CMSC 131: Chapter 8 (Supplement)  
More about Methods

Static and non-Static Methods

Consider the following class method calls:

```java
String s = JOptionPane.showMessageDialog("Input an integer");
Date bastilleDay = new Date(7, 14, 1789);
int y = Integer.parseInt(s);
double c = Math.sqrt((double)y);
int x = s.length();
String t = bastilleDay.toString();
```

Note that these are of two basic types:

The last two apply to a particular object instances of a class:

```java
int x = s.length();
String t = bastilleDay.toString();
```

The others do not reference any particular object (just the class):

```java
String s = JOptionPane.showMessageDialog("Input an integer");
int y = Integer.parseInt(s);
double c = Math.sqrt((double)x);
```

Static and non-Static Methods

Static methods:

**Example:** A method for the Date class that determines whether a given year is a leap year.

The parameter will be an integer year, yr.

**Leap Year:** A year is a leap year if:
Leap Year Method Example

```java
public class Date {
    private int month;
    private int day;
    private int year;

    public Date(int m, int d, int y) { ... }
    public String toString() { ... }
    public boolean equals(Date d) { ... }

    // Is the given year a leap year?
    public static boolean isLeapYear(int yr) {
        boolean answer;
        if (yr % 400 == 0) answer = true;  // multiple of 400
        else if (yr % 100 == 0) answer = false;  // multiple of 100
        else if (yr % 4 == 0) answer = true;  // multiple of 4
        else answer = false;  // not a multiple of 4
        return answer;
    }
}
```

Error Checking in Constructor

Error Checking: Let us add to the constructor a check for days that are out of range and issue a warning message. To do this, we will add a new method lastDayOfMonth():

```java
/* Revised Date constructor with day check */
public Date(int m, int d, int y) {
    month = m;  day = d;  year = y;
    if (d < 1 || d > lastDayOfMonth(m, y))
        System.out.println( "Warning: day is out of range" );
}
```

lastDayOfMonth(int mo, int yr): This utility function returns the (int) last day of the given month.
Utility Method lastDayOfMonth

lastDayOfMonth( int mo, int yr): Returns the last day of the given month.

Design: "30 days hath September, April, June and November..."
Utility Method lastDayOfMonth

```java
public class Date {
    private int month;
    private int day;
    private int year;

    public Date(int m, int d, int y) { ... }
    public String toString() { ... }
    public boolean equals(Date d) { ... }
    public static boolean isLeapYear(int yr) { ... }

    private static int lastDayOfMonth(int mo, int yr) {
        int nDays;
        if (mo == 4 || mo == 6 || mo == 9 || mo == 11)
            nDays = 30;
        else if (mo == 2) {
            if (isLeapYear(yr)) nDays = 29;
            else nDays = 28;
        } else {
            nDays = 31;
        }
        return nDays;
    }
}
```

Static and non-Static Variables

Static variable:

For example, in our Date class we might add:

```java
public static final int DAYS_PER_WEEK = 7;
public static final int MONTHS_PER_YEAR = 12;
```

These could then be accessed outside the class as:

```java
Date.DAYS_PER_WEEK and Date.MONTHS_PER_YEAR
```

Memory Maps

Java splits memory into two major sections:

- **Local storage**: This is where your local variables (including parameters) are stored
- **Heap**: This is where all objects (created by new) are stored.

How do memory maps work?

(Figure omitted.)
Java Memory Layout

Question: When a method calls another, how does Java save local variables until returning?

Java Call Stack: The local variables for each method are stored on a stack.
- whenever a new method is called, its local variables (including parameters) are pushed onto the stack
- when the method returns, the local variables are popped off the stack (and hence are no longer accessible).

Call Stack: Example

```java
public class CallStack {

    public static int numberLowerCase(String theStr) {
        int count = 0;
        for (int i = 0; i < theStr.length(); i++)
            if (Character.isLowerCase(theStr.charAt(i))) count++;
        return count;
    }

    public static void stats(String str) {
        int total = str.length();
        int lower = numberLowerCase(str);
        System.out.println("String: " + str);
        System.out.println("Total count: " + total);
        System.out.println("Lower case count: " + lower);
    }

    public static void main(String[] args) {
        CallStack.stats("Que Buend\"");
    }
}
```

Call Stack: Example

```java
public static int numberLowerCase(String theStr) {
    int count = ... int i = ...
}

public static void stats(String str) {
    int total = ... int lower = numberLowerCase(str);
    ...
}

public static void main(String[] args) {
    CallStack.stats("Que Buend\"");
}
```

Additional Notes on Memory Maps

Aliasing: Recall that when one object reference is assigned to another, the memory address is copied, but not the object itself. Aliasing arises in method calls when reference variables are passed as arguments. (Figure omitted)

Local variables: disappear when method execution finishes but objects not automatically destroyed. They remain in the heap as long as a reference is associated with them (even after the method has finished execution). (Figure omitted)
Additional Notes on Non-Static vs. Static

A non-static method in a class can call a non-static method in the same class without having to create an object, because an object is already created. (More about this shortly.)

- Example given earlier:

```java
public class GoinOut {
    public void getDressed ( ...) { ... }

    public void getReady ( ...) { ...
        getDressed(...);
    }
}
```

A method in a class can call a static method in the same class without using the class name.

- Example given earlier:

```java
public class CallStack{ ...
    public static int numberLowerCase( String theStr ) { ... }
    public static void stats( String str ) { ...
        int lower = numberLowerCase( str ); ...
    }
}
```

Additional Notes on Non-Static vs. Static (continued)

A static main method defined in a class can create an instance of that class in main. However, a main method that accesses its own class instances behaves differently from a main method in a different class.

- Example:

```java
public class Challenge {

    private int x;

    private void processData() {
        System.out.println("Process Data");
    }

    public static void main(String[] args) {
        Challenge c = new Challenge();
        c.processData();
        c.x = 100;
    }
}
```

Examples of Static Methods (Review):

- JOptionPane.showInputDialog, Integer.parseInt, Math.sqrt
- Note: We've seen the Math class before. (You can use this for Challenge in HW 3):
  ```java
double r = Math.random(); // random double from 0.0 to 1.0
```
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.

```java
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 )`

**Prototype**: of a method is the signature of the method with the return type and public or private modifier
Method Overloading and Signatures (continued)

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

Example:

```java
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

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

```java
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.

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

Because of type promotion, there Java sometimes cannot figure out which method to call.

```java
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.

```java
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 (continued)

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

```java
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 (continued)

```java
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 (continued)

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” (continued)

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” (continued)

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

```java
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.

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

These are called respectively as follows:

```java
Basic.add(t1, t2);                                 // instance method
t1.addTo(t2);                                     // static method
```

---

**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.