Principles of Computer Science I

Prof. Nadeem Abdul Hamid

CSC 120A - Fall 2004

Lecture Unit 5

Homework 3

- Posted on Viking Web
- Due Wednesday, September 22

DON'T LEAVE IT UNTIL THE LAST MINUTE

Review: Chapter 3

- Arithmetic expressions
- Relationship between primitive and reference types
- Why different numeric types have different ranges of values
- Differences between integral and floating-point types
- How precedence rules affect evaluation of an expression
- Implicit type conversion and explicit casting
- String and Math operations (methods)
- Value-returning methods

Review: Java Programs

- Java applications (programs) are made up of classes
- Classes are blueprints for objects
  - Define what data objects need (fields)
  - Define operations on that data (methods)

  (Classes can also contain `static` fields and methods, which are not associated with an object)
- Fields have a name and type
- Methods have a name, return type, and parameters
  - Method bodies contain a bunch of statements
  - Methods can define local variables to store intermediate calculations

Flow of Control

- The order in which statements are executed
- Recall the 5 basic control structures:
  - Sequence
  - Selection
  - Loop
  - Subprogram
  - Asynchronous
- So far we have just defined methods with a sequence of statements in their body

Write a (value-returning) method…

… to convert an integer representation of a SSN into a String formatted as "nnn-nn-nnnn" *(we did this before)*

```
public static String ssnToStr(int ssn) { … }
```

- What happens if we call the method with these arguments:

```
– ssnToStr(123456789);
– ssnToStr(012345678);       // !!!
– ssnToStr( 12345678);
```

SsnToStr Program

```
public class SsnToStr {

    public static String ssnToStr(int ssn) {
        String ssnstr;
        ssnstr = "" + (ssn/1000000) + "-" +
                      ((ssn%1000000)/10000) + "-" +
                      (ssn%10000);
        if (ssn < 100000000) ssnstr = "0" + ssnstr;
        return ssnstr;
    }

    public static void main(String args[]) {
        System.out.println(ssnToStr(123456789));
        System.out.println(ssnToStr( 12345678));
    }

}
```


The *Selection* Control Structure

- Use selection or *branching* when you want the computer to choose between alternative actions
- Make an assertion (a test condition)
  - If `true`, computer executes one (or more) statement(s)
  - If `false`, computer executes another

Java *if-else* Statement

```
if ( b )
   statementA;
else
   statementB;
```

- `b` is an expression that evaluates to a `boolean` value
- `boolean` data type constants:
  - `true`
  - `false`

Other Forms of the *if* Statement

```
if ( b )
   statementA;

if ( b1 )
   statementA;
else if ( b2 )
   statementB;
else
   statementC;
```

```
if ( b1 )
   statementA;
else if ( b2 )
   statementB;

if ( b1 ) {
   statementA;
   statementB;
   statementC;
}
```

- Wherever a single statement appears, we can instead put a block of statements enclosed in a pair of braces
- Notice indentation style of the statements that depend on the conditional test

Logical Expressions

- Arithmetic expressions (*e.g.*, 1+2) evaluate to a numeric value
- Logical expressions (or boolean expressions) evaluate to a boolean value
- Logical expressions are made up of:
  - `boolean` variables or constants
  - relational operators
  - logical operators

Relational Operators

- Compare two expressions
  - In other words, they test a relationship between two expressions or values

Operator	Relationship tested
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

- In the `ssnToStr` method:

```
if (ssn < 100000000) ssnstr = "0" + ssnstr;
```

Approximating Factorial

The factorial of a number n is $n*(n-1)*(n-2)*...*2*1$. Stirling's formula approximates the factorial for large values of n:

$$n! = \frac{n^n \sqrt{2\pi n}}{e^n}$$

where $\pi = 3.14159265$ and $e = 2.718282$.

Write a Java program that inputs an integer value, stores it in a double variable n, calculates the factorial of n using Stirling's formula, assigns the (truncated) result to a long integer variable, and then displays the result.

Depending on the value of n, you should obtain one of these results:

- A numerical result.
- If n equals 0, the factorial is defined to be 1.
- If n is less than 0, the factorial is undefined.
- If n is too large, the result exceeds Long.MAX_VALUE.

Comparison Examples

- Let x=5 and y=10, what do these expressions evaluate to:

```
x != y          x < y
x*y == y*x      x*2 <= y
'M' < 'R'       'm' < 'r'
'm' < 'R'
0 < 9           '0' < 9
```

- Beware the difference between the assignment operator (=) and relational operator (==)
  - Some people call == "equals equals"

Comparing Strings

- What does this print out?

```
String astr = "Hello World";
String bstr = "Hello";
String cstr = "World";
String dstr = bstr + " " + cstr;
String estr = "Hello World";

System.out.println(astr + "\n" + dstr);
System.out.println(astr == dstr);
System.out.println(astr == estr);
System.out.println(astr != dstr);
```


String Comparison (cont.)

- Comparing two objects using == or != means comparing the value of the pointers
- For object comparison, we can often use the `compareTo` method
  - `compareTo` returns 0 if two objects are equal, a positive integer if one is "greater" than the other, and a negative integer if one is "less" than the other
- For String objects, we can also use:

```
strA.equals(strB);
strA.equalsIgnoreCase(strB);
```


String Comparison (cont.)

- The `toLowerCase()` method for Strings converts all the characters to lower case
- The `toUpperCase()` method for Strings converts all the characters to upper case
- We can apply String methods to literal constants as well

```
name.equals("Doe")
"Doe".equals(name)
```



```
String astr = "Hello World";
String bstr = "Hello";
String cstr = "World";
String dstr = bstr + " " + cstr;
String estr = "Hello World";

System.out.println(astr + "\n" + dstr);

System.out.println(astr == dstr);
System.out.println(astr == estr);
System.out.println(astr != dstr);

System.out.println(astr.equals(dstr));
System.out.println(astr.equals(dstr.toLowerCase()));
System.out.println(astr.equalsIgnoreCase
                   (dstr.toUpperCase()));
System.out.println(bstr.compareTo(cstr));
```

Checking for Alphabetical Order

- Strings are compared by comparing each character, beginning from the first one
- Characters are compared according to their ordering in the Unicode character set
  - All capital letters come before lower case

```
"Macauley".compareTo("MacPherson")  //  >
```

  - Better to convert to lower (or upper) case before comparing words for alphabetical order

```
("Macauley".toLowerCase()).compareTo
   ("MacPherson".toLowerCase())     //  <
```


Comparing Strings of Different Length

- Shorter string is "less"

```
String word = "Small";
word.compareTo("Smaller") < 0;  // true
```

- Recap on String operations:
  - Apply a method to one String object
  - Provide the second object (*e.g.* for comparison) through method argument
    `object1.method(object2);`
  - Compares the contents of `object1` data fields to those of `object2`

Logical Operators

- Used to combine assertions (expressions of boolean value)
- In English,
  - If x > 6 and y < 5 then …
  - If x = 5 or y > 11 then …
- In Java,
  - And b1 && b2
  - Or b2 || b2
  - Not ! b
- && and || are binary operations, ! is unary

Logical Equivalences

```
!(a == b)                      a != b
!(hours < 40)                  hours >= 40
!(a==b || a==c)                a != b && a != c
!(a==b && c > d)               a != b || c <= d
!(x!=y && y==z)                    ?
!(x>y && (z/5>2 || q!=y))          ?
!((false || true) && true)
```

- DeMorgan's Law: distributing ! over &&, ||

Truth tables

- And

X	Y	X && Y
true	true	true
true	false	false
false	true	false
false	false	false

- Or

X	Y	X || Y
true	true	true
true	false	true
false	true	true
false	false	false

- Not

X	!X
true	false
false	true

Short-Circuit Evaluation

- Full evaluation: evaluate both subexpressions of a logical operator
- Short-circuit (in Java,C,C++): evaluate subexpressions starting from the left, stop as soon as the value of the entire expression is determinable

```
c <= d || e == f
a == 5 && b < 5
```

- Why useful?

```
if (i != 0 && n/i > 500) ...
```

Bitwise Logical Operators

- Operate on bit by bit on data values
  - And &
  - Or |
  - Not ~
  - (some others …)
- Can apply to either boolean or integral data types (~ doesn't work with booleans)
- Are not short-circuit operators

Operator Precedence

- Highest precedence:

```
()
!
++   --  (postfix)
++   --  (prefix)
+   -  (unary)
*    /    %
+ -
< <= > >=
==   !=
&&
||
=   +=   *=   ...
```

Lowest precedence: (last row above)

- Most associate left-to-right, except ! associates from right

Translate to Java

- "midterm grade or final grade equals A"
- "i equals either 3 or 4"
- "y is between 12 and 24"
- "x, y, and z are greater than 0"
- "x is equal to neither y nor z"

Relational Operators with Floating-Point

- Don't compare floating-point numbers for equality
  - Round-off errors accumulate in computation
  - Expression may evaluate to .9999999 instead of 1.0
- Test floating-point numbers for equality by subtracting and comparing to maximum allowable difference
  - Math.abs(sum – 1.0) < 0.0000001

if/else Statement with Blocks

- Surround block with { … }

```
if (divisor != 0) {
    System.out.println("Division performed.");
    result = dividend / divisor;
} else {
    System.out.println("Cannot divide by zero.");
    result = Integer.MAX_VALUE;
}
```

- Alternate brace style (be consistent)

```
if ( ... )
{
    ...
}
else
{
    ...
}
```


Nested if Statements

```
if today is Saturday or Sunday

      if it is raining
            Sleep late
      else
            Get up and go outside

else
      Go to work
```

Multiway Branches and Efficiency

```
if (month == 1) monthName = "January";
if (month == 2) monthName = "February";
if (month == 3) monthName = "March";
...
```

- versus

```
if (month == 1) monthName = "January";
else
  if (month == 2) monthName = "February";
  else
    if (month == 3) monthName = "March";
    else ...
```

- Style: Keep else's at same indentation level for this "if-else-if" structure
- Be careful about braces
- Multiple if's may allow several cases to execute; if-then-else structure only allows one case

Using Nested if Statements

- Display a message indicating appropriate activity for outdoor temperature:
  - Swimming temp > 85
  - Tennis 70 < temp <= 85
  - Golf 32 < temp <= 70
  - Skiing 0 < temp <= 32
  - Dancing temp <= 0

Dangling else Statements

- Sometimes confusing to match else with if
- Correct:

```
if (avg < 70.0)
   if (avg < 60.0)
      System.out.println("Failing");
   else
      System.out.println("Passing but marginal");
```

- Incorrect (semantically):

```
if (avg >= 60.0)
   if (avg < 70.0)
      System.out.println("Passing but marginal");
else System.out.println("Failing");
```

- Correct (use braces):

```
if (avg >= 60.0) {
   if (avg < 70.0)
      System.out.println("Passing but marginal");
}
else System.out.println("Failing");
```

Exercises

- Implement the Math.abs(x) method
- Implement the Math.min(x,y) method
- Write a program that takes as input a number less than 256 and prints out its binary (base-2) representation
  - Redo it, factoring out repetitious code into a method
- Write a program to input the first 9 digits of an ISBN number and compute the tenth
  - Use the Keyboard class for input

Exercise: Complex Numbers

- Define a data type (class) for complex numbers
  - Represent a complex number by explicitly storing the real and the imaginary parts as floating point numbers
  - Support the following standard arithmetic operations: add, multiply, absolute value (magnitude), opposite (negative), and complex conjugate.
  - Also support the utility operations: initialization, equality comparison, and conversion to string representation.
  - As an example, a client program should be able to write code like the following:

```
public static void main(String[] args) {
    Complex a = new Complex( 5.0, 6.0);     //   5 + 6i
    Complex b = new Complex(-2.0, 3.0);     //  -2 + 3i
    Complex c = b.times(a);                 // -28 + 3i
    System.out.println("c = " + c);
}
```

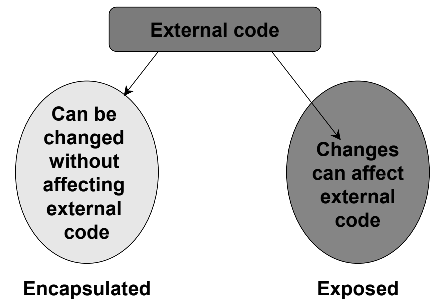

Designing Large Applications

- Most programs we've written so far have been fairly simple
  - Only a `main` method, or few others
  - Not much interaction between objects
- Once we learn some more basic control structures (loops), we will start to build larger, more complex programs
- Will need to start thinking about design and reusability of our classes
- Two important (related) concepts:
  - Encapsulation
  - Abstraction

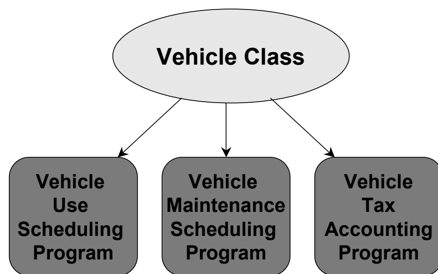
Encapsulation

- A capsule protects contents from outside contaminants or harsh conditions
- Design classes so that the internal implementation is protected from external code
  - External code interacts only through a well-designed interface
- Benefits:
  - Simplifies design of large programs by developing parts in isolation
  - Modifiability
  - Reuse

Encapsulated vs. Exposed Implementation



Class Reuse



Example: Circle program

- A non-encapsulated, non-reusable program:

```
import java.io.*;
public class Circle {
    public static void main(String [] args) throws IOException {
        float diameter;                 // Diameter of the cirlce
        float radius;                   // Radius
        float area;                     // Area
        float circumference;            // Circumference
        final float PI = 3.14159F;      // PI

        BufferedReader in = new BufferedReader(new
                            InputStreamReader(System.in));
        System.out.println("\nEnter the radius of the circle:");
        radius = Float.parseFloat(in.readLine());

        diameter = radius*2;
        area = PI*radius*radius;
        circumference = 2*PI*radius;

        System.out.println("Radius: " + radius);
        System.out.println("Diameter: " + diameter);
        System.out.println("Area: " + area);
        System.out.println("Circumference: " + circumference);
    }
}
```


Circle version 2 (more reusable)

```
public class CircleDriver {
    public static void main(String [] args) throws IOException
    {
        CircleNonEncap c = new CircleNonEncap(4);
        c.radius = 6;
        System.out.println("Diameter is: " + c.getDiameter());
    }
}

class CircleNonEncap {
    public int radius;

    // construct a circle object with given radius
    public CircleNonEncap(int newr) {
        radius = newr;
    }

    public int getRadius() { return radius; }

    public int getDiameter() { return radius * 2; }
}
```


Circle version 2, Modifiability?

```
class CircleNonEncap {
    public int diameter;

    // construct a circle object with given radius
    public CircleNonEncap(int newr) {
        diameter = newr * 2;
    }

    public int getRadius() { return diameter/2; }

    public int getDiameter() { return diameter; }
}
```

- Problem: The internal implementation is exposed
  - We don't want to let outside code depend on how the circle is represented by this class

Circle version 3

```
public class Circle {
    // publically accessible class constants
    public static final double PI = 3.14159;

    // private data of the implementation
    private double radius;

    // Constructor: inputs a radius value from the keyboard
    public Circle() {
        Keyboard in = new Keyboard();
        radius = in.readDouble();
        if (radius < 0.0) radius = 0.0;
    }

    // Constructor: takes a radius value as a parameter
    public Circle(double new_r) {
        if (new_r >= 0.0) radius = new_r;
        else radius = 0.0;
    }

    // "accessor" methods
    public double getRadius() { return radius; }
    public double getDiameter() { return radius * 2; }
    public double getArea() { return PI * radius * radius; }
    public double getCircumf() { return 2 * PI * radius; }

    public void setRadius(double new_r) {
        if (new_r >= 0.0) radius = new_r;
    }

    public void setDiameter(double new_d) {
        if (new_d >= 0.0) radius = new_d / 2.0;
    }

    public void printCircleData() {
        System.out.println("Radius: " + getRadius());
        System.out.println("Diameter: " + getDiameter());
        System.out.println("Area: " + getArea());
        System.out.println("Circumference: " + getCircumf());
    }

. . .
```


Price of Exposed Implementations

- *ZIP codes.*
  - In 1963, the USPS began using a 5 digit ZIP (Zoning Improvement Plan) code to improve the sorting and delivery of mail. Programmers wrote lots of software that assumed that zip codes would remain 5 digits forever, and represented them in their programs using a single integer. In 1983, the USPS introduced an expanded ZIP code called ZIP+4 which consists of the original 5 digit ZIP code plus 4 extra digits to assist in delivery. This broke hundreds of brittle programs and required millions of dollars to fix. One of the lessons we should learn from this is to use encapsulated data types so that if we must change the data representation, the effects are localized to one abstract data type (*i.e.* class, in Java) and we don't need to search through millions of lines of code to find all of the places where we assumed ZIP codes were 5 digits long.

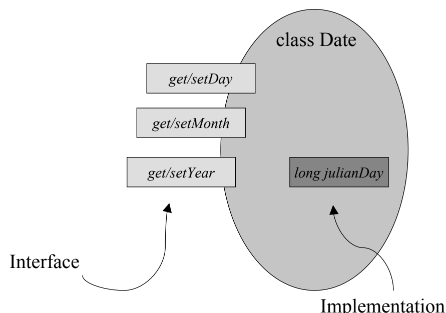
Time Bombs

- Y2K bug
- On September 14, 2004, Los Angeles airport was shut down due to software breakdown of a radio system used by air traffic controllers to communicate with pilots. The program used a Windows API function call *GetTickCount()* which returns the number of milliseconds since the system was last rebooted. The value is returned as a 32 bit integer, so after approximately 49.7 days it “wraps around.” The software developers were aware of the bug, and instituted a policy that a technician would reboot the machine every month so that it would never exceed 31 days of uptime. Oops. *LA Times* blamed the technician, but the developers are more to blame for shoddy design.

Abstraction

- To abstract = to simplify or eliminate irrelevant details
- Putting encapsulation into practice
  - Separating the logical properties of an object from its implementation
- Abstraction in other disciplines and across disciplines:
  - ***Chemistry is an abstraction of Physics***: The purpose of Chemistry is to understand molecular interactions without resorting to particle physics to explain every phenomenon.
  - ***Biology is an abstraction of Chemistry***: The purpose of Biology is to understand the growth and behavior of living things without resorting to molecular explanations for every aspect.

Abstraction: Date Class



Abstraction, cont.

- Abstraction does not mean the external view of data and implementation has to be different…
  - Date class could have used internal fields for the day, month, and year
- …but it should be irrelevant to the user of the class
  - Changes later on do not affect external code

Debugging

```
/* A messed up class ... can you
/* find (at least!) 6 errors?

public class CircleDriver {
    public static void main(String[] args) {
        c1 = new Circle(.90);
        String c2 = new Circle();

        c1.scale(1/2);  // reduce to half size

        boolean b = (c1==c2);
        if (b = true)
            System.out.print("They are ");
            System.out.println("equal!");

    }
}
```

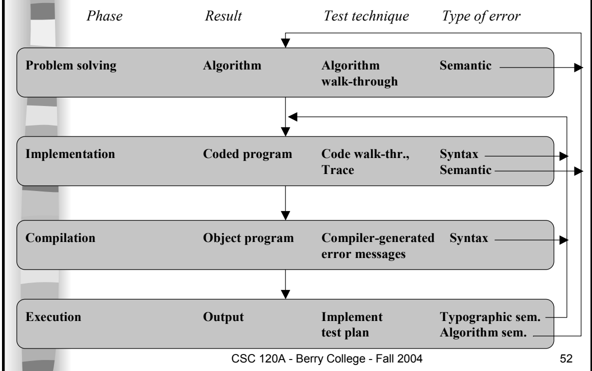

Types of Errors

- Syntactic errors
  - Found by the Java compiler
  - Misspellings, undeclared identifiers, missing semicolons/braces/parentheses, mismatched operands
- Semantic (Logical) errors
  - Mistakes that result in wrong answer(s) when the program is run
  - May be misspellings that are syntactically correct
  - Using = instead of ==; mistyping a constant literal; not enclosing braces around block of an if statement
  - Java detects very few of these: *e.g.* dividing by zero

Testing Strategies

- Purpose of testing is *not* to verify code is correct; it is to find flaws in your program
  - A successful test is one in which you find an error in your code
- Testing is part of the problem-solving and implementation stages (software life-cycle)
- In real world, tester and programmer are two different people
- Some hints
  - Compile and test your program early and often
  - Test after every change – modifications to one part may affect others
  - Use debugging printouts

Testing Process



Testing Techniques

- Walk-through
  - Perform a manual simulation of the code or design
- Inspection
  - Read the code or design line by line to identify errors
- Execution trace (hand trace)
  - Pretend you are the computer
  - Walk through program code with some actual values to trace execution of the program
  - Test branches using different data sets

Test Cases

- Test all possible paths in your code using different sets of input data
- White-box testing (code coverage)
  - Design tests while looking at the actual code implementation in the methods
- Black-box testing (data coverage)
  - Test as many allowable data values as possible for a method without looking at the actual code
  - Considering pre- and post-conditions of methods, use input that tests boundary cases and typical scenarios

Developing a Test Plan

- For a given program, a document specifying
  - Test cases that should be tried
  - Reason for each test case
  - Expected output
- Run the program and compare observed output to the expected
- Example: Complex numbers