How do I build a program?

Software development lifecycle:
- Analysis: What is the problem to be solved?
- Design: What is the general structure of the program?
- Implementation: Write the Java code to implement your design.
- Testing: Check each piece you have written.
- Maintenance: Specifications have changed, bugs fixed, new enhancements added.

The “cycle”: This is not a linear process.

Program Development Tips: Design

Analysis and Maintenance: No problem in CMSC 131. We give you the specifications.

Design: A good design strategy will save you enormous time in implementation and testing.

Stepwise Refinement:

Flowcharts:

Pseudo-code:
More Tips: Test and Implementation

Implementation and Testing:

Subtask Testing: As you complete the design of each task, write up a test implementation to see that your approach works.

Print: Use `System.out.println` to print intermediate results.

Hint: Don’t delete these until you are done debugging.

Debugger: Allows you to step through your program line by line.

Save: your work in a safe place, say, after implementing and testing each major task.

Classes and Objects

Back to Java...Brief review:
- Each Java variable stores either a primitive type (int, float, etc) or a reference to an object instance.
- A reference is the "address of" or a "pointer to" an object instance.
- There is a special reference, `null`, that refers to no object.
- Unlike primitive types, each object instance must be explicitly created using the "new" operator.

- Assigning (=) one reference variable to another copies the address, not the object contents.
- Comparing two references (with ==) compares the addresses, not the contents.
- A class is a definition or "blueprint" for an object. A class encapsulates both state (data) and behavior (methods).

Anatomy of a Class

A class contains declarations of:

Instance data: which form the state (values) of the object

Methods: which determine the behavior of the object
Example: Date

To illustrate this we define a class, called Date, that stores a date object (month, day, and year).

**Instance Data:**
- `int month`: Ranges from 1 to 12 (e.g. 1 = Jan, 2 = Feb, etc)
- `int day`: Ranges from 1 to 31 (depending on the month)
- `int year`: Four digit year (e.g. 2004)

**Methods:**
- `Date( int m, int d, int y )`: Creates a new Date object with the given month, day, and year.
- `toString()`: Returns a String representation of this date.
- `equals( Date d )`: Returns true if this date is the same as date d.

Example: Date.java (part 1)

```java
/*
 * Date: An object that stores a date
 */
public class Date {

    private int month;  // the month (from 1-12)
    private int day;  // the day of the month (1-31)
    private int year;  // the year (four digits)

    /* Constructor method initializes a new Date object */
    public Date( int m, int d, int y ) {
        month = m;  day = d;  year = y;
    }

    // (insert part 2 here)
}
```

Creating a Date Object

A Date object is created using "new":
```java
Date indepDay = new Date( 7, 4, 1776 );  // July, 4, 1776
```
This generates a call to the constructor:
```java
public Date( int m, int d, int y ) {
    month = m;  day = d;  year = y;
}
```
Example: Date.java (part 2)

```java
/* Converts to a string */
public String toString() {
    return new String(month + "/" + day + "/" + year);
}

/* Is this date equal to another? */
public boolean equals(Date d) {
    if ((year == d.year) && (month == d.month) && (day == d.day))
        return true;
    else
        return false;
}
```

String Conversion

Printing a Date object:

```java
Date indepDay = new Date(7, 4, 1776); // July, 4, 1776
System.out.println("Independence day is " + indepDay.toString());
```

This invokes the following method:

```java
public String toString() {
    return new String(month + "/" + day + "/" + year);
}
```

Output: Independence day is 7/4/1776

In fact, the following works as well:

```java
System.out.println("Independence day is " + indepDay);
```
Comparing Dates for Equality

Example:

```java
Date bobsBirthday = new Date( 7, 18, 1985 ); // July 18, 1985
Date carolsBirthday = new Date( 3, 23, 1985 ); // March 23, 1985
if ( bobsBirthday.equals( carolsBirthday ) ) ... // (false)
```

This invokes Bob's `equals` method:

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

Carol's birthday is the **actual parameter**. It is substituted for the **formal parameter** "d" in the method.
- year, month, day: Refer to this (Bob's) instance
- d.year, d.month, d.day: Refer to the **actual parameter** (Carol's) instance

Example: DateDemo.java

```java
/* This file demos the Date class */

public class DateDemo {
    public static void main( String[] args ) {
        Date bobsBirthday = new Date( 7, 18, 1985 ); // July 18, 1985
        Date carolsBirthday = new Date( 3, 23, 1985 ); // March 23, 1985

        System.out.println( "His birthday is " + bobsBirthday.toString( ) );
        System.out.println( "Her birthday is " + carolsBirthday.toString( ) );

        if ( bobsBirthday.equals( carolsBirthday ) )
            System.out.println( "Same birthday" );
        else
            System.out.println( "Different birthdays" );
    }
}
```
Class Elements
The Date example shows many features of classes and methods:
- Encapsulation and visibility (private and public)
- Method call and return
- Returning values from methods
- Method parameters and parameter passing
- Local data and scope
- Static and non-static methods

Next, we investigate each of these issues in greater detail.

Visibility and Encapsulation

Two views of a car:
- Driver's (External) view: (How to use it)
- Mechanic's (Internal) view: (What makes it work)

Two views of an object:
- Class user (client): sees the public interface.
- Class implementer: sees all the class's data and methods.

Visibility and Encapsulation

Visibility Modifiers:
private:
public:
[protected]: We will discuss this later.

Example:
public class Modifiers {
    public int pubData;
    private int privData;
    public void pubMethod() { /* omitted */ }
    private void privMethod() { /* omitted */ }
}

public class ModifierDemo {
    public static void main(String[] args) {
        Modifiers mod = new Modifiers();
        mod.pubData = 1;  // Okay: pubData is public
        mod.pubMethod();  // Okay: pubMethod is public
        mod.privData = 1;  // Illegal: privData is private
        mod.privMethod();  // Illegal: privMethod is private
    }
}

Visibility: Guidelines

What should be visible and what not?

Data instances should be private:

Example: month could reasonably be any of the following...

int from 1-12:
int from 0-11:
String: "Jan", "Feb", ...

Exception: Constants can be made public. The modifier final means that a variable is a constant, and its value cannot be changed.

public final int DAYS_PER_WEEK = 7;

Visibility: Guidelines

What should be visible and what not?

Methods in the public interface are public:

Utility/Support methods should be private:

Conventions: Methods are often further distinguished by their general function.

Accessors:

Mutators:

Examples: Possible additional methods for the Date class:

Accessor: int getMonth() { return month; }

Mutator: void incrementYear() { year++; }
Method Call and Return

When a method is invoked (or “called”), control jumps into the method. Control returns when:

- Control reaches the end of the method, or
- A "return" statement is explicitly executed

Return statements can be placed throughout a method.

Method Return Types and “void”

A method can either:

Return a value:

Return no value:

The special type "void" means that the method returns no value.

Examples: (Parameters and method bodies omitted)

```java
public String toString( ) // returns a String reference
public boolean equals( ... ) // returns a boolean
private double getPressure( ... ) // returns a double
public void printHelp( ) // returns no value
private void changeAddress( ... ) // returns no value
```
Methods and Parameter Passing

Information is passed into a method through a list of parameters. When defining a method, a list of formal parameters and their types is given:

```java
public void doSomething( double w, int x, String y) {
   ...}
```

To call the method, the corresponding actual parameters are given:

```java
int count = 53;
doSomething( 1.25, count+2, "Hello" );
```

Parameter Passing: These are copied to the formal parameters. Types must be compatible.

```java
public void doSomething( double w, int x, String y){
   System.out.println( w + " " + x + " " + y);
}
```

Pass by Value:

Static and non-Static Methods

Consider the following class method calls:

```java
String s = JOptionPane.showInputDialog( "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.showInputDialog( "Input an integer" );
int y = Integer.parseInt( s );
double c = Math.sqrt( (double) x );
```
Static and non-Static Variables

Static variable:

For example, in our Date class we might add:

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

These could then be accessed outside the class as:

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

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:

```
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..."
public class Date {
    private int month;
    private int day;
    private int year;

    public Date(int m, int d, int y) {
        // Constructor implementation...
    }

    public String toString() {
        // Method implementation...
    }

    public boolean equals(Date d) {
        // Method implementation...
    }

    public static boolean isLeapYear(int yr) {
        // Method implementation...
    }

    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;
    }
}