CS 117
Fall 2003

Variables
Numerical operations
Formatting numbers

September 24, 2003

Variables

- Memory locations that store data
- 3 components:
  - data type (String, int, double, JFrame)
  - name
  - value
- Today: numerical data

Numerical data types in Java

- **byte** integer [-128, 127]
- **short** integer [-32768, -32767]
- **int** integer [-2.1x10^9, 2.1x10^9]
- **long** integer [-9.2x10^18, 9.2x10^18]
- **float** real [-3.4x10^38, 3.4x10^38]
- **double** real [-1.7x10^308, 1.7x10^308]

Declaring and assigning values to variables

- Variables must be **declared** before we can use them
  
  int a, b, c;
  
  a = 10;
  
  b = 30;
  
  c = 40;

- Can combine these two steps (declaration and assignment)
  
  int a = 10;
  int b = 30;
  int c = a + b;
Declaring and assigning values to variables, part 2

- In the previous example, we just assigned a value right away to an integer variable
- In some cases, we have to create a new object of the class first, before we can use the variable
  
  ```java
  JFrame win = new JFrame("My window");
  win.setSize(300,200);
  ```

- Q: Why is there a difference?

Java has 2 kinds of data types

- **Primitive data types** already have memory allocated for them, so we don't have to create them
  - e.g., int, double
  - String acts like one most of the time
- **Reference data types** do not have memory allocated automatically
  - when we create a new object of a class, we are allocating memory to store it
  - e.g., JFrame, Account

What happens when we initialize a primitive data type variable?

```java
int a = 265;
```

- Set aside 4 bytes of memory in the next available memory slot in the program
- Copy the value 256 into this spot
- Label this spot “a”

What happens when we create a new object?

```java
JFrame win = new JFrame("My window");
```

- Set aside the next slot of memory in the program
- Create a new JFrame object somewhere else in memory. Get this spot's address.
- Copy the address into the slot
- Label this slot “win”
Why the difference?

- Primitive data types, like `int`, always take up the same amount of memory
  - each `int` is 4 bytes
- Reference data types, like `JFrame`, take up varying amounts of memory
  - the size of a `JFrame` object depends on many factors: the values of the data fields, the size of the frame, the number of objects inside the frame, ...
- The Java compiler likes consistency
  - so, it sets aside the same amount of memory for all reference data types, and uses these memory slots to store the addresses of the actual objects
  - it then creates the objects wherever there's room, and just "points" to them (stores their addresses)

Arithmetic expressions in Java

- Binary operators: `+ - * / %`
  - `*` is multiplication
  - `/` is division
    - dividing 2 integers also yields an integer:
      - `3/2 = 1` but `3/2.0 = 1.5` and `3.0/2 = 1.5`
  - `%` is *modulo division* (the remainder of dividing two numbers)
    - `10 modulo 5` is `0`
    - `10 modulo 7` is `3`
- Unary operators: `+ -`

Arithmetic precedence rules

- Determines the order of operations in an arithmetic expression
- Rules:
  - parentheses first
  - then unary (+/-)
  - then multiplication, division, modulo
  - then addition, subtraction

Examples

- `15 + 4 * 6 / 3 – 8`
- `2 * (1 + (-3) / 2) * (4 % (2 + 1))`
- `3 + 5 / 3`
- `(18 – 4) / 6.0`
- `(6 * 3 + 2) % (5 + 3)`
Constants

- A data value that does not change during the program's execution
- Use the keyword `final` to indicate a constant
- Declare constants with the rest of the variables, at the top of the program (after the class declaration)
- Typically, names are all capital letters
- Examples
  ```java
  final int MAX GRADE = 100;
  final double MIN BALANCE = 100.00;
  ```

Predefined constants

- Java defines some constants for us already
  - $\pi$, $e$
- Can access these through the `Math` class:
  ```java
double circleArea = Math.PI * r * r;
double number = 2 * Math.E;
```
More advanced arithmetic operations

- The Math class contains common math functions:
  - Trig functions
    \[ \sin(a), \cos(b), \tan(a+b), \acos(a), \ldots \]
  - Logarithms/exponential
    \[ \exp(-a) = e^{-a} \]
    \[ \log(b) = \ln(b) \]
  - Powers
    \[ \text{pow}(a, b) = a^b \]
  - Rounding
    \[ \text{ceil}(a) \text{ rounds up: } \text{ceil}(10.6) = 11.0 \]
    \[ \text{floor}(a) \text{ rounds down: } \text{floor}(10.6) = 10.0 \]
- See chart on page 114-115 of Wu

Formatting numerical data

- Sometimes we want to output our numerical data in a certain way
  - e.g., in the Account class, we might want to print out the current balance as $xxxx.xx, not $xxxx.x or $xxxx.xxx
- Solution: use the DecimalFormat class

Example: Account class

```java
class Account {
    ...
    public static void main(String[] args) {
        DecimalFormat df = new DecimalFormat("0.00");
        Account acct = new Account("Jane Doe", 1000);
        System.out.println("Account holder:  "+acct.name);
        System.out.println("Account balance:  "+df.format(acct.balance)); /* balance is displayed to 2 decimal places */
    }
    ...
}
```

Numerical representation

- Computers store numbers in terms of 1's and 0's
  - binary numbers
- Q: How do we convert from an integer to a binary number (and vice versa), and from a real number to a binary number (and vice versa)
  - also, what happens with negative numbers?
Binary numbers

- also called base-2
- Examples:
  - 1 is 00000001
  - 2 is 00000010
  - 3 is 00000011
  - 4 is 00000100
  - 5 is 00000101
  - ...

Binary numbers (cont.)

- $00010111 = (0 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 11$
- $00100101 = ?$
- 7 in binary:
  - $7/4 = 1$ remainder 3
  - $3/2 = 1$ remainder 1
  - $1/1 = 1$ remainder 0 --> $00000111$
- 22 in binary = ?

Negative integers in binary

- Idea: Use 8 bits as before, but the first bit is a “sign bit”
  - 0 = positive, 1 = negative
- 1 = 00000001, -1 = 10000001
- 7 = 00000111, -7 = 10000111
- 0 = 00000000 and 10000000
  - bad!

Twos complement

- Allows us to represent negative numbers in binary, without the “two zeros” problem
- All positive numbers start with a 0, as before
- Negative numbers:
  - Invert all the bits (change 0’s to 1’s and 1’s to 0’s)
  - Add 1
- Example: $7 = 00000111$, -7 = ?
  - Invert: 00000111 --> 11111000
  - Add 1: 11111000 + 1 = 11111001
Representing real numbers in binary

- Use scientific notation: \( a \times 2^N \)
- \( a \) and \( N \) are both binary
- Format:
  - \( S \): sign bit
  - \( N \): exponent
  - \( A \): binary number in the format 1.xxxxxx...