Program Development

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:
Stepwise Refinement and Pseudo-Code

(Figure omitted)

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
 */

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 indDepDay = 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)

/* 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;
}

Using the "toString" Method

Printing a Date object:

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

This invokes the following method:

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

Output: Independence day is 7/4/1776

In fact, the following works as well:

    System.out.println( "Independence day is " + indepDay);
Using the "equals" method

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

**Visibility and Encapsulation: Example**

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

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", ...

Methods in the public interface are public:

Utility/Support methods should be private:

Methods

- 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)

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)
- 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

Return

A method that returns a value must have a return statement.
- Where can it appear?
- What is the general form?
- What about type void?

```java
public int max(int x, int y) {
    int max = x;
    if (y > x) {
        max = y;
    }
    return max;
}
```

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

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

Scope: of a variable means the portion of the program (e.g., class, method, block) where a variable can be accessed.

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

Consider this example:

(code omitted)
do while ...
{
    double z = Math.random();
}
System.out.println("In dumb: z = " + z);  // ERROR: z cannot be resolved

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 dumb(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 dumb: z = " + z);
    }
}
Example: Test of Local Variables

```java
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 x = 5;
        int y = -6;
        LocalTest.smarter(3);
        System.out.println("Back in main: z = " + x + " y = " + y);
    }
}
```

Test Drivers

The previous example showed the use of a driver program.

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

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