# Some important aspects

# Agenda

- Scope of variables
- Method Calls
- Wrappers in Java
- Memory Management

# Scope of variables

*The* **scope** *of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name*

*The* **scope of a local variable declaration** *in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement*

*The* **scope of a parameter** *of a method or constructor is the entire body of the method or constructor*

# Scope of variables

```
public class Test {
    public static void display() {
        System.out.println( "a = " + a );
    }
    public static void main( String[] args ) {
        int a;
        a = 9; // valid access, within the same block
        if (a < 10) {
            a = a + 1; // another valid access
        }
        display();
    }
}
```

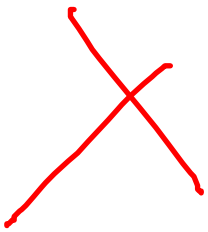
**Is this a valid program?**

# Scope of variables

```
public class Test {
    public static void main( String[] args ) {
        System.out.println( sum );
        for ( int i=1; i<10; i++ ) {
            System.out.println( i );
        }
        int sum = 0;
        for ( int i=1; i<10; i++ ) {
            sum += i;
        }
    }
}
```

not defined

**Is this a valid program?**



# Method calls

```
public class Test {

  public static void increment( int a ) {
    a = a + 1;
  }

  public static void main( String[] args ) {
    int a = 5;
    System.out.println( "before: " + a );
    increment( a );
    System.out.println( "after: " + a );
  }
}
```

What will printed?

# Method calls

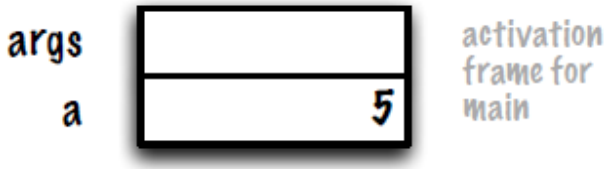
```
public class Test {

  public static void increment( int a ) {
    a = a + 1;
  }

  public static void main( String[] args ) {
    int a = 5;
    System.out.println( "before: " + a );
    increment( a );
    System.out.println( "after: " + a );
  }
}
```

What will printed?

```
before: 5
after: 5
```

# Method calls

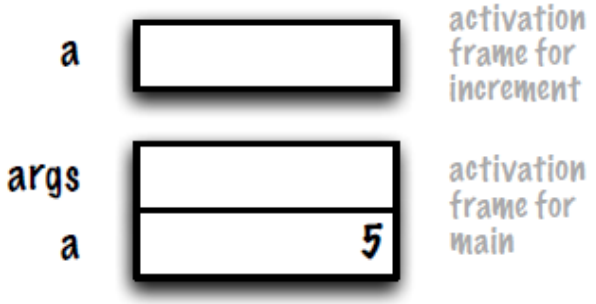
```
   public static void increment( int a ) {
     a = a + 1;
   }
   public static void main( String[] args ) {
>      int a = 5;
       increment( a );
   }
```




Each **method call** has its own **activation frame**, which holds the parameters and local variables (here, **args** and **a**)

# Method calls

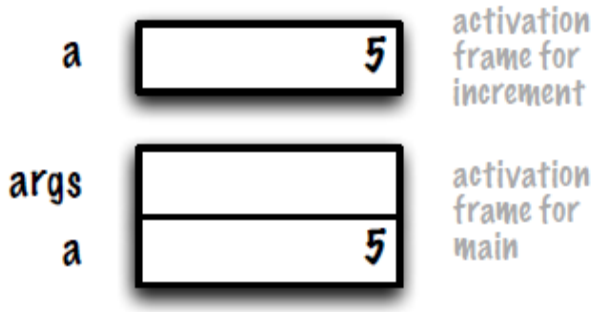
```
  public static void increment( int a ) {
    a = a + 1;
  }
  public static void main( String[] args ) {
      int a = 5;
>     increment( a );
  }
```




When **increment** is called a new activation frame is created

# Method calls

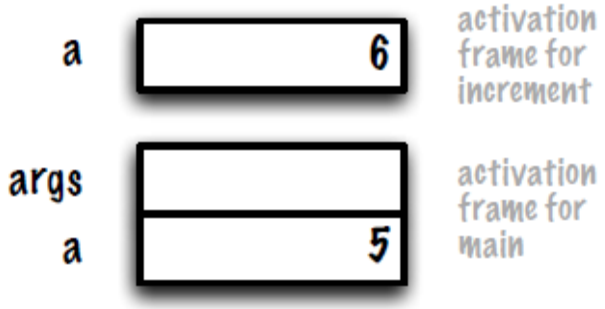
```
  public static void increment( int a ) {
    a = a + 1;
  }
  public static void main( String[] args ) {
      int a = 5;
>     increment( a );
  }
```




The value of the **actual parameter** is copied to the location of the **formal parameter**

# Method calls

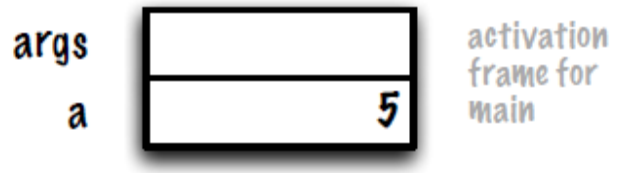
```
   public static void increment( int a ) {
>      a = a + 1;
   }
   public static void main( String[] args ) {
       int a = 5;
       increment( a );
   }
```




The execution of **a = a + 1** changes the content of the formal parameter **a**, which is a distinct memory location the local variable **a** of the **main** method

# Method calls

```
   public static void increment( int a ) {
       a = a + 1;
   }
   public static void main( String[] args ) {
       int a = 5;
       increment( a );
> }
```




Control returns to the main method, the activation frame for **increment** is destroyed

# Method calls

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.
**When a method is called**:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed
- a return value or (void) is saved
- (the activation frame is destroyed)
- the execution of the calling method restarts with the next instruction

# Method calls

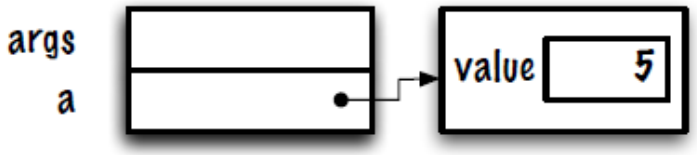
```
class MyInteger {
    int value;
    MyInteger( int v ) {
        value = v;
    }
}
class Test {
    public static void increment( MyInteger a ) {
        a.value++;
    }
    public static void main( String[] args ) {
        MyInteger a = new MyInteger (5);
        System.out.println("before: " + a.value);
        increment(a);
        System.out.println("after: " + a.value);
    }
}
```

**What will be printed out?**

before : 5
after : 6

# Method calls

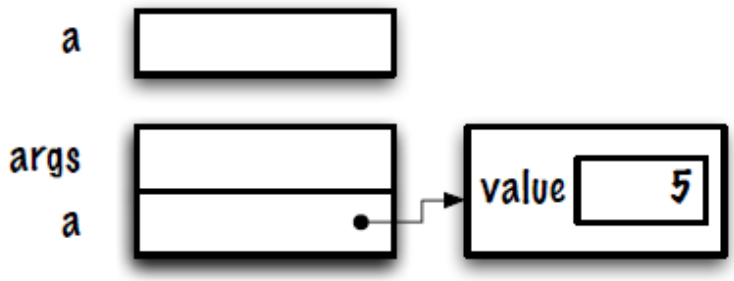
```
 static void increment( MyInteger a ) {
     a.value++;
 }
 public static void main( String[] args ) {
>    MyInteger a = new MyInteger( 5 );
     increment( a );
 }
```




The local variable **a** of the **main** method is a reference to an instance of the class **MyInteger**

# Method calls

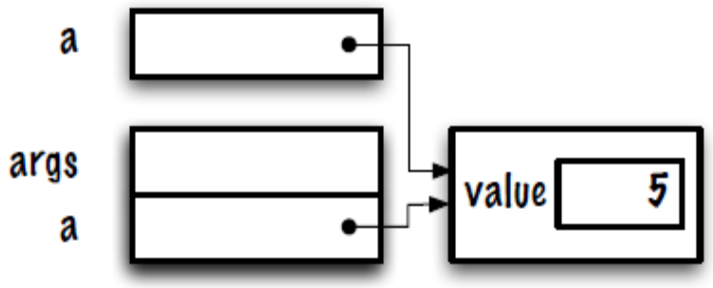
```
  static void increment( MyInteger a ) {
      a.value++;
  }
  public static void main( String[] args ) {
      MyInteger a = new MyInteger( 5 );
>     increment( a );
  }
```




Calling **increment**, creating a new activation frame

# Method calls

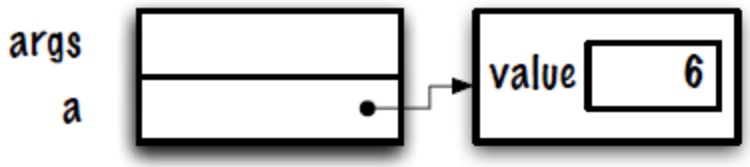
```
  static void increment( MyInteger a ) {
      a.value++;
  }
  public static void main( String[] args ) {
      MyInteger a = new MyInteger( 5 );
>     increment( a );
  }
```




Copying the **value** of the **actual parameter** into the **formal parameter** of **increment**

# Method calls

```
static void increment( MyInteger a ) {
    a.value++;
}
public static void main( String[] args ) {
    MyInteger a = new MyInteger( 5 );
    increment( a );
>}
```




Returning the control to the **main** method

# Wrapper classes

Suppose we created the following class:

```
class Integer {
    int value;
}
```

Usage:

```
Integer a;
a = new Integer();
a.value = 33;
a.value++;
System.out.println( "a.value = " + a.value );
```

We use the dot notation to access the value of an instance variable. Java has a pre-defined class named **Integer**. It is called a "wrapper" class; it wraps an **int** value into an object.

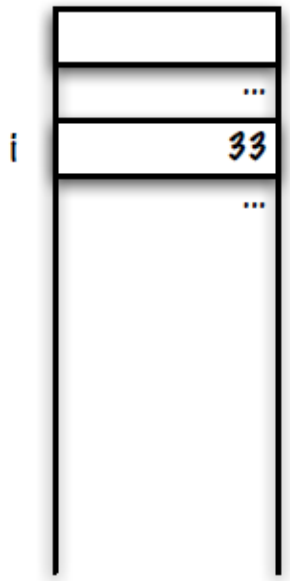
Adding a constructor

```
class Integer {
    int value;
    Integer( int v ) {
        value = v;
    }
}
```

Usage:

```
Integer a;
a = new Integer( 33 );
```

# Primitive vs reference variables

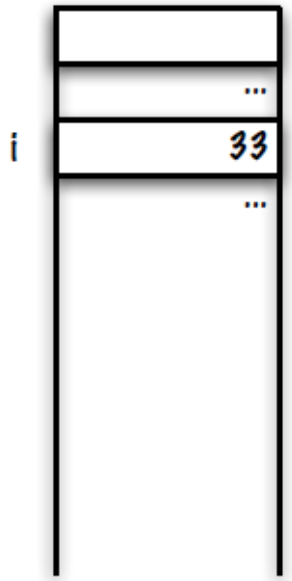



```
int i = 33;
```

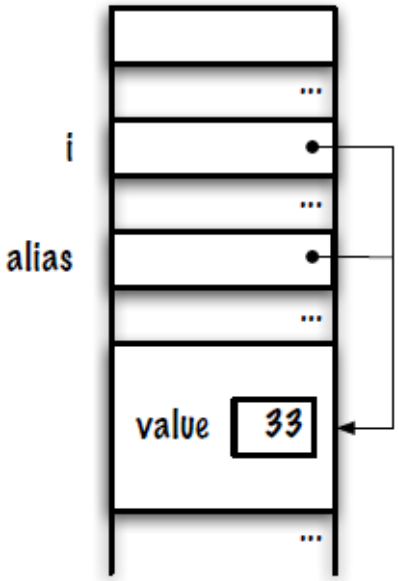

```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

At compile time the necessary memory to hold the reference is allocated

# Primitive vs reference variables




```
int i = 33;
```

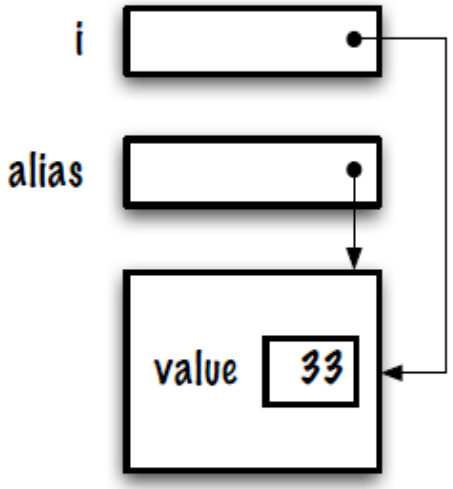



```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

**i** and **alias** are both designating the same object!

# Primitive vs reference variables


```
int i = 33;
```




```
Integer i, alias;
i = new Integer( 33 );
alias = i;
```

Using the memory diagram representation

# Wrapper classes

- For every **primitive type** there is an associated **wrapper class**
- For instance, **Integer** is the wrapper class for the primitive type **int**
- A wrapper stores a value of a primitive type inside an object
- This will be paramount for stacks, queues, lists and trees
- Besides holding a value, the wrapper classes possess several class methods, mainly to convert values from/to other types, e.g. **Integer.parseInt( "33" )**

# Auto-boxing

Java 5 and 6 **automagically** transform the following statement

```
Integer i = 1;
```

into

```
Integer i = new Integer( 1 );
```

This is called **auto-boxing**.

# Auto-unboxing

Similarly, the statement **i = i + 5**

```
Integer i = 1;
i = i + 5;
```

is transformed into

```
i = new Integer( i.intValue() + 5 );
```

where the value of the wrapper object designated by **i** is extracted, **unboxed**, with the method call, **i.intValue()**.

# Boxing/unboxing

| Primitive | Reference |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| bool | Boolean |
| char | Character |

All 8 primitive types have a corresponding **wrapper** class. The automatic conversion from primitive to reference type is called **boxing**, and the conversion from reference to primitive type is called **unboxing**.

# Does it matter?

```
long s1 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s1 = s1 + (long) 1;
}
```

**49 milliseconds**

```
Long s2 = (long) 0;
for ( j=0; j<10000000; j++ ) {
    s2 = s2 + (long) 1;
}
```

**340 milliseconds**

- Why?

On the right side, **s2** is declared as a **Long**, hence, the line,

```
s2 = s2 + (long) 1;
```

is rewritten as,

```
s2 = new Long( s2.longValue() + (long) 1 );
```

# Programming tip: benchmarking your code

```
long start, stop, elapsed;

start = System.currentTimeMillis(); // start the clock

for ( j=0; j<10000000; j++ ) {
    s2 += (long) 1; // stands for 's2 = s2 + (long) 1'
}

stop = System.currentTimeMillis(); // stop the clock

elapsed = stop - start;
```

where **System.currentTimeMillis()** returns the number of milliseconds elapsed since midnight, January 1, 1970 UTC (Coordinated Universal Time).
**System.nanoTime()** also exists.

# Memory management

What happens to objects when they are not referenced? Here what happens to the object that contains the value 99?

```
MyInteger a = new MyInteger( 7 );
MyInteger b = new MyInteger( 99 );
b = a;
```

- The JVM recuperates the memory space
- This process is called **garbage collection**
- Not all programming languages manage memory automatically

Java is not immune to memory leaks as will see in a few weeks...