CS 241 — Introduction to Problem Solving and Programming

Fundamentals of Programming

Primitive types, Strings, and operators

Jan 14, 2005

Outline/overview

- Types
- Arithmetic
- Expressions vs. statements
- Operators
- Strings

Variables and types

Recall this example:

```
public class Variable {

    public static void main (String[] args) {

        int number;
        number = 5;
        System.out.println("Here is a number: " + 5);
        System.out.println("Here is the number again: " + number);

    }
}
```

Variables and types

Look more carefully at the declaration.

```
int number;
```

- A declaration gives information about the variable.
- `int` says that this variable is used to store integers.
- This kind of information is called the variable's type.

Primitive types

These types are called primitive types.

⋆ `int` for integers

`short` for small integers, using less memory

`long` for big integers, using extra memory

`float` for real numbers in scientific notation

⋆ `double` for more precise real numbers, using extra memory

⋆ `char` for typographic characters (letters, digits, punctuation...)

⋆ `boolean` for booleans (truth values)

`byte` for bytes of memory

Types

The code

```
int number;
number = 7.3;
```

will be rejected by the compiler.

```
ar1121: 177 javac Variable.java
Variable.java:7: possible loss of precision
found   : double
required: int
        number = 7.3;
                 ^
1 error
```

Two declaration shortcuts

A variable's declaration and initialization can be combined:

```
        int x = 47;
```

Variables of the same type can be declared together (note the comma):

```
        int x, y;
```

Arithmetic

Now try something more interesting:

```
public class Sum {
    public static void main (String[] args) {

        int first, second, sum;

        first = 5;
        second = 8;

        System.out.println("The numbers are " + first
                           + " and " + second);

        sum = first + second;

        System.out.println("Their sum is " + sum);
    }
}
```

```
..

The numbers are 5 and 8
Their sum is 13
```

Printing text

Notice this statement:

```
System.out.println("The numbers are " + first
                   + " and " + second);
```

- Several parts of text may be joined together.
- We can go to the next line in the source program.
- We need to add spaces in the quotes for it to look nice.

Arithmetic

More interesting is

```
        sum = first + second;
```

- The plus sign is used for addition (surprise).
- Adding two `int`s produces an `int`.
- The result can be stored in an `int`-typed variable.

Expressions and statements

An expression is programming language construct that has a value (or returns a value, or evaluates to a value).

A statement is a programming language construct that has no value but is executed for its effect.

`first + second`	*expression*	value: 5
`sum = first + second;`	*statement*	effect: `sum` given value 5

A semi-colon makes an expression a statement.

Syntax forms

Some syntax forms we know:

Declaration:	*Type Variable, Variable, . . . ;*
Assignment:	*Variable = Expression;*
AdditionExpression:	*Expression + Expression*

Arithmetic and types

```
public class Quotient {
    public static void main (String[] args) {

        int first, second, quotient;
        first = 23;
        second = 4;

        System.out.println("The numbers are " + first
                           + " and " + second);

        quotient = first / second;

        System.out.println("Their quotient is " + quotient);
    }
}
```

What's the output?

Arithmetic and types

```
public class Quotient {
    public static void main (String[] args) {

        int first, second, quotient;
        first = 23;
        second = 4;

        System.out.println("The numbers are " + first
                           + " and " + second);

        quotient = first / second;

        System.out.println("Their quotient is " + quotient);
    }
}
```

..

```
ar1121: 198 javac Quotient.java
ar1121: 199 java Quotient
The numbers are 23 and 4
Their quotient is 5
```

Arithmetic and types

Why?

```
first / second
```

This performs integer division ; both subexpressions are `int`s, the result is an `int`, and the variable storing the result is an `int`.

Arithmetic and types

```
public class Quotient {
    public static void main (String[] args) {

        int first, second;
        quotient;
        first = 23;
        second = 4;

        System.out.println("The numbers are " + first
                           + " and " + second);

        quotient = first / second;

        System.out.println("Their quotient is " + quotient);
    }
}
```

How about this?

Arithmetic and types

```
public class Quotient {
    public static void main (String[] args) {

        int first, second;
        quotient;
        first = 23;
        second = 4;

        System.out.println("The numbers are " + first
                           + " and " + second);

        quotient = first / second;

        System.out.println("Their quotient is " + quotient);
    }
}

..

ar1121: 206 javac Quotient.java
ar1121: 207 java Quotient
The numbers are 23 and 4
Their quotient is 5.0
```

Arithmetic and types

Why?

```
   quotient = first / second;
```

This still performs integer division, the result is merely stored in a double variable.

The `int` 5 is converted or cast to the `double` 5.0.

Arithmetic and types

```
public class Quotient {
    public static void main (String[] args) {

        double first, second, quotient;
        first = 23;
        second = 4;

        System.out.println("The numbers are " + first
                           + " and " + second);

        quotient = first / second;

        System.out.println("Their quotient is " + quotient);
    }
}

...

ar1121: 213 javac Quotient.java
ar1121: 214 java Quotient
The numbers are 23.0 and 4.0
Their quotient is 5.75
```

Arithmetic and types

- An operator is a symbol (usually based on punctuation characters) that performs an operation that is built into the language.
- Values given to the operator are called operands.
- What operators do depends on the types of their operands.

Arithmetic and types

What if we want to treat an integer as a real number? We can convert it:

```
        int first, second;
        double quotient;
        first = 23;
        second = 4;

        quotient = (double) first / (double) second;

...

The numbers are 23 and 4
Their quotient is 5.75
```

This is called type casting (also type promotion or type coercion).

Automatic type casting

Sometimes this happens automatically.

```
        int first, second;
        quotient;

        quotient = first / second;

..

The numbers are 23 and 4
Their quotient is 5.0
```

Casts happen automatically when converting from less memory/precision to more memory/precision. Compatibility chain:

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

Type casting

The compiler accepts the first and rejects the second:

```
        double x = 5;
        int y = 5.2;
```

```
...

Program.java:6: possible loss of precision
found   : double
required: int
        int y = 5.2;
                ^
1 error
```

Arithmetic operators

+	Addition	`int` and `double`
-	Subtraction	`int` and `double`
*	Multiplication	`int` and `double`
/	Division	`int` and `double`
%	Modulus (remainder)	`int`

Modulus operator

```
        int first, second, quotient, remainder;
        first = 23;
        second = 4;
        quotient = first /  second;
        remainder = first % second;
        System.out.println(first + " / " + second + " = " + quotient
                           + " R " + remainder);
```

```
..

23 / 4 = 5 R 3
```

Assignment chaining

An assignment is an expression— although it has a side effect, it also has a value.

Variable = *Expression*

Side effect. Store the value of *Expression* in *Variable*.

Value. Return the value of *Variable*.

This means we can chain assignments:

```
y = x = 5;
```

This makes both `x` and `y` equal to 5.

Assignment and arithmetic

Assignment shorthands:

x += n		x = x + n
x -= n		x = x - n
x *= n		x = x * n
x /= n		x = x / n
x %= n	means	x = x % n
x++		x = x + 1 but return old x
++x		x = x + 1
x--		x = x - 1 but return old x
--x		x = x - 1

Unary (one operand) operator:

-x	negates x.

Arithmetic operators

ArithmeticExpression: Expression BinOp Expression

You may combine expressions into arbitrarily long expressions.:

```
y += x = 2 + 3 * 5 - 2;
```

The value and effects of these expressions and statement depend on

Precedence. Which operators are executed first (mathematical *order of operation*).

Associativity. What order operators of equal precedence are executed (left associative: left-to-right; right associative: right-to-left).

Precedence and associativity

Operators we have seen so far.

Precedence		**Associativity**
Highest precedence	++, --, unary -, and type casting	Right associative
	*, /, and %	Left associative
	+ and -	Left associative
Lowest precedence	= and friends	Right associative

Precedence and associativity

```
y += x = 2 + 3 * 5 - 2;
```

Precedence and associativity

```
y += x = 2 + 3 * 5 - 2;

y += x = 2 + 15 - 2;
```

Precedence and associativity

```
y += x = 2 + 3 * 5 - 2;

y += x = 2 + 15 - 2;

y += x = 17 - 2;
```

Precedence and associativity

```
y += x = 2 + 3 * 5 - 2;

y += x = 2 + 15 - 2;

y += x = 17 - 2;

y += x = 15;
```

Precedence and associativity

```
y += x = 2 + 3 * 5 - 2;

y += x = 2 + 15 - 2;

y += x = 17 - 2;

y += x = 15;

y += 15;
```

Parentheses

To override precedence rules, use parentheses.

(Parentheses together make an operator which has the highest precedence)

Java: 43 operators, 14 precedence classes.

Don't memorize. . . remember a few obvious ones and use parenthesis when in doubt.

Characters

A `char` is any single letter, digit, punctuation, or anything you would make with a keystroke.

A literal `char` value must be enclosed in single quotes.

```
        char aChar;
        aChar = ’A’;
        System.out.println("A character: " + aChar);
```

```
...

A character: A
```

Characters

What if you want to store a single quote itself?

Use an escape sequence– a backslash followed by a special character.

```
        char aChar;
        aChar = ’\’’;
        System.out.println("A character: " + aChar);
```

```
...

A character: ’
```

Characters

Commonly used escape sequences:

Sequence	**Description**
\´	Single quote
\"	Double quote
\ n	New line
\ t	Tab

Strings

A block of several characters is called a string.

To model strings, Java has a type `String`.

`String` is not a primitive type. Later, when we look at classes, we'll see that it is actually a class.

String

You can declare variables of type String. Literals are enclosed with double quotes.

```
        String greeting;
        greeting = "aloha, ahoy, bon jour, salve, ni hao";
        System.out.println(greeting);
```

...

```
ar1121: 256 java FirstString
aloha, ahoy, bon jour, salve, ni hao
```

Notice how greeting is used in println.

Strings

The use of the plus we've seen is called concatenation.

concatenate. To link together as in a series or chain. (Merriam-Webster.) From Latin *catena*, chain.

```
String greeting, message;
greeting = "aloha";
System.out.println(greeting + " ahoy");
greeting = greeting + " salve";
System.out.println(greeting);
message = "ni hao";
greeting += message;
System.out.println(greeting);
```

Strings

```
        String greeting, message;
        greeting = "aloha";
        System.out.println(greeting + " ahoy");
        greeting = greeting + " salve";
        System.out.println(greeting);
        message = "ni hao";
        greeting += message;
        System.out.println(greeting);
```

```
...

aloha ahoy
aloha salve
aloha salveni hao
```

Note that you must put spaces explicitly where you want them.

Strings

What's really happening here?

```
System.out.println("Here is a number: " + 5);
```

Strings

What's really happening here?

```
System.out.println("Here is a number: " + 5);
```

When plus is used with at least one `String`, it is interpreted as concatenation, and the other value is automatically cast to `String`.

Strings

Strings have methods (something we'll learn about in a couple weeks) which define operations on them.

For example:

```
greeting.length()
```

Calculates the length (number of characters) in the string stored in variable `greeting`.

Strings

```
String greeting = "aloha and ahoy!";
int greetingLength = greeting.length();
System.out.println("\"" + greeting + "\" is " + greetingLength
                   + " characters long.");
```

```
...

"aloha and ahoy!" is 15 characters long.
```

- Make sure you understand what we did with slashes and quotes.
- Note that spaces and punctuation are included in the count.
- Note that `length()` returns an `int` when it is called. This is its return type.

String methods

There are methods to convert a String to all lower case or all uppercase.

```
        String virgil = "Arma virumque cano Trojae qui primus ob oris";
        String lowerCase = virgil.toLowerCase();
        String upperCase = virgil.toUpperCase();

        System.out.println(virgil);
        System.out.println(lowerCase);
        System.out.println(upperCase);

...

Arma virumque cano Trojae qui primus ob oris
arma virumque cano trojae qui primus ob oris
ARMA VIRUMQUE CANO TROJAE QUI PRIMUS OB ORIS
```

String methods

Note:

```
        String lowerCase = virgil.toLowerCase();
```

- The return type of `toLowerCase()` is `String`.
- The contents of the variable `virgil` is unchanged.

String methods

`trim()` removes leading or trailing whitespace.

```
        String message = "         \n    O nuntii mihi beati!       ";
        String trimmedMessage = message.trim();

        System.out.println("<" + message + ">");
        System.out.println("<" + trimmedMessage + ">");
```

```
...

<
    O nuntii mihi beati!       >
<O nuntii mihi beati!>
```

String

A String is represented as an ordered sequence of characters indexed starting at zero.

`"dux femina facti"`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d	u	x		f	e	m	i	n	a		f	a	c	t	i

We'll find that indexing from zero is true for other data structures . . .

String

`charAt(`*position*`)` returns a char at a given position.

```
        String message = "timidumque ad lumina lumen attolens";
        char letter8 = message.charAt(8);
        char letter16 = message.charAt(16);

        System.out.println(message);
        System.out.println(letter8 + "   " + letter16);
```

```
...

timidumque ad lumina lumen attolens
u   m
```

String

`substring(...)` returns a `String` that is part of the `String` on which it is called.

- Given one `int`, it interprets it as the starting point and returns the string from there to the end.
- Given two `int`s, it interprets them as the starting and ending points.

```java
String message = "Varus me meus ad suos amores";
String subMessage1 = message.substring(22);
String subMessage2 = message.substring(6, 8);

System.out.println(message);
System.out.println(subMessage1);
System.out.println(subMessage2);
```

```
...

Varus me meus ad suos amores
amores
me
```

Later we'll see that this is an instance of overloading a method. . .

Strings

Note that the second index refers to *one past* the last item in the range.

0	1	2	3	4	5	6	7	8	9
q	u	o	d	c	u	m	q	u	e
				↑			↑		

```
String message = "quodcumque";
String subMessage = message.substring(4,7 );

System.out.println(message);
System.out.println(subMessage);
```

```
...

quodcumque
cum
```

String

A `String` variable can change, but a `String` itself cannot (it is immutable).

```
        String message1 = "o fortunati quorum moenia iam surgunt";
        String message2 = message1;
        message1 += ".";

        System.out.println("message1: " + message1);
        System.out.println("message2: " + message2);
```

```
...

message1: o fortunati quorum moenia iam surgunt.
message2: o fortunati quorum moenia iam surgunt
```

Summary

Be able to identify the following concepts:

- Type
- `int`, `double`, and `char`
- Expression
- Statement
- Operator and operands
- Integer division
- Modulus
- Type cast
- Assignment shorthands and increment/decrement
- Precedence
- Associativity
- Escape sequence
- `String`
- Concatenation
- String methods
- Indexing from zero