Why do we need variables?

Why do we need variables?

2+2

3+21

−

×

÷

Java Basics:
Variables & Data Types

Variables and Values

- *Variables* store data such as numbers and letters
  - Think of them as places to store data
  - They are implemented as memory locations
- The data stored by a variable is called its *value*
  - The value is stored in the memory location
- Its value can be changed

Variables

- Variables have two attributes: a *name* and a *type*

- We will consider these two attributes one at a time. First let us consider names…

Java Identifiers

- An *identifier* is a name, such as the name of a variable, a method, or a class.

- Identifiers may contain only
  - letters (both lower & upper case)
  - digits (0 through 9)
  - and the underscore character (_)

  but the first character cannot be a digit.

Java Identifiers, cont.

- Identifiers may *not* contain any spaces, dots (`.`), asterisks (`*`), or other characters:

  `seven-eleven` `netscape.com` `util.*` (are not allowed)

- Java is *case sensitive*, thus `stuff`, `Stuff`, and `STUFF` are different identifiers

- Identifiers can be arbitrarily long

Keywords or Reserved Words

- Some words, such as `public`, are called *keywords* or *reserved words* and have special, predefined meanings
- Keywords cannot be used as identifiers
- Other keywords: `static, void, class`
  - We will be introduced to many more

Naming Conventions

Java uses the following conventions:

- Variable names and method names, regardless of their type, begin with a lowercase letters (e.g. `myName`, `processData`)
  - Multiword names are “punctuated” using uppercase letters (usually called “camel case”)
- Class names begin with an uppercase letter (e.g. `String` or `PrintStream`)
- Names should be meaningful and/or descriptive

Variables

- Variables have two attributes: a *name* and a *type*
- Next let us consider types…

Data types

- Internally, computers store everything as 1s and 0s

```
154    → 0000000010011010
"hi"   → 0110100001101001
```

- We need to tell the computer how to interpret a given sequence of 1s and 0s, and how to operate on it

- A *type* defines a category or set of data values
  - It also defines the operations that can be performed on the data

Java's Primitive Types

- Four integer types (`byte`, `short`, `int`, and `long`)
  - We will always use the `int` type
- Two floating-point types (`float` and `double`)
  - We will always use the `double` type
- One character type (`char`)
- One boolean type (`boolean`)

- Types that are not primitive are called *object types* (seen later)

Java's Primitive Types, cont.

Type Name	Kind of Value	Memory Used	Size Range
`byte`	*integer*	*1 byte*	–128 to 127
`short`	*integer*	*2 bytes*	–32768 to 32767
`int`	*integer*	*4 bytes*	–2147483648 *to* 2147483647
`long`	*integer*	*8 bytes*	–9223372036854775808 *to* 9223372036854775807
`float`	*floating-point number*	*4 bytes*	$\pm 3.40282347 \times 10^{+38}$ *to* $\pm 1.40239846 \times 10^{-45}$
`double`	*floating-point number*	*8 bytes*	$\pm 1.76769313486231570 \times 10^{+308}$ *to* $\pm 4.94065645841246544 \times 10^{-324}$
`char`	*single character (Unicode)*	*2 bytes*	*all Unicode characters*
`boolean`	`true` *or* `false`	*1 bit*	*not applicable*

Examples of Primitive Values

- Integer types

```
0   -1   365   12000
```

- Floating-point types

```
0.99   -22.0   3.14159   -0.25   9.4e3
```

- Character type

```
'a'   'A'   '#'   (use single quote mark)
```

- boolean type

```
true   false
```

Variables

- Variables have two attributes: a *name* and a *type*
- Now that we know about names and types, we can combine them to declare variables

Declaring Variables

- When you *declare* a variable, you provide its type and name:

```
int numberOfRings;
```

Declaring Variables

- When you *declare* a variable, you provide its type and name:

```
int numberOfRings;
```

Declaring Variables

- When you *declare* a variable, you provide its type and name:

```
int numberOfRings;
```

- A variable must be declared before it is used

- Choose names that are descriptive
  - Such as `numberOfRings` rather than `n`

Declaring Variables

- Syntax:

  type variable_1, variable_2, …;

- Examples (from the traveling circus):

```
int numberOfRings, attendance;
double ringDiameter, weightOfLion;
boolean lionWasFed;
```

Where to Declare Variables

- Declare a variable…
  - just before it is used for the first time, or
  - at the beginning of the section of your program that is enclosed in { }

```
public void process( )
{
  // declare variables here

}
```

Java Basics:
Variables and Assignment

Assignment Statements

- An assignment statement is used to assign a value to a variable

```
answer = 42;
```

- The “equal sign” is called the *assignment operator*
  - It does not mean “equality” in the mathematical sense

- We say, “`answer` is assigned the value 42.”

Assignment Statements, cont.

- Syntax:

  variable = expression;

  where ***expression*** can be another variable, a *constant* (such as a number), or something more complicated which combines variables and constants using *operators* (more on this in a moment).

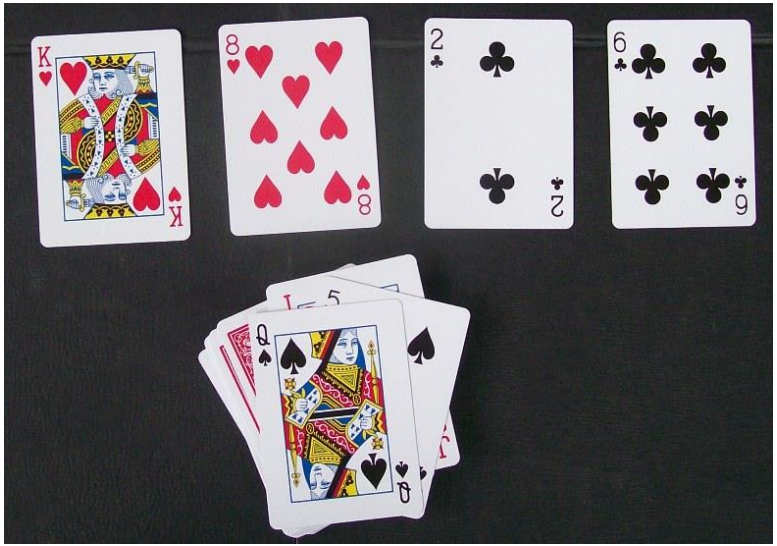
  The variable is always on the left and the expression is always on the right.

Assignment Examples

```
numOfStacks = 4;

topCard = 'Q';

points = 10 * cardsPlayed;
```



Assignment Evaluation

1. The expression on the right-hand side of the assignment operator (=) is evaluated **first**

Assignment Evaluation

1. The expression on the right-hand side of the assignment operator (=) is evaluated **first**

2. **Then** the result is used to set the value of the variable on the left-hand side of the assignment operator

Assignment Evaluation

1. The expression on the right-hand side of the assignment operator (=) is evaluated **first**

2. **Then** the result is used to set the value of the variable on the left-hand side of the assignment operator

```
cardsInHand = cardsInHand - 1;
```

Before assignment →

Assignment Evaluation

1. The expression on the right-hand side of the assignment operator (=) is evaluated **first**

2. **Then** the result is used to set the value of the variable on the left-hand side of the assignment operator

```
cardsInHand = cardsInHand - 1;
```

After assignment →

Using variables

- Once given a value, a variable can be used in expressions, and can assign a value more than once :

```
int x, y, z;
x = 3;              // x is 3
y = 2 * x;          // y is 6
z = 5 + x*y;        // z is 23
x = 4 + y;          // x is now 10
...
```

Java Basics: Expressions

Expressions

- An *expression* is value or operation that computes a value
  - Examples:

```
5 + 14
8 / 2 * (4 + 3)
17
```

- The simplest expression is a *literal value*
  - Such as the literal value 17 above
- A complex expression can use operators and parentheses

Arithmetic Operators

- Arithmetic expressions can be formed using *operators* which allow us to combine values referred to as the *operands*

- Arithmetic operators include:

  `+` addition
  `-` subtraction (or negation)
  `*` multiplication
  `/` division
  `%` modulus (a.k.a. mod or remainder)

The Division Operator

- The division operator (`/`) behaves as expected if one of the operands is a floating-point type

- When both operands are integer types, the result is truncated, not rounded, to produce an integer value
  - Thus, `14/4` produces the value `3`

- If you want a floating-point result, at least one operand must be a floating-point type
  - Both `14/4.0` and `14.0/4` produce the value `3.5`

The `mod` Operator

- The mod (%) operator is used with operands of integer type to obtain the remainder after integer division

- 14 divided by 4 is 3 *with a remainder of 2*
  - Hence, `14%4` produces the value `2`

- The mod operator has many uses, including
  - determining if an integer is odd or even
  - determining if one integer is evenly divisible by another integer
  - obtaining the lower-ordered digits of a number

Parentheses and Precedence

- As in algebra, parentheses can communicate the order in which arithmetic operations are performed
- examples:

```
(10 + 213) * 37
10 + (213 * 37)
```

- Without parentheses, an expression is evaluated according to the *rules of precedence*

Precedence Rules

Highest Precedence

First: the unary operators: +, –, ++, – –, and !

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and –

Lowest Precedence

Precedence Rules

- When binary operators have equal precedence, they are evaluated left-to-right.

`1+2-3+4` is the same as `((1+2)-3)+4`

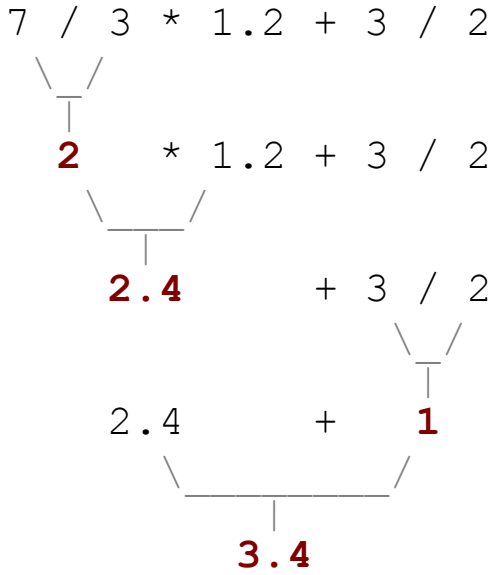
- When unary operators have equal precedence, they are evaluated right-to-left.

Mixing types

- When `int` and `double` are mixed, the result is a `double`

  `4.2 * 3` is `12.6`

- The conversion is per-operator, affecting only its operands



Java Basics:
More on Assignment

Specialized Assignment Operators

- Assignment operators can be combined with arithmetic operators (including `+`, `-`, `*`, `/`, and `%`)

```
time = time + 10;
```

can be written as

```
time += 10;
```

yielding the same results

Specialized Assignment Operators

Shorthand	Equivalent longer version
variable += expr;	**variable = variable + (expr);**
variable -= expr;	**variable = variable - (expr);**
variable *= expr;	**variable = variable * (expr);**
variable /= expr;	**variable = variable / (expr);**
variable %= expr;	**variable = variable % (expr);**

Increment (and Decrement) Operators

- used to increase (or decrease) the value of a variable by 1
- easy to use, important to recognize
- the increment operator is ‘++’

  `count++` or `++count`

- the decrement operator is ‘--’

  `count--` or `--count`

Increment (and Decrement) Operators

- Equivalent statements:

```
count++;
++count;
count = count + 1;
count += 1;
```

- Also:

```
count--;
--count;
count = count - 1;
count -= 1;
```

Increment (and Decrement) Operators

- The increment operator also produces a value
- Pre-increment: `++count`
  - Increments the variable and returns the new value
- Post-increment: `count++`
  - Increments the variable but returns the original value
- Most often, we use these to simply increment a variable and we don't care about the return value
  - In which case, you can use either one
  - This occurs when increment is the only operation in a statement

Initializing Variables

- A variable that has been declared, but not yet given a value is said to be *uninitialized*
- The compiler will not allow you to use an uninitialized variable
- To protect against an uninitialized variable (and to keep the compiler happy), it is good practice to assign a value at the time the variable is declared

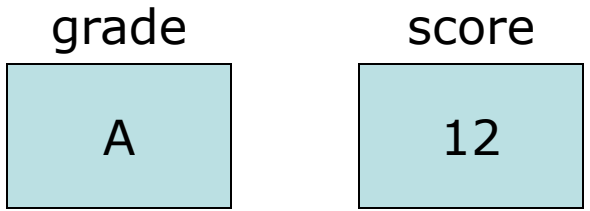
Declaration & initialization

- A variable can be declared & initialized in one statement

- Syntax:

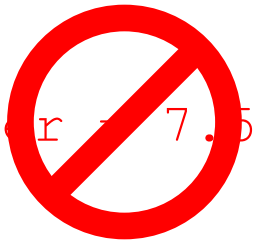
```
type name = value;
```

```
char grade = 'A';
int score = (14 % 4) + 10;
```



Assignment Compatibilities

- Java is said to be *strongly typed*
  - You can't, for example, assign a floating point value to a variable declared to store an integer



```
int myNumber = 7.5; // Error: Compiler will not allow
```

Assignment Compatibilities

- Sometimes conversions between numbers are possible

```
double myVariable = 7;  // this is OK
```

  It is okay to assign an integer value to a variable of type `double`

  In this case, the compiler will automatically convert the integer `7` into a floating point `7.0`

- This automatic conversion is called a *coercion*

Assignment Compatibilities

- A value of one type can be assigned to a variable of any type further to the right

  ```
  byte --> short --> int --> long --> float --> double
  ```

  but not to a variable of any type further to the left.

- E.g., you can assign a value of type `char` to a variable of type `int`, or a value of type `int` to a variable of type `double`, but you cannot assign a value of type `double` to a variable of type `int`.

Type Casting

- A *type cast* temporarily changes the *value* of a variable from the declared type to some other type. It does **not** change the variable.

- For example,

```
double distance;
distance = 9.0;
int points;
points = (int)distance;
```

  the above is illegal without the `(int)`

Type Casting, cont.

- The value of `(int)distance` is `9`, but the value of `distance`, both before and after the cast, is `9.0`

- Any nonzero value to the right of the decimal point is *truncated* rather than *rounded*
  - Thus if the value of `distance` was `9.7`, the value of the expression `(int)distance` would still be `9`
  - Again, the value of the variable `distance` is not changed and would still be `9.7`