COMP112/18 - Programming I

19 Review

Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP112/18/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

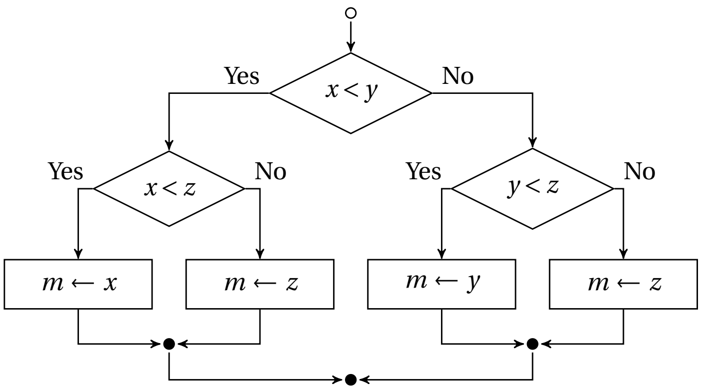
November 30, 2018

Outline

Flowchart Programs

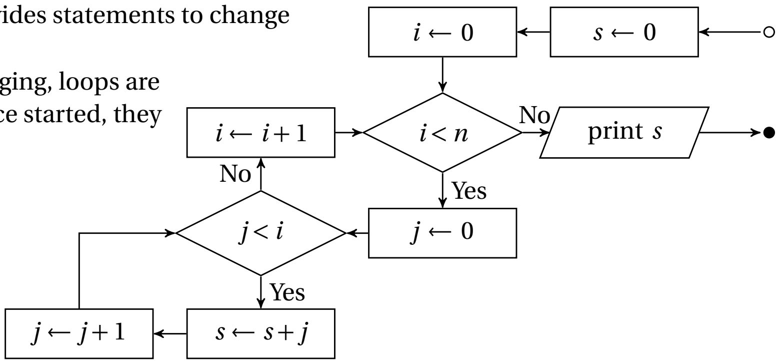
- A program can be expressed by a flowchart. Flowchart programs and Java programs can be translated into each others.

```
if ( x < y )
  if ( x < z )
    m = x;
  else
    m = z;
else
  if ( y < z )
    m = y;
  else
    m = z;
```



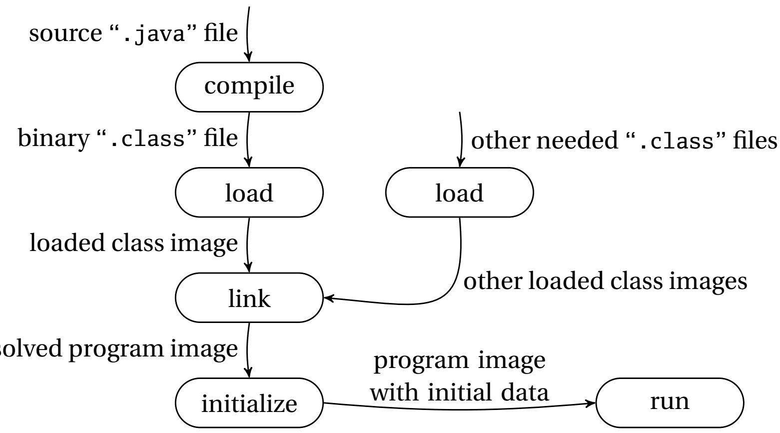
States and Loops

- Each variable contains a value, all the variables and their associated values at a certain moment is called a *state* of a program. If we change any variable, the state is changed.
- *Imperative programming* is a programming paradigm that provides statements to change a program's state.
- Without state changing, loops are meaningless — once started, they will never end.



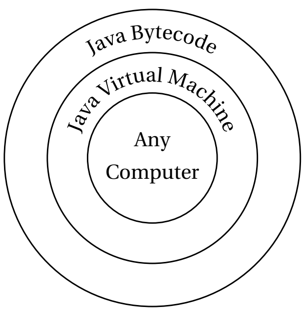
Java Compilation and Execution Process

This diagram shows how a Java program is processed from the source form.



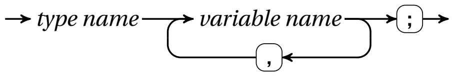
Java Virtual Machine

- A program in Java must be *compiled* to Java *bytecode* to run on the Java Virtual Machine (JVM).
- The bytecode is similar to machine instructions but is *architecture neutral*.
- A bytecode program can run on any platform that has a JVM.
- Java bytecode is usually *interpreted* by the JVM.
- A JIT compiler compiles a segment of bytecode when it is about to be *executed* (hence the name "just-in-time"), and then caches and reuses the result later without recompiling.



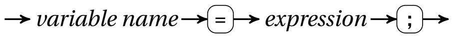
Using Variables

- **Declaration**. Before a variable can be used, we must declare it and give it a type.




```
int x, y;         // Declare x and y to be two integer variables.
double radius;    // Declare radius to be a double variable.
char a;           // Declare a to be a character variable.
```

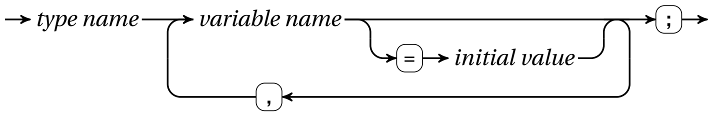
- **Assignment**. We can assign a value to a variable, the value can be a constant, or a value taken from another variable, or the result of an *expression*.




```
x = 1;                 // Assign 1 to x.
radius = 1.0;          // Assign 1.0 to radius.
a = 'A';               // Assign 'A' to a.
radius = radius*2;     // Double the radius.
```


Variable Initialization and Named Constants

- **Variable initialization**. We can give an initial value to a variable when we declare it.




```
int x = 1;
double d = 1.4, g = 7.8;
```

- **Named constant declaration**. We may also declare variables that cannot be assigned with new values, these variables are called *named constants*.


```
final double PI = 3.14159;
final int SIZE = 3;
```


Primitive Data Types

Name	Range	Storage Size
`byte`	-2^7 to -2^7-1 (−128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15}-1$ (−32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31}-1$ (−2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63}-1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235×10^{38} to -1.4×10^{-45} Positive range: 1.4×10^{-45} to 3.4028235×10^{38}	32-bit IEEE 754
`double`	Negative range: $-1.7976931348623157\times10^{308}$ to -4.9×10^{-324} Positive range: 4.9×10^{-324} to $1.7976931348623157\times10^{308}$	64-bit IEEE 754
`boolean`	`false`, `true`	JVM-dependent
`char`	\u0000 (0) to \uFFFF (65535)	16-bit unsigned

Floating-Point Numbers

- Just like writing very large or very small numbers on paper with limited space, we divide a number into two parts: *significant digits* and *scale factors*, such as

$$1.1023 \times 10^{120} \qquad -7.3000 \times 10^{-302}.$$

- We use a fixed number of bits to represent real numbers of very large range with a fixed precision.

IEEE 754 32-bit single precision (`float`) $\pm 1.M \times 2^{E'-127}$

S	E'	M
sign bit	*8-bit excess-127 exponent*	*23-bit mantissa fraction*

IEEE 754 64-bit double precision (`double`) $\pm 1.M \times 2^{E'-1023}$

S	E'	M
sign bit	*11-bit excess-1023 exponent*	*52-bit mantissa fraction*

Operator Precedence

Precedence Class	Operator	Associativity
postfix	*expr*`++` *expr*`--`	
unary	`++`*expr* `--`*expr* `+`*expr* `-`*expr* `~` `!`	
multiplicative	`* / %`	left to right
additive	`+ -`	left to right
shift	`<< >>>`	left to right
relational	`< > <= >=`	left to right
equality	`== !=`	left to right
bitwise	`& ^ |`	left to right
logical AND	`&&`	left to right
logical OR	`||`	left to right
conditional	$expr_1$ `?` $expr_2$ `:` $expr_3$	right to left
assignment	`= += -= *= /= %=`	right to left

Integer Division and Remainder Operation

- Integer division truncates the result *towards zero* (the nearest integer between the result and zero).

`5/2` yields 2 and `-17/3` yields −5,
`5.0/2` yields a double value 2.5 and `-17.0/3` yields −5.666666666666667.

- The remainder operation returns the remainder of the division: `5 % 2` yields 1.
- In Java, the sign of the remainder agrees with the sign of the dividend, regardless the sign of the divisor.
- If the dividend is negative, the remainder is negative: −12 % 5 and −12 % −5 all yield −2.

Conditional Expression

- The conditional operator (`?:`) selects one of the two expressions to evaluate based on the result of the boolean expression:

  → *boolean expression* → `?` → $expression_1$ → `:` → $expression_2$ →

- If the boolean expression evaluates to `true`, then $expression_1$ that follows the (?) is evaluated as the value of the conditional expression. Otherwise, $expression_2$ that follows the (`:`) is evaluated. Only one of the expressions is evaluated.

```
x = 1; y = 2; x = x < y ? 10 : 5; // x becomes 10.
s = 100; d = 0; s = d != 0 ? s/d : 1; // s becomes 1.
```

- The (precedence of) conditional operator is higher than all assignment operators, lower than relational and logical operators. Also the conditional operator is right associative. Therefore, the statement

```
name = 1 <= day && day <= 5 ? "workday" : day == 6 ? "Sat" : "Sun";
```

  makes sense.

Boolean Operations

- **Negation (not)**. Negation returns the opposite of its operand. Negation is a unary operation. The negation operator in Java is `!`.

```
boolean b = !(1 < 5); // b becomes false.
```

- **Conjunction (and)**. Conjunction returns `true` only if both the operands are `true`. Conjunction is a binary operation. The conjunction operator in Java is `&&`.

```
boolean b = '0' <= c && c <= '9'; // b becomes true if c is a decimal digit.
```

- **Disjunction (or)**. Disjunction returns `false` only if both the operands are `false`. Disjunction is a binary operation. The disjunction operator in Java is `||`.

```
boolean b = 100 == 80 || 70 < 100; // b becomes true.
```

- Without parentheses, negations are evaluated first, then conjunctions, finally disjunctions.

Logical Operators and Short-circuit Evaluation

- **AND** → $boolean\ expression_1$ → `&&` → $boolean\ expression_2$ →

  If $boolean\ expression_1$ evaluates to `false`, the AND-expression is `false`, and $boolean\ expression_2$ is *not* evaluated at all; otherwise, if $boolean\ expression_1$ evaluates to `true`, $boolean\ expression_2$ is evaluated as the result of the AND-expression.

- **OR** → $boolean\ expression_1$ → `||` → $boolean\ expression_2$ →

  If $boolean\ expression_1$ evaluates to `true`, the OR-expression is `true`, and $boolean\ expression_2$ is *not* evaluated at all; otherwise, if $boolean\ expression_1$ evaluates to `false`, $boolean\ expression_2$ is evaluated as the result of the OR-expression.

- As with the conditional expression, to determine the result by partial evaluation is called *short-circuit* evaluation. Many useful expressions rely on short-circuit evaluation.

```
divisor != 0 && total/divisor < 5        salary == 0 || top/salary >= 10
```

Assignment Expressions

- Assignments are usually used as statements, however, they can also be expressions.
- An **assignment statement** is actually an *assignment expression* followed by a semicolon (`;`).
- The evaluation of an assignment expression has a *side effect* of changing some variable, and the value of the assignment expression is exactly the value assigned to the variable.
- The following statement assigns 10 to x, y and z. The assignment operator is right associative.

```
x = y = z = 10;
```

  This is equivalent to

```
x = (y = (z = 10));
```

- Usually, assignment expressions are parenthesized due to the low precedence.

```
int lg = 0;
while ( (x = x/10) > 0 ) ++lg;  // computes the integer part of lg(x).
```


Augmented Assignment Operators and Self-Increment/Decrement

- Augmented assignment operators.

Operator	Example	Equivalent to
`+=`	`i += 8`	`i = i + 8`
`-=`	`f -= 8.0`	`f = f - 8.0`
`*=`	`i *= 8`	`i = i * 8`
`/=`	`i /= 8`	`i = i / 8`
`%=`	`i %= 8`	`i = i % 8`

- Increment and Decrement Operators.

Operator	Name	Description
`++var`	preincrement	The expression (`++var`) increments *var* by 1 and evaluates to the new value in *var* after the increment.
`var++`	postincrement	The expression (`var++`) evaluates to the original value in *var* and increments *var* by 1.
`--var`	predecrement	The expression (`--var`) decrements *var* by 1 and evaluates to the new value in *var* after the decrement.
`var--`	postdecrement	The expression (`var--`) evaluates to the original value in *var* and decrements *var* by 1.

Numeric Type Conversion and Casting

- **Conversion Rules**. When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:
  1. If one of the operands is `double`, the other is converted into `double`.
  2. Otherwise, if one of the operands is `float`, the other is converted into `float`.
  3. Otherwise, if one of the operands is `long`, the other is converted into `long`.
  4. Otherwise, both operands are converted into `int`.
- Implicit type casting:

```
double d = 3;   // type widening
```

- Explicit type casting:

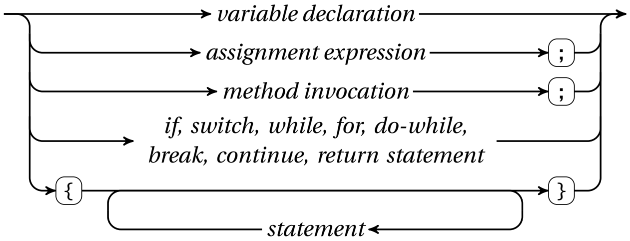
```
int i = (int)3.0;   // type narrowing
int i = (int)3.9;   // fraction part is truncated
```

- The range of a data type increases in the following order:

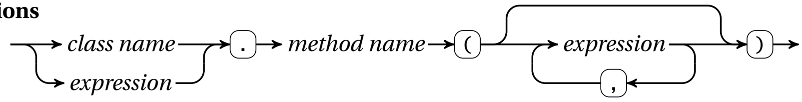
  `byte, short, int, long, float, double.`

Syntax Diagram of Statements

Statements

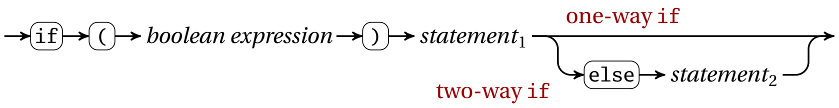


Method invocations



`if` Statement

- A statement can be conditionally executed by using `if` statement, it has the form shown in the syntax diagram below (one-way `if`).



- If the boolean expression evaluates to `true`, *statement*$_1$ is executed, otherwise skipped.

```
if ( age >= 18 ) System.out.println("Adult.");
```

- Two statements can be selectively executed by using the alternative form of `if` statement, shown in the syntax diagram above (two-way `if`).
- If the boolean expression evaluates to `true`, *statement*$_1$ is executed, otherwise *statement*$_2$ is executed. One and only one of the statements is executed.

```
if ( mark >= 50 ) message = "Pass.";
else message = "Fail.";
```


`switch` Statement

- The `switch` statement transfers control to a *case-label* within its body. It has the following form:

```
switch ( expression ) {
    case constant1:   statement-list1
    case constant2:   statement-list2
    ...
    case constantn:   statement-listn
    default:          statement-listd
}
```

- If the *expression* evaluates to *constant*$_i$, then the control is transferred to the "`case` *constant*$_i$" case-label.
- If no constant is matched, then the control is transferred to the `default` case-label.

break Statements in a switch Body

- A break statement in the switch body exits the body.
- Case-labels (including default) are just labels. They do not affect the execution sequence, specifically, they do not stop the previous statements to exit the switch body.
- You must use break to exit, otherwise, the execution flow continues.
- Multiple case-labels can appear in front of a statement that multiple cases can have the same processing.
- The default case-label can be omitted. If present, at most once. If default is omitted and there is no case matched, the entire switch body is skipped.

while Statement

- A loop is a block of statements which is written once but may be repeated several times in succession.
- A while loop consists of two parts: a boolean expression as the *loop condition*, and a block of statement as the *loop body*.
- The loop condition is evaluated first, if true, the loop body is executed, then the control goes back to the loop condition; otherwise the loop body is skip entirely.

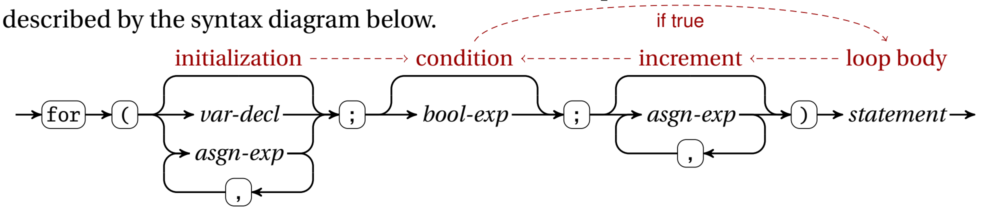
→ while → (→ *boolean expression* →) → *statement* →

- Here is an example of using a while-loop to compute the quotient.

```
q = 0;
while ( n >= 13 ) {
    n -= 13;
    ++q;
}
```


for Statement

- The for statement provides a more convenient and clearer way to combine the initialization, the condition test and the increment step into one structure, whose form is described by the syntax diagram below.



- The initialization is performed first and only once; next, the condition is evaluated and checked; if true, the loop body is executed and the increment is performed after that. The execution comes back to the evaluation and checking of the condition.
- Any of the three parts can be omitted, if the condition is omitted, it is assumed true.
- The variables declared in the initialization part are not available outside the loop.

Defining a Method

- Every method belongs to a class. We must define a method within a class, say *MyClass*.
- Every method has a signature, which mentions the method name, the types of parameters and return value.

```
int multiply(int x, int y)
```

- A statement block following the method header defines the method body.

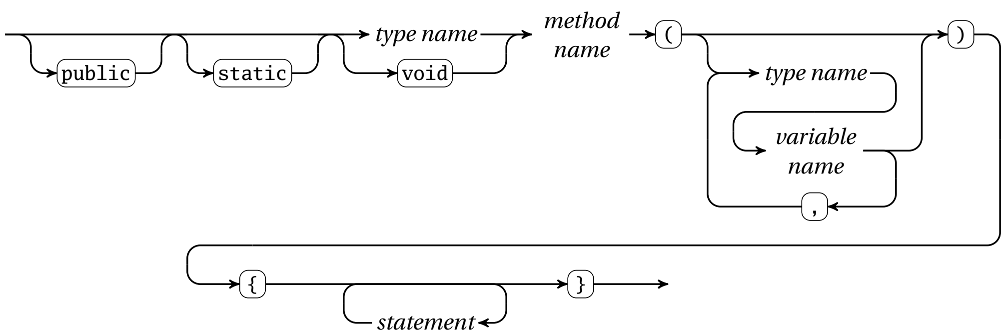
```
{ return this.r * x * y; }
```

- The task that the method performs is specified in the block.
- A method returns when the `return` statement is executed, or when the execution reaches the end of the block if the method has no return value, that is, the return type is `void`.

```
static void doubleStarBars() {
    for ( int i = 0; i < 80; ++i )
        System.out.print((i+1)%40 == 0 ? "*\n" : "*");
}
```


Syntax Diagram of Method Definitions

method definition



Calling a Method

- We call our own methods just like calling methods provided by the system.
- We write an object, say *myObj*, of the class defining the method, followed by the method name and arguments.

```
MyClass myObj = ...
int a = 100 + myObj.multiply(100, 200); // multiplies myObj.r with 100 and 200
```

- The arguments (100 and 200) are assigned to the parameters (*x* and *y*) declared in the signature, the object *myObj* used to call the method is assigned to a special parameter "`this`", they are used as variables in the method.
- A method call is an expression, if it returns a value. The value returned is the value of this expression. You can put it anywhere that an expression fits.
- A method with the `void` return type must be called as a statement. We call a static method by the class name.

```
MyClass.doubleStarBars();
```

Declaring and Initializing Array Variables

- A type name T followed by a pair of brackets `[]` results the type name T`[]` for the arrays of T, such as `int[]` for the arrays of `int` and *String*`[]` for the arrays of *String*.
- We declare an array variable just like declaring other variables, except that we use an array type name.

```
int[] a, b; char[] c; double[] d, e; // five array variables
```

- An array variable can be initialized by 1) a new array, 2) another array vairable, or 3) an *array initializer*.
- An array initializer is a comma-separated list of expressions, enclosed by braces { and }.
- A new array is created by the `new` operator, followed by the array type name with either the length specified in the brackets, or a further array initializer.

```
int[] a = new int[100], b = a;
double[] d = new double[] {1.0, 2.0, 3.0}, e = {1.0, 4.0, 9.0};
```


Setting Array Elements in a Loop

- Often, the value of an element is a function of its index.
- An array of 100 odd numbers:

$$1, 3, 5, \ldots, 199$$

can be created by a loop:

```
int[] a = new int[100];
for ( int i = 0; i < a.length; ++i ) a[i] = 2*i+1;
```

- Sometimes, array elements can be read from the input device repeatedly:

```
import java.util.Scanner;
...
int[] a = new int[50];
try ( Scanner scanner = new Scanner(System.in) ) {
    for ( int i = 0; i < a.length; ++i )
        a[i] = scanner.nextInt();
}
```


Passing Arrays

- Array variables store references, so copying an array variable copies only the reference but not the array.

```
int[] a = {1,2,3}, b;
b = a; // b and a point to the same array.
b[1] = 100; // changing the array pointed to by b also changes the array pointed to by a.
System.out.println(a[1]); // prints 100.
```

- When we pass an array to a method, we transfers only the reference to the array.
- The method below returns the sum of the elements of an array in a range.

```
public static int sumIntArray(int[] a, int startIndex, int stopIndex) {
    int s = 0;
    for ( int i = startIndex; i < stopIndex; ++i )
        s += a[i];
    return s;
}
```

For-each Loop

- Java supports a convenient `for` loop, known as a *for-each* loop, which enables you to traverse an array sequentially without using an index variable.

```
int[] a = new int[] {1,3,5,7,9,11,13,15,17,19};
for ( int u : a )
    System.out.println(u);
```

- You can read the code as “for each element *u* in *a*, do the following.” The array is viewed as a collection of elements.
- Below is the syntax diagram for the for-each statement.

→ `for` → `(` → *type name* → *variable name* → `:` → *array* → `)` → *statement* →

- You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

Classes and Objects

A class template:

Class Name: *Rectangle*
Data Fields:
width is ________
height is ________
Methods:
getArea

Three objects of the *Rectangle* class:

Rectangle Object 1	*Rectangle* Object 2	*Rectangle* Object 3
Data Fields:	Data Fields:	Data Fields:
width is <u>10</u>	*width* is <u>16</u>	*width* is <u>40</u>
height is <u>5.5</u>	*height* is <u>10</u>	*height* is <u>30</u>

- *Classes* are constructs that define objects of the same type, including the layout of the data fields and the definition of the methods.
- Objects of the same class each have their own *instances* of data fields, but share the same definition of the methods.
- Additionally, a class provides a special type of methods, known as *constructors*, which are invoked to construct objects from the class.

An Example of Classes

```
class Rectangle {
    // data fields
    double width, height;

    // constructors
    Rectangle() { width = 1.0; height = 1.0; }

    Rectangle(double width, double height) {
        this.width = width;   // Local variables hide the fields with the same names.
        this.height = height;
    }

    // method
    double getArea() { return width * height; }
}
```

Constructors

- Constructors are a special kind of methods that are invoked to initialize objects.
- A constructor with no parameters is referred to as a *no-arg constructor*.

```
Rectangle() { width = 1.0; height = 1.0; }
```

- Constructors must have the same name as the class itself.
- Multiple constructors can be defined as long as they take different types of parameters.

```
Rectangle(double width, double height) {
    this.width = width;   // Local variables hide the fields with the same names.
    this.height = height;
}
```

- Constructors do not have a return type — not even `void`.
- Constructors are invoked using the `new` operator when an object is created.

```
Rectangle a = new Rectangle();              // a 1.0 × 1.0 rectangle
Rectangle b = new Rectangle(10.0, 5.5);     // a 10.0 × 5.5 rectangle
```


Variables of Reference Types

- Objects must be accessed via references.

```
Rectangle a = new Rectangle();
a.width = 10; a.height = 20;
```

- Variables of reference types store references (pointers) to objects.
- *String* is a system defined class, so variables of *String* are references.
- Assignments to value type variables copy the values.
- Assignments to reference type variables copy the references, but not the objects.
- Two reference variables are equal only if they point to the same object.

```
String a = new String("ABC"), b = a, c = new String("ABC");
```

  We have `a == b` but `a != c`. However, `a.equals(c)` returns `true`.
- References returned by the `new` operator are different from all existing references.

Graphics

- The upper-left corner of the canvas is the origin $(0,0)$
- The X-coordinate increases from left to right, and the Y-coordinate increases from top to bottom.
- A line is determined by its two end points: (x_1, y_1) and (x_2, y_2).
- A cubic curve is determined by its two end points and two control points: (x_1, y_1), $(ctrlx_1, ctrly_1)$, $(ctrlx_2, ctrly_2)$ and (x_2, y_2).
- Shapes can be transformed by the affine transformation, including

  translation, rotation, reflection, scaling and *shearing*.