Announcements


- Tutoring schedule is ready (see course web)
  - Try to get help from tutors as much as you can
  - Tutoring starts from Wednesday, September 7

- Labs from now on – go to your lab as in schedule
  - Bring your laptop

- First programming assignment will be ready today

- Reading assignment for this week: Chapter 2 of Liang
  - Note that my lecture notes are not following in the exact same order of topics as in the textbook
Variables

• You can think of variables as mailboxes that store data in the computer’s memory (RAM)

  • Like mailboxes, variables have a unique number or memory address associated
    • Usually, we are not interested in the specific numerical address

  • Like mailboxes, variables have a name associated with it and we are definitely interested in using this name

  • Small mailboxes cannot hold big packages
    • This applies to variables too – variables can only hold certain types of data
Variable naming

- Usually, the first letter is not capitalized but later words are capitalized, e.g.,
  - x, count, myAccount, studentID, . . .
  - Note: different than class or method naming conventions

- Use informative names: bookTitle is much more informative than just b or bt

- Remember identifier naming rules:
  - Only use letters, numbers, and underscores in variable names
Declaring variables

- Before you can use a variable, you must first declare it with name AND what kind of data it holds
  - In other words, a variable has an associated data type
  - General format: `<type> <variable_name>;
    int count;  // holds integer values
    double rate;  // holds double precision real values

- Declaring a variable, sets aside memory for that variable
  - Variables of different types utilize different amounts of memory

- Variables can hold primitive types (next slide) or refer to objects (coming later)

- You can declare variables anywhere inside any block (class, method, for now)
Java’s 8 primitive types

- 4 data types represent integers (whole numbers):
  - byte, short, int, long

- 2 data types represent floating point numbers (real numbers):
  - float, double

- 1 data type represents characters such as an ‘A’:
  - char

- 1 data type represents boolean (can only be true false):
  - boolean

- In CSE 114, we will mostly use the ones in blue
Bits and bytes

- Underneath the hood, all data in a computer is represented using bits and bytes

- A bit represents a binary digit (1 or 0)
  - A single bit can represent 2 values
  - 2 bits can represent 4 values
  - 3 bits can represent 8 values
  - k bits can represent $2^k$ values
  - Each bit doubles the number of values

- A byte is 8-bits and can represent $2^8$ values
  - For unsigned values, the range is 0 to $2^8 - 1$
  - For signed values, the range is -128 to 127

- The size for integers is 4 bytes which can represent $2^{32}$ values
- The size for longs is 8 bytes which can represent $2^{64}$ values
# Data type memory size

The data types take up different amounts of space in memory. In Java:

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>&lt; -9x10^{18}</td>
<td>&gt; 9x10^{18}</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>+/- 3.4x10^{38} with 7 significant digits*</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>+/- 1.7x10^{308} with 15 significant digits*</td>
<td></td>
</tr>
</tbody>
</table>

*: IEEE 754 format
A programmer walks into a bar... (self.ProgrammerHumor)
submitted 7 months ago by rustyshaklferd

and asks the bartender for 1.000000000000003123939 root beers. Bartender says, I'll have to charge you extra, that's a root beer float. Programmer says, better make it a double.
Literals

- **literal**: a value that is ‘literally’ written into a program’s code

  - “hi, there” (string literal)
  - 23 (integer literal)
  - 3.14 (floating point literal)
  - ‘A’ (character literal)
Variable initialization

- Not only does a variable need to be declared before it can be used, a variable must also be initialized

  ```java
  int n;
  double x;
  System.out.println(n);  // error: n is uninitialized
  ```

- You can initialize a variable with it is declared

  ```java
  int n = 3;   // 3 is an integer literal
  double x = 2.3;  // 2.3 is a double precision literal
  ```

- You can change the value of a variable later by assigning to it

  ```java
  n = 94;
  ```

- When a variable is referenced in a program, its current value is used

  ```java
  System.out.println(n);    // output: 94
  ```
Literals and variables Example

```java
class CookieMonster {
    public static void main(String[] args) {
        int cookies;
        cookies = 22;
        System.out.print("Cookies per box: ");
        System.out.println(cookies);
    }
}
```

- Program output:
  Cookies per box: 22

- Notice the use of print instead of println
  - print leaves the current print position on the same line whereas println always adds a line break at the end
Variable assignment

• A variable can be assigned or reassigned a value using the assignment operator: 

\[ \text{<variable> = <expression>;} \]

cookies = 22;

• “=” does not indicate equality! This is not algebra!
  • Think of assignment as an “arrow”:
    
    \[ \text{a = 22 \hspace{1cm} think of as: \hspace{1cm} a \leftarrow 22} \]

• Assignment means the value of the expression on the right side of the “=” is assigned to the variable on the left
  • Assignment proceeds right to left
    
    \[
    \begin{align*}
    \text{int n = 3;} & \quad \text{// initialization} \\
    \text{x = 9.3;} & \quad \text{// x is assigned the value of 9.3} \\
    9.3 = y; & \quad \text{// error} \\
    i = i + 1; & \quad \text{// i assigned the value of i plus 1}
    \end{align*}
    \]
Assignment type matching

- In general, you can only assign/initialize a value to a variable that is consistent with the variable’s declared type
  
  ```java
  int i, n;
  n = 7;   // ok
  i = 2.5; // error
  ```

- However:
  
  ```java
  double x, y;
  y = 4.3;  // ok
  x = 32;  // ok, not an error! Why?
  ```

- In the `x = 32` case, Java converted the literal integer 32 to a literal double 32.0 and assigned it to `x`

- Why didn’t a similar type conversion happen in the first case?
Type conversion

- *Type conversion* refers to changing a value’s type, e.g.,
  - `int` into `double` or `double` into `int`

- In general, Java will perform implicit or automatic type conversion. . .
  - If there is no loss of information or precision

- Java will automatically convert from one type to another type that is “wider” in terms of precision and/or range
  - `int` to `float` (or `double`) but not the other way around
Type casting

• You can explicitly force type conversion using a technique called type casting
  
  ```java
  int m = (int) 4.32;
  
  Here, truncation occurs and 4 is assigned to m
  ```

• Casting is a convenient but a potentially dangerous tool
  
  • Circumvents some of the error-checking that Java does to prevent run-time errors
Variable scope (next)

- All variables exist and are visible in a certain “region”. This is known as a variable **scope**.
- Basically, variables only exist inside the block of code in which the variables are declared.

```java
public static void main(String[] args) {
    int i = 4;
    {  int n = i;  // i is still in scope
        System.out.println(n);
    }
    n = 100;  // error: n is not in scope here
}
```

- When the program flow “exits” a block of code, all the variables declared inside that block go out of scope and are no longer visible/accessible in the outer block.
Constants

- What if you don’t want assignment to occur after a variable has been initialized?
  - In other words, you want a variable to remain unchanged or constant

- To enforce this, use the `final` keyword:
  ```java
  final int CELL_COUNT = 100;
  final double PI = 3.14159;
  ```

- Later, you’ll get a compile-time error if you try to reassign a value to a constant
  ```java
  CELL_COUNT = 1000;
  ```

- Convention is to name constants all-caps
Expressions

- Think of expressions as a combination of operators, literals, and variables
  - that ultimately *evaluate* to a single value

4
3 * 3.14
x + 2
y / (x * 3)
Pi * radius * radius
Expression vs. statement

- Represents something
  - Java evaluates it
  - End result is a value

- Examples
  - 23
  - 4.3215
  - \((4 - 32 \times 3) / 2.3\)

- Does something
  - Java executes it
  - Need not result in a value (but can!)

- Examples
  - \(\ldots\)println(“Hi, there”)
  - \(x = x + 34\)
Arithmetic operators

- Java supports the basic arithmetic operations using the following operators:
  - Addition:  
  - Subtraction:  
  - Multiplication:  
  - Division:  
  - Modulo (remainder):  

- As a program runs, its expressions are evaluated
  - 1 + 3 evaluates to 4
  - System.out.println(4 – 2); prints 2
  - How would you print the text 4 – 2 instead?
Incrementing a variable

- Let’s see how incrementing a variable works
  ```
  int k = 8;
  k = k + 3;
  ```

- One simple addition statement is really three simpler steps where the last step is assignment via “=”

  1. Read value of k from memory
  2. Add 3 to loaded value
  3. Write sum back to k
A surprising operation (integer division with `/`)

- When we divide numbers in the real world, there’s usually no problem with precision. However, dividing numbers in code can sometimes surprise you:
  
  \[
  8 \div 4 \rightarrow 2 \\
  5 \div 2 \rightarrow 2 \quad \text{what?!!} \\
  5.0 \div 2 \rightarrow 2.5 \\
  (\text{double})5/2 \rightarrow 2.5
  \]

- The type of the operands determines whether integer division or floating-point division is performed
  
  - An integer divided by an integer uses integer division which results in truncation of the fractional part

  ```java
double x = 1/2;
System.out.println(x); // What’s printed? (Surprise!)
```
Integer remainder with %

- The % operator computes the remainder from integer division
  
  \[
  \begin{align*}
  15 \% 4 & \quad \text{is} \quad 3 \\
  211 \% 10 & \quad \text{is} \quad 1
  \end{align*}
  \]

- Useful applications of % operator

  - Obtain last digit of a number: \(234325 \% 10\) is 5
  - Obtain last 3 digits: \(3267345 \% 1000\) is 345
  - See whether a number is odd: \(9 \% 2\) is 1, \(48 \% 2\) is 0

What is the result?

- \(43 \% 5\)
- \(5 \% 5\)
- \(3 \% 23\)
- \(12 \% 0\)
Compile-time error

- It occurs during compilation phase (`javac` command) of a program lifecycle

```java
public static void main(String[] args) {
    int x = 4;
    int y = 3;
    System.out.println(x + y + z);
}
```

- Compile-time errors are usually easy to find, particularly with a tool like Eclipse
Run-time error

- It occurs during execution phase (java command) of a program lifecycle
- Program crashes when it runs, e.g., divide by zero!
- Java tries to help with detecting some run-time errors

```java
public static void main(String[] args) {
    int x = 4;
    int y = 0;
    System.out.println(x/y);
}
```
Logic error

- Program runs and does not crash but there is an error in the program’s logic or algorithm
- Can be hard to find especially in complex programs
- Test your code!

```java
public static void main(String[] args) {
    int x = 4;
    int y = 10;
    System.out.println(x/y);  // You may’ve meant to say x * y instead
}
```