Principles of Computer Science I

Prof. Nadeem Abdul Hamid

CSC 120A - Fall 2004

Lecture Unit 3

Recap: Types of Methods

- Class methods
  - Associated with a class instead of object instances
  - Defined with the `static` modifier
- Instance methods
  - Methods that operate on the data of an object
- Constructors
  - Special methods called automatically when you create a new object of a class
  - Name must be exactly the same as the class
  - Do not specify a return type (like `void`)
- Void methods
- Value-returning methods
- Helper methods
  - Declared `private`ly in a class; used internally

Recap: Java Applications

- A class or classes containing fields and methods
- Fields: identifier and type
  - Can be variables or constants
- Methods: declarations, statements, expressions, method calls, input, output
- Comments
- One class contains the `main` method

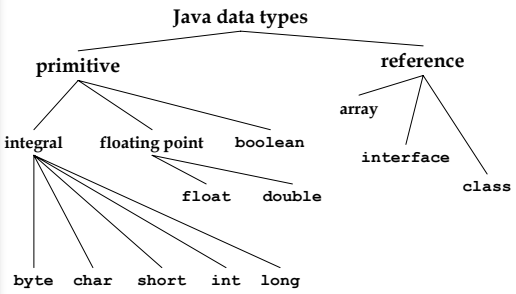
Review Chapter 2 Goals

- Understand the distinction between syntax and semantics
- Why is it important to use meaningful identifiers in programming
- Understand similarities and differences
  - built-in (primitive) types and objects
  - `char` and `String`
  - named constant and variable
  - assignment of an object and of a primitive type value
  - void and value-returning methods
- Understand how a Java application is composed of a class with one or more methods

Quick Quiz

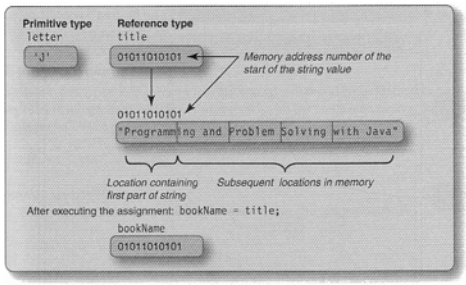
- Declare a class called `Quiz`
- Declare a field of type `char` called `ch`
- Declare a field of type `String` called `str`
- Define a constructor (method) for the class that assigns `'!'` to `ch` and `"Ok"` to `str`
- Define a void method called `printit` with no parameters
- Declare a local string variable called `x` in the method and assign it the concatenation of `str` and `ch`
- Insert a statement in the method to print out `x` on the screen

Java Data Types (Complete List)



Primitive vs. Reference Types

```
char letter = 'J';
String title = "Programming and Problem Solving with Java";
String bookName = title;
```



Copying References

- Changing the object through one reference affects all other references
  - Can be useful
  - Can be confusing
  - Better avoid this for now

Integer Data Types

- `byte`: 8 bits (less frequently used)
- `short`: 16 bits (less frequently used)
- `int`: 32 bits (most used – up to 9 decimal digits)
- `long`: 64 bits (up to 18 decimal digits)
- Literals
  - `int:` `0` `2001`
  - `long:` `0L` `18005551212L`
- Java does not produce an error message if overflow occurs
- Caution: Integer literal beginning with zero is assumed to be octal (base-8): `015` = 13_{10}

Floating-Point Types

- Integer part and fractional part
  - `float`: 32 bits
  - `double`: 64 bits (default)
- Examples:

```
18.0    127.54    0.57    4. 193145.8523
.8           1.74536E-12          7e20

2.001E3F      0.0f      (float literals)
```

- Many decimal floating-point values can only be approximated in base-2 system
  - Program may print `4.799998` instead of `4.8`

Scientific Notation

2.7E4 means 2.7×10^4 =

2.7000 =

27000.0

2.7E-4 means 2.7×10^{-4} =

0002.7 =

0.00027

Declarations for Numeric Types

- Can define fields, variables, constants just as for `char` and `String`
- Named constant examples:

```
final double   PI = 3.14159;
final float    E = 2.71828F;
final long     MAX_TEMP = 1000000000L;
final int      MIN_TEMP = -273;
final char     LETTER = 'W';
final String   NAME = "Elizabeth";
```

- Variables:

```
int num = 2;
char ch = '2';
```

Why Named Constants?

- Readability
- Ease of modification
- Reliability

Arithmetic Expressions

- Made up of constants, variables, operators, and parentheses

```
num + 2       rate – 6.0      4 - alpha
```

- Arithmetic operators:
  - unary plus: `+259.65` `+alpha`
  - unary minus: `-54` `-rate`
  - addition: `a + b`
  - subtraction: `b - a`
  - multiplication: `a * b`
  - division (floating-point or integer): `b / a`
  - modulus (remainder): `b % a`

Division and Modulus

- Integer
  - 7 / 2 = 3
  - 7 % 2 = 1
    - 3 % 2 = 1
    - 3 % -2 = 1
    - -3 % 2 = -1
    - -3 % -2 = -1
  - 7 / 0 and 7 % 0 → error
- Floating-point
  - 7.0 / 2.0 = 3.5
  - 7.2 % 2.1 = 0.9
  - 7.0 / 0.0 = *infinity* 7.0 % 0.0 = *not a number (NAN)*

Assignment Statements

```
int num;
int alpha = 10;

num = 6;
num = num + alpha;
alpha = alpha % 7;
```

- Remember: this = in Java is assignment, *not* mathematical equality

Increment / Decrement Operators

- ++ and --

`num++;` *is the same as* `num = num + 1;`

- *Prefix* as well as *postfix* versions:

```
num++;      ++num;

num = 5; alpha = num++ * 3;  // alpha = 15, num = 6
num = 5; alpha = ++num * 3;  // alpha = 18, num = 6
```


Operator Precedence

- Determines which operator is applied first in an expression having several operators

```
avgTemp = FREEZE_PT + BOIL_PT / 2.0;
```

- Highest precedence:
  - `()`
  - `++ --` *(postfix)*
  - `++ --` *(prefix)*
  - `+ -` *(unary)*
  - `* / %`
- Lowest precedence: `+ -`

- Change order of evaluation using parentheses

Operator Associativity

- In Java: * / % + - are left associative
  - in an expression having two operators with the same priority, the left operator is applied first
- 9 – 5 – 1 means (9 – 5) – 1 = 3
- Evaluate:

```
7 * 10 - 5 % 3 * 4 + 9
```

= 71

Type Conversion

- Integers and floating-point numbers are represented differently in the computer
- What happens here?

```
double someDouble = 12; // Java automatically con-
                        // verts value to 12.0
int someInt = 4.8;   // Java compiler gives an error
                     // "possible loss of precision"
```


What do we get?

```
double A = 3 * 7 - 2;
double B = 7 / 3 + 1;
double C = 7 / 3 + 1.0;
double D = 7.0 / 2 + 1;

System.out.println(A);
System.out.println(B);
System.out.println(C);
System.out.println(D);
```

More Conversion

- Widening conversion
  - e.g. from int to double (OK)
- Narrowing conversion
  - e.g. from double to int (not OK if implicit)

- Type casting
  - Tell Java explicitly to convert values

```
double someDouble = (double) 12;
int someInt = (int) 4.8;   // Java accepts this now
System.out.println( (double)(7/2) );
System.out.println( (double)(7) / (double)(2) );
```


Type Casting

- Makes it clear that type mixing is intentional
  - Even if the program would run fine without the explicit casts
- Often necessary for correct results

```
int sum, count;
double average;
...  // compute sum = 60, count = 80
average = sum / count;  // average is 0.0

average = (double)sum / count;  // this is better
average = (double)sum / (double)count; // or this
```


String Conversion

- Java automatically converts numbers to Strings when mixed in an expression with the + (concatenation) operator

```
double value = 27.85;
String answer = "The answer is: " + 27.85;
        // answer is now "The answer is: 27.85"
```

- But be careful…

String Conversion Examples

```
answer = "Result: " + 27 + 18 + " and " + 9;
       // answer is "Result: 2718 and 9"
answer = "Result: " + 27 + ", " + 18 + " and " + 9;
       // answer is "Result: 27, 18 and 9"
```

- Why isn't the first one: "Result: 45 and 9"?
  - *Hint: precedence and evaluation order (associativity)*
- How about this:

```
answer = 27 + 18 + 9 + " are the results."
```

- Or these:

```
answer = 27 + 18 + 9;
answer = "" + 27 + 18 + 9;
answer = "" + (27 + 18 + 9);
```


Useful Methods in the `Math` class

- Table 3.1 (page 122) in textbook
  - Math.abs(x)
  - Math.cos(x)
  - Math.sin(x)
  - Math.log(x) *// natural logarithm (base=e)*
  - Math.pow(x,y)
  - Math.min(x,y)
  - Math.max(x,y)
  - Math.random()
  - Math.round(x)
  - Math.sqrt(x)
- Use them like this:

```
double root = Math.sqrt(99);
```


`String` Methods

- The length() method returns an int value that is the number of characters in the string

```
String name = "Alexandra";
int len = name.length();   // len = 9
```

- indexOf() searches for a substring and returns the beginning position in the string (starting from 0) or -1 if it's not a substring

```
String phrase = "The dog and the cat";
int posA = phrase.indexOf("the"); // posA = ?
int posB = phrase.indexOf("rat"); // posB = -1
```

Substrings

- Method substring() returns a substring of a string, but does not change the string itself

```
String name = "Programming and Problem Solving";
name.substring(0,7);          // "Program"
name.substring(7,15);         // "ming and"
name.substring(10,10);        // ""
name.substring(24,25);        // "S"
```

  - First parameter: starting position
  - Second parameter: one past the ending position
  - Returns: a String value

Using `substring` Safely

- Bad parameters result in a runtime error:

```
String name = "Programming and Problem Solving";
name.substring(10,50);
        // Error: String index out of range: 50
```

- Safer method call:

```
String name = "Programming and Problem Solving";
int start = 10;
int len = 40;
name.substring(start, Math.min(start+len,
                                  name.length());
```


String operations

- What does this print out?

```
String fullname = "Jonathan Alexander Peterson Jr.";
int start = fullname.indexOf("Peterson");
String name = "Mr. " + name.substring(start,
                                  name.length());
```

Converting Strings to Numbers

- The BufferedReader class allows us to get lines of text (strings) from the keyboard
- To convert input from String type to numeric type, we must use the appropriate method:

Primitive type	Object type	Method
int	Integer	parseInt
long	Long	parseLong
float	Float	parseFloat
double	Double	parseDouble

Getting Number Input

```
BufferedReader in = new BufferedReader(new
                    InputStreamReader(System.in));
int myNumber;
System.out.println("Enter an integer number: ");
myNumber = Integer.parseInt(in.readLine());
System.out.println(myNumber + " squared is " +
                           (myNumber*myNumber));
```

- What if evil user enters something besides a number?
  - Our program crashes with a NumberFormatException (until Chapter 9)

Applications with Multiple Class Files

- Many benefits
  - Smaller chunks of stuff to work with at a time
  - Reuse in other applications
  - Compile/test/debug one at a time
- In Java, name each file *exactly* the same as the class defined inside it
- We only have to make classes `public` if they are to be accessed by other entities outside the directory
  - like the JVM – needs to get to the main method
- New Name/NameDriver example
  - Book uses "import Name;" statement – you don't (you'll probably get an error if you try)