Method Calls in Java

First, some information about the Stack, local variables & parameters

• The Stack keeps the state of all active methods (those that have been invoked but have not yet completed)

• When a method is called a new stack frame for it is added to the top of the stack, this stack frame is where space for the method’s local variables and parameters are allocated

• When a method returns, its stack frame is removed from the top of the stack (space for its local vars and params is de-allocated)

• Space for local variables and parameters exist only while the method is active (it has a stack frame on the Stack)

• local variables and parameters are only in scope when they are in the top stack frame (when this method’s code is being executed)
An Example

```java
public class TestMethodCalls {
    public static void main(String[] args) {
        Foo f1, f2;
        int x=8;
        f1 = new Foo(10);
        f2 = new Foo(12);
        f1.setVal(x);
        x = f1.add(f2, x);
    }
}

public class Foo {
    private int x;
    public Foo(int val) { x = val; }
    public void setVal(int val) { x = val; }
    public int getVal() { return x; }
    public int plus(Foo f, int val) {
        int result;
        result = f.getVal() + x + val;
        return result;
    }
}
```
We start executing code in main:
  • there is a single stack frame containing main’s local variables

```java
class TestMethodCalls {
  public static void main(String[] args) {
    Foo f1, f2;
    int x = 8;
    f1 = new Foo(10);
    f2 = new Foo(12);
    f1.setVal(x);
    x = f1.add(f2, x);
  }
}
```

Only through variables `f1` and `f2` can `main` access the objects’ public members.
When main calls f1’s setVal method a new stack frame is added that holds setVal’s parameters and local variables.

```
public class TestMethodCalls {
    public static void main( ...) {
        ...
        f1.setVal(x);
    }
}
```

```
public class Foo {
    private int x;
    public void setVal(int val){
        x = val;
    }
}
```

Parameter `val` gets its value from its argument (the value of `x` in main)
Implicitly, a reference to the object referred to by f1 is passed as well:
- `setVal` is called from within this object, so its members are in scope as well as all parameters and local variables of `setVal`.

```java
public class TestMethodCalls {
    public static void main( ... ) {
        ...
        f1.setVal(x);
    }
}

class Foo {
    private int x;
    public void setVal(int val){
        // set data member x’s value
        x = val;
    }
    public add(Foo f, int x) { ... }  
```

All methods and data members of my object can be accessed from within `setVal` too.
When main calls add, we are passing the value of object ref f2:
- add’s parameter f refers to the same object as f2 does

```java
public class TestMethodCalls {
    public static void main( ...) {
        ...          
        x = f1.add(f2, x);
    }
}
```

```java
public class Foo {
    private int x;
    public int plus(Foo f, int val) {
        int result;
        result = f.getVal() + x + val;
        return result;
    }
}
```
If a method changes the value of a parameter, it does not change the argument’s value:

```java
public class TestMethodCalls {
    public static void main( ... ) {
        ... 
        x = f1.add(f2, x);
    }
}
```

```java
public class Foo {
    private int x;
    public int plus(Foo f, int val) {
        int result = 0;
        f = new Foo(6);
        val = 20;
        return result;
    }
}
```