  
    {  
        memberVar = param1;  
        ...  
    }  
}
```

In the above example, the constructor with no input parameter calls the other constructor with the default initial value. This gives an option to the user, to create the object with the default value or create the object with a specified value.

Static Method

We defined method above as an operation on an object. Static Methods are defined inside a class, but they are not an operation on an object. No object needs to be created to execute a Static Method, they are simply global functions, with input parameters and a return value.

```
public class MyObject  
{  
    static public String myStaticMethod()  
    {  
        ...  
        return("I am a Static Method");  
    }  
}
```

Static Methods can be referenced anywhere prefixed by the class name. See below:

```
{  
...  
}
```

```
// --- Call myStaticMethod ---
System.out.println( "Output from the myStaticMethod:" +
MyObject.myStaticMethod() );
...
}
```

You can write a non object oriented program by using only Static Methods in java. Because java evolved from the C programming language, Static Method is a left over from a non object oriented language.

You write Static Method the same way as normal method, the only difference is that you can not reference any member variables and any object methods. Static Methods can reference only Static variables and call only other Static Methods. However you can create an object and use it inside a Static Method.

```
public class MyObject
{
    public String memberVar;

    static private String memberStaticVar;

    static public String myStaticMethod()
    {
        memberVar = "Value"; --> ERROR Cannot reference member var.

        memberStaticVar = "Value"; // --- This is okay, static vars. can
be used

        // --- Create an object ---
        MyObject obj = new MyObject();
        obj.memberVar = "Value"; // --- This is okay since an object is
created --
        ...
        return("I am a Static Method");
    }
}
```

External links

- Abstract methods, interactive Java Lesson ^[25]

Primitive Types

Navigate **Language Fundamentals** topic:

Types in Java define the allowed data values and the operations that may be performed on those values. Each expression, field, method parameter, local variable, and method return values, have an associated type. The syntax template for a field declaration in Java is

`[Modifiers] Type FieldName [= Expression];`

Method, method parameter, and local variables have similar syntax: optional modifiers, a required type, and the name of the value being declared.

In java, there are three different kind of *Types*, those are:

- Primitive Type
- Object Reference Type
- An array type

About Java Types

Java is **strongly typed** in that all expressions and declarations have types (either declared or inferred) and the language only allows programs which adhere to the type constraints. You cannot write a statement or expression in Java which directly invokes an operation on a value that is not allowed by that value's type, nor can you perform assignments that violate the type system: you cannot assign a String value to an integer typed variable. This helps achieve a high level of type safety in Java. Most type errors are caught and detected by the Java compiler. However, some type errors can still occur at runtime because Java supports a cast operation which is a way of changing the type of one expression to another. However, Java performs run time type checking when doing such casts, so an incorrect type cast will cause a runtime exception rather than succeeding silently and allowing data corruption.

Java is also known as a **hybrid language**. While supporting object oriented (OO) programming, Java is not a pure OO language like Smalltalk or Ruby. Instead, Java offers both object types and primitive types. Primitive types are used for boolean, character, and numeric values and operations. This allows relatively good performance when manipulating numeric data, at the expense of flexibility. For example, you cannot subclass the primitive types and add new operations to them.

Examples of Types

Below are two examples of Java types and a brief description of the allowed values and operations for these types. Additional details on each are available in other modules.

Example: int

The primitive type **int** represents a signed 32 bit integer value. The allowed data values for int are the integers between -2147483648 to 2147483647 inclusive.

The set of operations that may be performed on int values includes integer arithmetic such as +, -, *, /, %, comparison operations (==, !=, <, >, <=, >=), assignments (=, ++, --, +=, -=), bit-wise operations such as logical and, logical or, logical xor, negation (&, |, ^, ~), bit shift operations (<<, >>, >>>), conversions to other numeric types and promotion to other integer types.

For example, to declare a **private** integer instance field named length, one would use the declaration

```
private int length;
```

Example: String

You use **class** and **interface** definition in Java to define new types. Class and interface types are associated with object references also sometime referred to as *Reference types*. An object reference has two main attributes:

- Its **type** associated with a class or an interface
- A java object it references, that is created by instantiating a class

The String class is one such example. String values are a sequence of 0 or more Unicode characters. The **null** reference is another valid value for a String expression.

The operations on a String reference variable are those available for all reference types, such as comparison operations ==, != and assignment =.