

# CMSC 131: Chapter 8 (Supplement)

## More about Methods

### Methods Revisited

#### Review:

- **Methods** are Java's **basic computational units**. Methods are also called functions or procedures.
- Information can be passed into a method through its parameters. The calling process provides the **actual parameters** (sometimes called **arguments**), and these are copied to the **formal parameters** (sometimes simply called **parameters**).
- Parameters are **passed by value**. Modifying a formal parameter does not alter the value of the corresponding actual parameter.
- By default a method (non-static) is associated with a particular class instance. A **static method is shared** by all instances of a class.
- A method can **return** a single value, a **primitive type** or an **object reference**.

### Method Syntax

(Omitted - See the textbook)

### Local Variables and Scope

**Local variable:** is a variable declared within a method.

- a local variable is **only accessible within the method** in which it is declared.
- **formal parameters** are considered to be local variables.
- if a variable with the same name is defined within another method, it is an **entirely different variable**.
- **global variables(?)**: In **Java** every variable is either **local** or is an **instance variable**.

## Local Variables and Scope

**Block:** is a collection of statements enclosed in curly braces {...}.

**Scope:** of a variable means the portion of the program where a variable can be accessed.

```
public class LocalTest0 {
    public static void dumb( int y ) {
        do {
            double z = Math.random( );
            System.out.println( "z = " + z );
        } while ( --y > 0 );
        // ERROR: z cannot be resolved
        System.out.println( "In dumb: z = " + z );
    }
}
```

### Example: Test of Local Variables

**Duplicate Variables:** Java does not allow two local variables to have the same name.

```
public class LocalTest1 {
    public static void dumber( int y ) {
        int y; // ERROR: Duplicate variable y
        double z;
        do {
            double z = Math.random( ); // ERROR: Duplicate variable z
            System.out.println( "z = " + z );
        } while ( --y > 0 );
        System.out.println( "In dumber: z = " + z );
    }
}
```

### Example: Test of Local Variables

```
public class LocalTest {
    public static void smarter( int y ) {
        double z;
        do {
            z = Math.random( );
            System.out.println( "z = " + z );
        } while ( --y > 0 );
        System.out.println( "In smarter: z = " + z + " y = " + y );
    }
}

public class LocalTestDriver {
    public static void main( String[ ] args ) {
        int z = 5;
        int y = -6;
        LocalTest.smarter( 3 );
        System.out.println( "Back in main: z = " + z + " y = " + y );
    }
}
```

## Test Drivers

The previous example showed the use of a **driver program**.

```
public class SomeClass {
    public static void method1( ) { ... }
    public static void method2( ) { ... }
}

public class TestDriver {
    public static void main( String[ ] args ) {
        SomeClass.method1( );
        SomeClass.method2( );
    }
}
```

## Initialization of Local Variables

Initialization of Variables:

**Instance variables:** Java provides default values automatically.

boolean variables:	false
numeric variables (int, float, etc):	0 (zero)
object references:	null

**Local variables:** are not initialized automatically. You need to give them an initial value before using them or the compiler will not compile your program.

## Method Overloading

**Overloading:** Java allows methods to have the **same name**, even within the same class.

```
public void setDate( int m, int d, int y ) { ... } // month given as integer
public void setDate( String m, int d, int y ) { ... } // month given as string
public void setDate( int m, int y ) { ... } // day defaults to 1
```

**Sample calls:**

```
Date dueDate = new Date( 10, 5, 2004 ); // set initial due date
dueDate.setDate( 10, 7, 2004 ); // delay the due date
dueDate.setDate( "Nov", 12, 2004 ); // delay it further
dueDate.setDate( 1, 2005 ); // delay until next year
```

**Question:** How does Java know which one to call?

**Answer:** It looks at the number and of types of arguments.

## Method Overloading and Signatures

**Overloading:** using the **same identifier name** for different methods.

**Signature:** of a method consists of the name of the method and the types of the parameters.

**Example:**

```
public float doSomething( int x, double z, double w, String s )
```

**Corresponding Signature:**

```
doSomething( int, double, double, String )
```

## Method Overloading and Signatures

Note that the **return type** of a method is **not** part of the signature.

**Example:**

```
public int toCelsius( double t ) { ... }  
public double toCelsius( double t ) { ... }  
...  
System.out.println( toCelsius( 98.6 ) );
```

**Which method should be called?** Unfortunately, Java cannot read your mind.

## Parameter Type Promotion

**Arithmetic Promotion:** We have seen that, in arithmetic expressions, Java promotes numeric types to the higher type:

```
double total = ... ;  
int count = ... ;  
double average = total / count;
```

**Promotion of Parameters:** Java automatically promotes each actual parameter to match the type of its formal parameter.

```
int area = 1024;  
double s = Math.sqrt( area );
```

## Ambiguous Overloading

Because of type promotion, there are times when Java cannot figure out which method to call.

```
public void fooBar( int x, double y ) { ... }
public void fooBar( double u, int v ) { ... }
...
fooBar( 10, 23.0 );      // okay, use the first
fooBar( 10.0, 23 );     // okay, use the second
fooBar( 10, 23 );      // ???
```

Do we promote 23 to 23.0 and call the first, or promote the 10 to 10.0 and call the second?

Java issues a **compile-time error**, since it cannot resolve the ambiguity.

## Class References as Parameters

**Pass by Value:** Recall that actual parameters (arguments) are passed to a method by **copying** their values to the formal parameters.

```
public void foo( ... ) {
    int x = 23;
    bar( x );
    System.out.println( x );      // this prints 23
}

public void bar( int x ) { x++; } // change the formal parameter
```

This is not as obvious when the actual parameter is an **object reference**.

## Class References as Parameters

To see why object references behave differently, let us create a toy class, called **RefTest**.

```
public class RefTest {
    private int data;                // instance data is a single integer

    public RefTest( int d ) { data = d; } // constructor
    public String toString( )           // convert to string "[ data ]"
        { return new String( "[" + data + "]" ); }
    public void changeMe( ) { data++; } // increment data
}
```

## Class References as Parameters

```
public class RefTestDriver {

    public static void main( String[] args ) {
        int x = 0;
        RefTest ref = new RefTest( 5 );
        System.out.println( "Before:  x = " + x +
                             " ref = " + ref );
        changeThem1( x, ref );
        System.out.println( "After-1: x = " + x +
                             " ref = " + ref );
        // ... (other stuff omitted)
    }

    public static void changeThem1( int x, RefTest ref ) {
        x = x+1;
        ref = new RefTest( 2 );
        System.out.println( "Inside-1: x = " + x +
                             " ref = " + ref );
    }
}
```

Output:

Before: x = 0 ref = [5]

Inside-1: x = 1 ref = [2]

After-1: x = 0 ref = [5]

## Class References as Parameters

```
public class RefTestDriver {  
  
    public static void main( String[] args ) {  
        int x = 0;  
        RefTest ref = new RefTest( 5 );  
        System.out.println( "Before:  x = " + x +  
                             " ref = " + ref );  
  
        // ... other stuff omitted  
        changeThem2( x, ref );  
        System.out.println( "After-2: x = " + x +  
                             " ref = " + ref );  
    }  
  
    public static void changeThem2( int x, RefTest ref ) {  
        ref.changeMe();  
        System.out.println( "Inside-2: x = " + x +  
                             " ref = " + ref );  
    }  
}
```

Output:

Before: x = 0 ref = [5]

Inside-2: x = 0 ref = [6]

After-2: x = 0 ref = [6]

## Returning to "return"

Recall that the return statement returns control from a method.

- It can appear **anywhere** in the method, but is best at the end.
- If the method has type **void**, then there is no return value given. **Example:**

```
public void printSecret( String s ) {  
    if ( s == null ) return;  
    System.out.println( "The secret of life is " + s );  
}
```

## Returning to "return"

Recall that the return statement returns control from a method.

- If the method has a return value, you must return a value of a compatible type. Numeric promotion is allowed, e.g., returning an `int` for a `double`.

```
public int thisBeBroken( double x ) {
    if ( x < 0 ) return x;    // ERROR! cannot return double as int
}                            // ERROR! if x >= 0, nothing is returned
```

- You can return an expression.

```
return w*x - 42*y + Math.sqrt( z );
```

## This is about "this"

The keyword `this` can be used within a class to generate an explicit reference to the current object.

**Example:** Let `Date` be a class with instance members, `month`, `day`, `year`.

```
public boolean equals( Date d ) {
    if ( ( year==d.year ) && ( month==d.month ) && ( day==d.day ) )
        return true;
    else
        return false;
}
```

We can replace the **implicit references** to `year`, `month`, and `day` with the **explicit references**: `this.year`, `this.month`, and `this.day`.

```
public boolean equals( Date d ) {
    if ( ( this.year==d.year ) && ( this.month==d.month ) && ( this.day==d.day ) )
        return true;
    else
        return false;
}
```

**Are you joking?** Why would anyone ever want to do this?

## This is about "this"

**Better Example:** Consider a class `Basic` that holds a single `int` data and has a static method that adds two such objects:

```
public class Basic {
    private int data;           // instance data
    public Basic( int d ) { data = d; } // constructor

    public static int add( Basic t1, Basic t2 ) // static add
    { return t1.data + t2.data; }
}
```

Your boss asks you to add a non-static method that adds the current object to another `Basic` object. We can do this by calling the `add` method, and passing ourselves as the argument.

```
public int addTo( Basic t ) { // nonstatic add
    return add( this, t );
}
```

These are called respectively as follows:

```
Basic.add( t1, t2 );           t1.addTo( t2 );
```

## Java File Structure

**Java program:** consists of

- one or more `.java` files

**Java file:** (e.g., `FooBar.java`) consists of:

- (optional) import statements
- one public class definition (named `FooBar`).

**Class definition:** consists of:

- (optional) instance variable declarations
- (optional) method declarations

These elements can appear in any order.

**Main method:** One file in the program should have a main method. This is where execution starts.