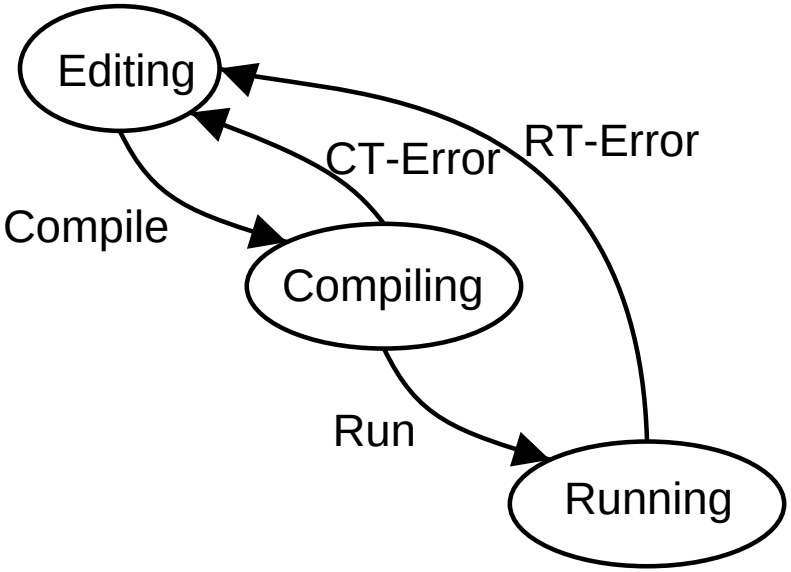
Errors

- Errors:
  - Compile-time errors
  - Run-time errors
    - Exceptions
    - Logical

Errors



Basic Java Syntax

- A Java program is made up of one or more *class definitions*
- A class definition is made up of zero or more *method definitions*
- A method definition is made up of zero or more *statements* and *variable declarations*
- Roles:
  - Classes: Modules and Types of objects
  - Methods: procedures, functions, algorithms
  - Statements: instructions

Basic Java Syntax

```
public class ClassName
{
    // Body of ClassName
    // ...
    // List of method definitions
}
```

Basic Java Syntax

```
public class HelloWorld
{
    // Body of ClassName
    // ...
    // List of method definitions
}
```

Basic Java Syntax

```
public class Classname
{
    // method header
    {
        // method body: list of statements
    }
}
```

Basic Java Syntax

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
        System.out.println("Good bye");
    }
}
```

Bad Java Syntax

```
public class HelloWorld
{
    System.out.println(“Hello”);
    System.out.println(“Good bye”);
}
```

Bad Java Syntax

```
public static void main(String[] args)
{
    System.out.println(“Hello”);
    System.out.println(“Good bye”);
}
```

Bad Java Syntax

```
public static void main(String[] args)
{
    public class HelloWorld
    {
        System.out.println(“Hello”);
        System.out.println(“Good bye”);
    }
}
```

Indentation

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println(“Hello”);
        System.out.println(“Good bye”);
    }
}
```

Indentation

```
public class HelloWorld
{
public static void main(String[] args)
{
System.out.println(“Hello”);
System.out.println(“Good bye”);
}
}
```

Indentation

```
                public class HelloWorld
{
 public static void main(String[] args)
                                    {
        System.out.println(“Hello”);
System.out.println(“Good bye”);
    }
                   }
```

Indentation

```
public class HelloWorld{public static void main(St
ring[] args){System.out.println(“Hello”);System.ou
t.println(“Good bye”);}}
```

User Interface

- The user interface of a program is the way it interacts with the user: keyboard/mouse/windows/text
- Graphical User Interface:
  - Windows: buttons, text boxes, slidebars, graphics, etc.
  - Input with mouse and keyboard.
- Textual User Interface:
  - Console window: plain text
  - Input: keyboard only
  - Output:

```
System.out.println("text");
```

Introduction to statements

- The print statement

```
System.out.println(string_literal);
System.out.print(string_literal);
```

- String literals:

```
"(almost)any characters"
"This is a string literal"
"String literals can contain almost any character,
"a"
""
"24"
```

Introduction to statements

- String concatenation:

  string_literal + *string_literal*
  string_literal + *number_literal*

  “This is a ”+“message”
  “This is a message”
  “There are ”+70+“ students in this class”

- String literals with numbers are not numbers: “17” is not the same as 17

  “17” + “29”

is

  “1729”

while

  17 + 29

is

  46

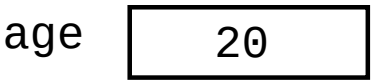
Simple programs

```
// File: PrintingStuff.java
public class PrintingStuff
{
    public static void main(String[] args)
    {
        System.out.println(“This trivial program j
        System.out.println(“prints this text to a
        System.out.println(“Window.”);
    }
}
```

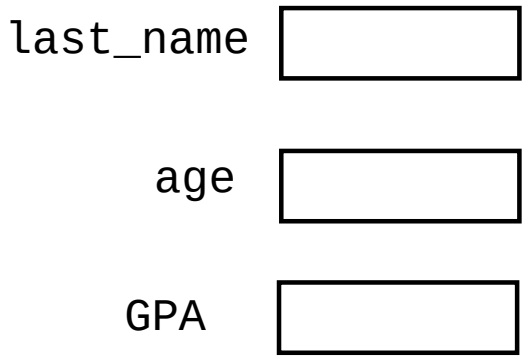
Variables

- A variable is a memory location
- A variable can contain information
- A variable has a symbolic name

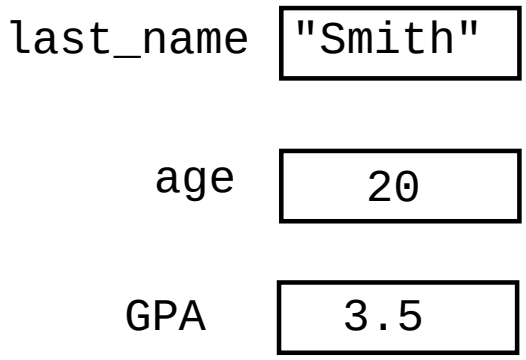
Variables



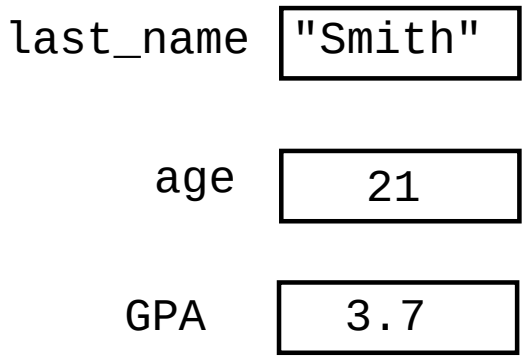
Variables



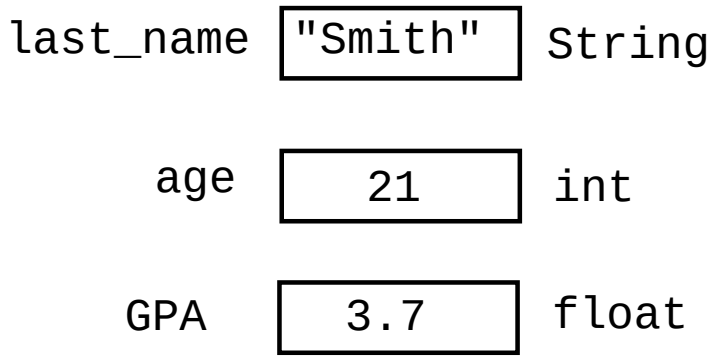
Variables



Variables



Variables



Variable declaration

- A *variable declaration* is a statement that declares tha a variable is going to be used.
- A variable declaration goes inside some method
- A variable declaration has the form:

  type identifier;

- Examples:

```
String last_name;
int age;
float GPA;
```

Assignment

- An *assignment* is a statement that gives a value to a variable
- An assignment goes inside some method
- An assignment has the form:

  variable = *value*;

- Its meaning it to put the value into the memory location of the variable
- Examples:

```
last_name = "Smith";
age = 20;
```

- Note that the following are *incorrect*:

```
20 = age;
"Smith" = last_name;
```

Assignment

- The variable must be declared before being assigned a value

```
String last_name;
last_name = "Smith";
```

- But the following is wrong:

```
age = 20;
int age;
```

- The type of the value must be the same as the type of the variable

```
last_name = 20; // Incorrect
age = "Smith"; // Incorrect
```

Variables and String expressions

- Variables can be used with concatenation in String expressions

  `“your age is ”+age`

- is equivalent to

  `“your age is 19”`

- if the variable age contains the value 19

A simple program

```
public class PrintData
{
  public static void main(String[] args)
  {
    String last_name;
    int age;
    last_name = “Smith”;
    age = 20;
    System.out.println(“Your last name is ” + last_name);
    System.out.println(“You are ” + age + “ years old”);
  }
}
```

Basic java programs

```
public class ClassName
{
    public static void main(String[] args)
    {
        // Statements
    }
}
```

Statements

- Print statement

```
System.out.println(string_expression);
```

- Variable declaration

```
type identifier;
```

- Assignment

```
variable = value;
```

- Statements in a method are executed in *sequential order* from top to bottom

Assignment

- In an assignment

```
variable = value;
```

- the variable must have been declared before,

```
x = 7; // incorrect
int x;
```

- the type of the variable must match the type of the value

```
int x;
x = "7"; // incorrect
```

Sequential execution

```
public class OrderTest
{
    public static void main(String[] args)
    {
        int a;
        int b;
        a = 2;
        b = 3;
        b = 5;
        a = 8;
        System.out.println(a);
        System.out.println(b);
    }
}
```

Sequential execution

```
public class OrderTest
{
    public static void main(String[] args)
    {
        int a;
        int b;
        b = 5;
        a = 8;
        a = 2;
        b = 3;
        System.out.println(a);
        System.out.println(b);
    }
}
```

Some syntactic shortcuts

- Several variables of the same type can be declared in the same variable declaration:

  type var1, var2, ..., varn ;

- Examples:

```
int a;
int b;
```

is equivalent to

```
int a, b;
```

Some syntactic shortcuts

- A variable can be initialized when declared

```
int a;
a = 2;
```

is equivalent to

```
int a = 2;
```

- But a variable cannot be redeclared, so

```
int b = 3;
int b = 2;
```

is incorrect, while the following is correct

```
int b = 3;
b = 2;
```

User Interface

- Interaction between the user and some program
- Textual UI
  - Output:

    ```
    System.out.println(string_expression);
    ```

  - Input:

    ```
    scanner.nextInt();
    scanner.nextLine();
    ```

- Examples:

```
Scanner myScanner = new Scanner(System.in);
int n;
n = myScanner.nextInt();
```

User Interface

```
import java.util.Scanner;
public class UserInputTest {
    public static void main(String[] args)
    {
        Scanner myScanner = new Scanner(System.in);
        String name;
        int age;
        System.out.print(“Enter your name: ”);
        name = myScanner.nextLine();
        System.out.print(“Enter your age: ”);
        age = myScanner.nextInt();
        System.out.println(“Your name is ” + name);
        System.out.println(“You are ” + age + “ years old”);
    }
```

```
}
```

Data types

- Each variable has a *data type*

```
String major;
int age;
```

- A data type is a set of possible values
  - `int` is the set of integers
  - `String` is the set of strings
  - `float` is the set of rational numbers written as a decimal expansion
  - `double` is the set of rational numbers as a decimal expansion, with double precision
  - `char` is the set of characters
  - `boolean` is the set {`true`, `false`}
  - `byte` is the set of bytes, writen in decimal

Data types

Data type	Possible values	Examples
int	all integers between -2^{31} and $2^{31}-1$	0, 1, 2, -3, -1729
String	all character strings enclosed in " "	"hello bye", "", "a"
float	rationals between -3.4×10^{38} and 3.4×10^{38}	0.0f, -2.3f, 111.001f
double	rationals between -1.7×10^{308} and 1.7×10^{308}	0.0, -2.3, 111.001
char	all individual characters enclosed in ' '	'a', 'b','z','7','+','A'
boolean	only true and false	true, false
byte	all integers between -128 and 127	-128, 0, 8
long	integers between -2^{63} and $2^{63}-1$	0l, 65536l, -3l
short	integers between -2^{15} and $2^{15}-1$	-3, -2, 0, 1, 4

Data types

Data type	Size	Range
boolean	8 bits (7 unused)	0 - 1
byte	8 bits	-2^7 to $2^7 - 1$
char	8 bits (ASCII), 16 bits (Unicode)	0 to 2^8 (ASCII) 0 to 2^{16} (Unicode)
short	16 bits	-2^{15} to $2^{15} - 1$
int	32 bits	-2^{31} to $2^{31} - 1$
long	64 bits	-2^{63} to $2^{63} - 1$

Real numbers

$\sqrt{2}$

$\sqrt{3}$

π

$e = 2.718...$

$\varphi = \frac{1 \pm \sqrt{5}}{2} = \begin{cases} 1.618... \\ 0.618... \end{cases}$

Assignment and data types

```
int x = 3.141592;          // Line 1
float pi = 3.141592f;      // Line 2
double e = 2.718;          // Line 3
float phi = 1.618;         // Line 4
int n = 32768;             // Line 5
int m = 32767;             // Line 6
long o = 32768l;           // Line 7
String letter = ‘A’;       // Line 8
char letter2 = ‘A’;        // Line 9
char letter3 = “B”;        // Line 10
```

Arithmetic expressions

- If a variable is of a numeric type (int, float, long, etc.) then an assignment can take the form

  variable = *arithmetic_expression*;

- where arithmetic_expression is an expression involving:
  - numbers (of the appropriate type)
  - operators (+, -, *, /, %)
  - variables (of numeric type)
  - parenthesis

- Example:

```
double grade, assignments, midterm, final;
assignments = 97.5;
midterm = 75.5;
final = 80.0;
grade = assignments * 0.25
      + midterm * 0.20
      + final * 0.55;
```

Arithmetic expressions

- Parenthesis are used to group operations:

```
double grade, assignments, midterm, final;
double a1 = 20, a2 = 19, a3 = 9, a4 = 14, a5 = 18;
assignments = a1 + a2 + a3 + a4 + a5;
midterm = 75.5;
final = 80.0;
grade = assignments * 0.25
      + midterm * 0.20
      + final * 0.55;
```

is equivalent to

```
double grade, midterm, final;
double a1 = 20, a2 = 19, a3 = 9, a4 = 14, a5 = 18;
midterm = 75.5;
final = 80.0;
grade = (a1 + a2 + a3 + a4 + a5) * 0.25
      + midterm * 0.20
      + final * 0.55;
```

Operator precedence

- Operators are evaluated depending on their precedence:

```
result = 6 + 5 * 3;
```

- If the operators did not have precedence, the expression would have as value 33
- But it's real meaning is:

```
result = 6 + (5 * 3);
```

- Which evaluates to 6 + 15 which is 21.
- Operators have "associativity":

```
result = 6 + 5 + 3 + 9;
```

- is evaluated as

```
result = ((6 + 5) + 3) + 9;
```

Operator precedence

Precedence level	Operator	Operation	Associativity
1	+	unary plus	right to left
	-	unary minus	
2	*	multiplication	left to right
	/	division	
	%	remainder (modulo)	
3	+	addition	left to right
	-	substraction	
	+	string concatenation	

Operator precedence

```
r = 8 / 2 / 2;
```

is evaluated as

```
r = ((8 / 2) / 2);
```

and

```
s = 12 * 2 - - 3;
```

is evaluated as

```
s = (12 * 2) - (-3);
```

and

```
t = -2 * 4 + - (a - 1);
```

is evaluated as

```
t = ((-2) * 4) + (- (a - 1));
```

Precedence

a	+	b	+	c	+	d	+	e
	1		2		3		4	

a	+	b	*	c	-	d	/	e
	3		1		4		2	

is the same as (a+(b*c))-(d/e)

(a	+	b)	*	(c	-	d)	/	e
	1		3		2		4	

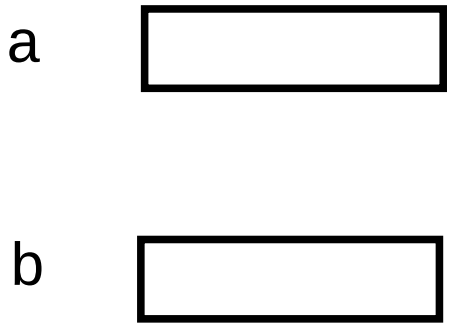
Sequential execution

```
double a, b;
a = 2.0;
b = a;
a = 3.0;
System.out.println(a);
System.out.println(b);
```

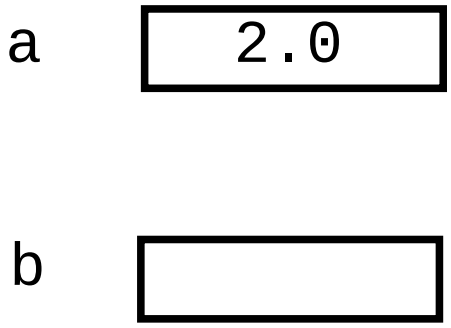
Sequential execution

```
double a, b;
a = 2.0;
b = a;
a = 3.0;
System.out.println(a);
System.out.println(b);
// Prints
// 3.0
// 2.0
```

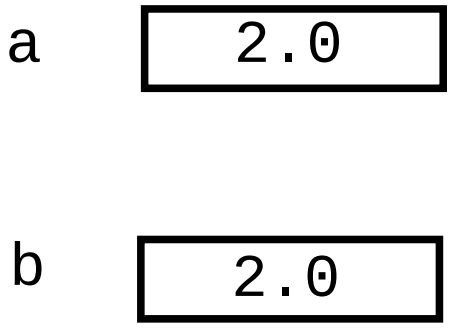
Sequential execution



Sequential execution



Sequential execution



Sequential execution

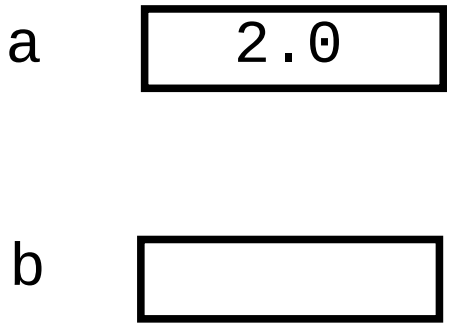
Sequential execution

```
double a, b;
a = 2.0;
a = 3.0;
b = a;
System.out.println(a);
System.out.println(b);
```

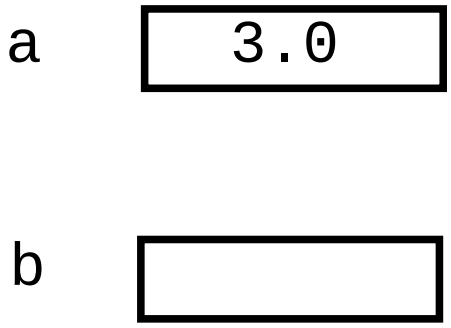
Sequential execution

```
double a, b;
a = 2.0;
a = 3.0;
b = a;
System.out.println(a);
System.out.println(b);
// Prints
// 3.0
// 3.0
//
// a and b have the same contents (the same value)
// but they are different variables
```

Sequential execution



Sequential execution



Sequential execution

Sequential execution

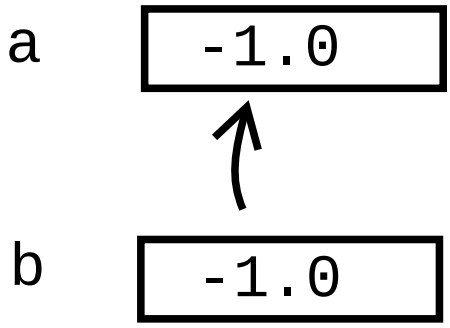
```
double a, b;
a = 2.0;
b = -1.0;
a = b;
b = a;
System.out.println(a);
System.out.println(b);
```

Sequential execution

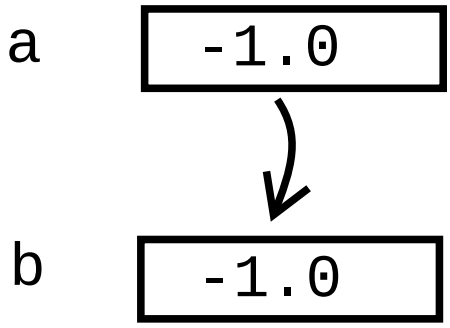
```
double a, b;
a = 2.0;
b = -1.0;
a = b;
b = a;
System.out.println(a);
System.out.println(b);
// Prints
// -1.0
// -1.0
```

Sequential execution

Sequential execution



Sequential execution



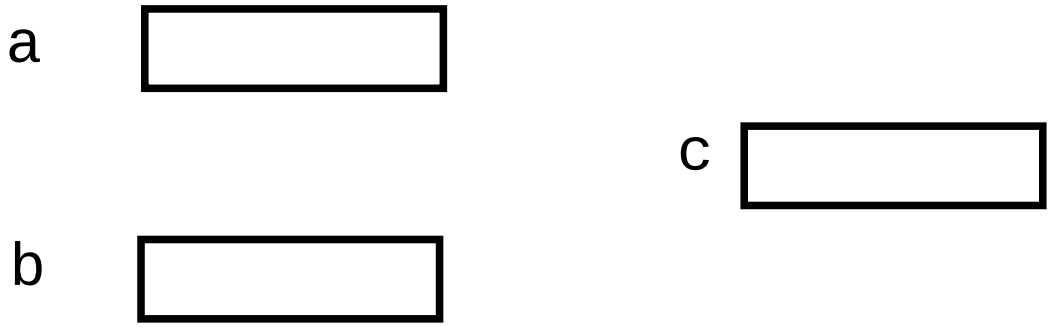
Sequential execution

```
double a, b, c;
a = 2.0;
b = -1.0;
c = a;
a = b;
b = c;
System.out.println(a);
System.out.println(b);
```

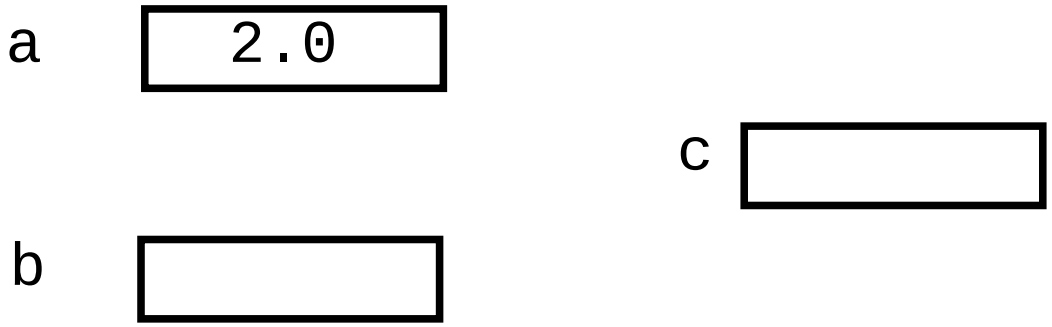
Sequential execution

```
double a, b, c;
a = 2.0;
b = -1.0;
c = a;
a = b;
b = c;
System.out.println(a);
System.out.println(b);
// Prints
// -1.0
// 2.0
// This implements a 'swap' between variables
```

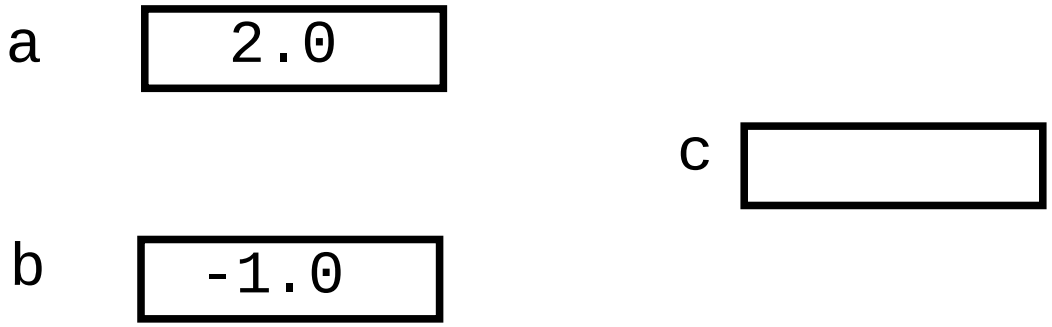
Sequential execution



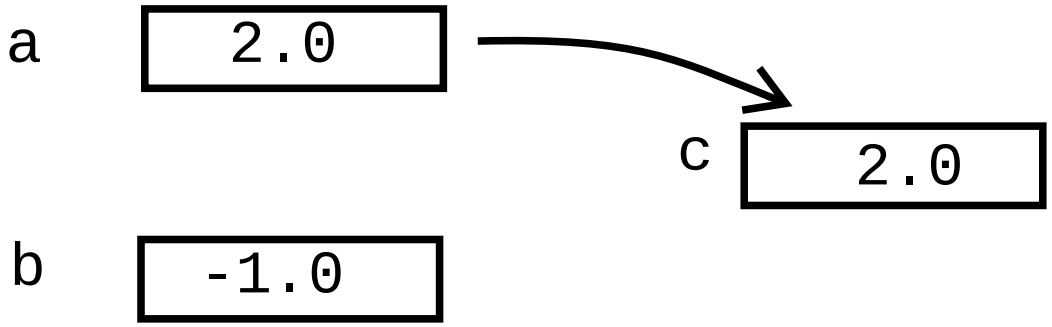
Sequential execution



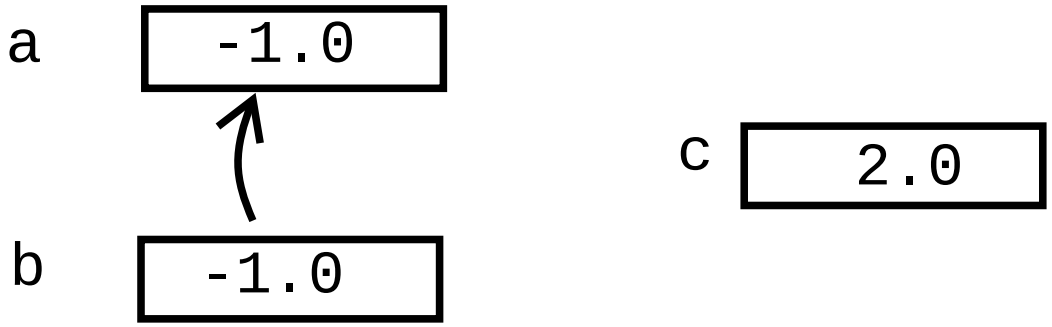
Sequential execution



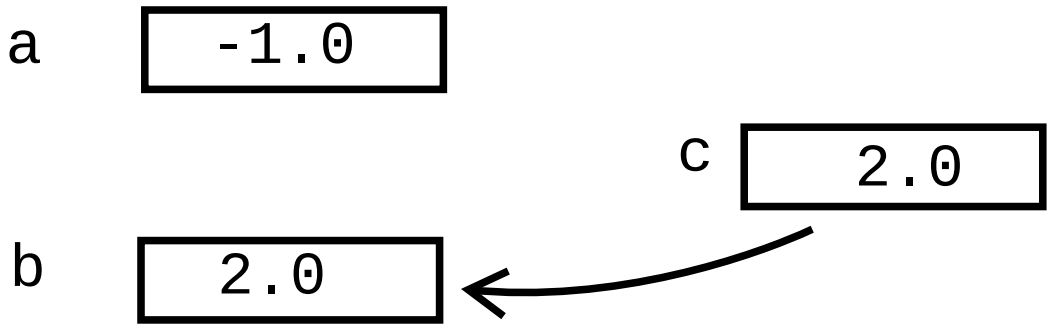
Sequential execution



Sequential execution



Sequential execution



Sequential execution

ORDER MATTERS

More on arithmetic operations

- The remainder operator % computes the remainder of an integer division (not percentages!)
  - 8 % 2 evaluates to 0
  - 7 % 2 evaluates to 1
  - 8 % 3 evaluates to 2
  - 3 % 5 evaluates to 5
- In general, for every integers a and b, $0 \leqslant a\%b < b$
- Division has a different meaning for integers and for floats and doubles
- The division between two integers is an integer
- The division between a float or double and an integer is a float or double
  - 8 / 3 evaluates to 2
  - 8.0 / 3 evaluates to 2.666666...

Statements

- Variable declaration

```
type identifier;
```

- Assignment

```
variable = expression;
```

- User Interface: output

```
System.out.println(string_expression);
```

- User Interface: input

```
variable = scanner.nextType();
```

Primitive Data Types

General category	Type	Description	Examples
Numeric	int	Integers	0,1,-3
	long	Long integers	65537l
	short	Short integers	2,-6
	byte	Bytes	255
	float	Rationals	1.33f
	double	Rationals	1.618
Text	char	Single characters	'x', ' '
	String	Sequences of characters	"abc"
Logic	boolean	Truth values	true, false

Syntax of string expressions

- A *string expression* e is either
  - a string literal (characters enclosed in " ")
  - or a variable of String type
  - or an expression of the form

    $$e_1 + e_2$$

    where e_1 is a *string expression*, + is the concatenation operator, and e_2 is either a *string expression* or an *arithmetic expression* or a *character expression*

- Examples:

```
"this is a long       literal"
"my name is " + name
"the average is " + (a + b)/2
"I am taking " + n + " courses"
"My initials are " + 'E' + 'P'
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    // List of statements
  }
}
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    int n = 4;
    “I am taking ” + n + “ courses”; // WRONG!
  }
}
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    int n = 4;
    System.out.println(“I am taking ” + n + “ courses”);
  }
}
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    int n = 4;
    String message = “I am taking ” + n + “ courses”;
  }
}
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    float a = 4, b = 2;
    (a + b) / 2           // WRONG!
  }
}
```

Method structure

```
public class SomeProgram {
  public static void main(String[] args)
  {
    float a = 4, b = 2, c;
    c = (a + b) / 2;
  }
}
```

Statements vs expressionfs

- A statement is not an expression
- An expression is not a statement
- An expression is a term (e.g. x*2, "a"+b, etc.) inside a statement, which has a value.
- A statement is an instruction to be executed (e.g. assignment, print, etc.) and it has no value.
- A method's body is a list of statements, not a list of expressions

Strings

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = Smith;
System.out.println(first_name);
System.out.println(last_name);
```

Strings

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = Smith;
System.out.println(first_name);
System.out.println(last_name);
```

Strings

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = "Smith";
System.out.println(first_name);
System.out.println(last_name);
```

Sequential execution

```
String first_name, last_name;
first_name = “Adam”;
last_name = “Smith”;
last_name = first_name;
first_name = last_name;
System.out.println(first_name);
System.out.println(last_name);
```

Sequential execution

```
Adam
Adam
```

Sequential execution

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = "Smith";
temp = last_name;
last_name = first_name;
first_name = temp;
System.out.println(first_name);
System.out.println(last_name);
```

Sequential execution

```
Smith
Adam
```

Sequential execution

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = "Smith";
temp = last_name;
last_name = first_name;
first_name = temp;
System.out.println(first_name);
System.out.println(last_name);
```

Sequential execution

```
String first_name, last_name, temp;
first_name = "Adam";
last_name = "Smith";
temp = last_name;
first_name = temp;
last_name = first_name;
System.out.println(first_name);
System.out.println(last_name);
```

Sequential execution

```
Smith
Smith
```

Assignment

- Assignment **is not** equality
- The right-hand side of an assignment can contain the same variable as the left hand-side:

```
int count = 0;
// Here the value of count is 0
count = count + 1;
// Here the value of count is 1

String name;
name = "Bond";
name = "James " + name;
```

- String concatenation is not commutative (a+b is not b+a)

Operators and types

- The meaning of an operator depends on its context, and in particular on the types of its arguments

```
int a = 8, b = 3, c;
c = a / b; // Integer division

double d = 8.0, e = 3.0, f;
f = d / e; // Floating point division

int g;
g = a + b; // Addition

String h = “one”, i = “two”, j;
j = h + i; // String concatenation
```

Assignment

- If *variable* is of numeric type (int, float, etc.)

  ```
  variable = arithmetic_expression;
  ```

- If *variable* is of String type

  ```
  variable = string_expression;
  ```

- If *variable* is of boolean type

  ```
  variable = boolean_expression;
  ```

Boolean expressions

```
true

false

true && true

false || true

!false

!true && false || !false

!(sunny && false)

5 < 7

6 >= 8

2 + x == 9 && b < 8 || c == true
```

Syntax of boolean expressions

- A *boolean expression* e is either
  - the constant `true`
  - or the constant `false`
  - or a boolean variable
  - or an expression of the form

    $$e_1 \textit{ boolop } e_2$$

    where e_1 and e_2 are boolean expressions and *boolop* is one of the binary boolean operators: && (and) or || (or)
  - or an expression of the form

    $$!e'$$

    where e' is an boolean expression and ! is the unary boolean operator for negation.
  - or an expression of the form

    $$(e')$$

    where e' is an boolean expression
  - or an expression of the form

    $$e_1 \textit{ relop } e_2$$

    where e_1 and e_2 are arithmetic expressions and *relop* is one of the binary *relational* operators: <, <=, ==, >=, >, !=

Boolean expressions

```
boolean a, b;
a = true;
b = !a;

int x, y, d;
boolean c;
c = x - y >= 0 && x - y < d;
c = ((x - y) >= 0) && ((x - y) < d);

float temp = -25.2f, windchill = -35.2f;
boolean sunny = true, rain, windy, cold, ski;
rain = !sunny;
windy = windchill - temp > -10.0f;
cold = temp < -20.0f;
ski = sunny && !windy || !cold;
```

Boolean expressions

```
float temp = -25.2f, windchill = -35.2f;
boolean sunny = true, rain, windy, cold, ski;
rain = !sunny;
windy = windchill - temp > -10.0f;
cold = temp < -20.0f;
ski = sunny && !windy || !cold;
rain = true;

boolean b = true;
b = false;
b = !b;
b = true && false;
```

Precedence

Precedence	Operator	Operation	Associativity
1	+	Unary plus	right to left
	-	Unary minus	
	!	Logical negation (NOT)	
2	*	Multiplication	left to right
	/	Division	
	%	Remainder (modulo)	
3	+	Addition	left to right
	-	Substraction	
	+	String concatenation	
4	<	Less than	Left to right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
5	==	Equals to	Left to right
	!=	Different to	
6	&&	Logical conjunction (AND)	Left to right
7	||	Logical disjunction (OR)	Left to right

Precedence

4	+	x	==	9	&&	b	<	8	||	!	c
	2		4		5		3		6	1	

is the same as (((4+x)==9)&&(b<8))||(!c)

4	+	x	==	9	&&	b	<	8	||	!	(	1	<	x	)
	3		5		6		4		7	2			1		

is the same as (((4+x)==9)&&(b<8))||(!(1<x))

Semantics of expressions

- The meaning of an expression is the value of the expression
  - An arithmetic expression is evaluated to a number
  - A string expression is evaluated to a string
  - A boolean expression is evaluated to a truth-value (true or false)

Semantics of boolean expressions

- The value of `true` is true
- The value of `false` is false
- The value of a boolean variable is whatever value it contains
- The value of $e_1 \&\& e_2$ is true if the values of e_1 and e_2 are both true, and false otherwise
- The value of $e_1 || e_2$ is true if the value of e_1 or the value of e_2 true, and false if the values of both are false
- The value of $!e$ is true if the value of e is false, and false if the value of e is true

Semantics of boolean expressions

- The value of $a_1 < a_2$ is true if the value of a_1 is strictly less than the value of a_2
- The value of $a_1 <= a_2$ is true if the value of a_1 is less or equal to the value of a_2
- The value of $a_1 > a_2$ is true if the value of a_1 is strictly greater than the value of a_2
- The value of $a_1 >= a_2$ is true if the value of a_1 is greater or equal to the value of a_2
- The value of $a_1 == a_2$ is true if the value of a_1 is equal to the value of a_2
- The value of $a_1! = a_2$ is true if the value of a_1 is different to the value of a_2

Semantics of boolean expressions

- Truth tables
- Assume that a and b are boolean expressions

a	!a
`true`	false
false	`true`

a	b	a && b
`true`	`true`	`true`
`true`	false	false
false	`true`	false
false	false	false

a	b	a || b
`true`	`true`	`true`
`true`	false	`true`
false	`true`	`true`
false	false	false

Semantics of boolean expressions

- The value of

```
true && false || !false
```

is the same as the value of

```
(true && false) || (!false)
```

which is

```
(true && false) || true
```

which is

```
false || true
```

which is

```
true
```

Semantics of boolean expressions

- Exclusive or: either a is true or b is true but not both

  a && !b || !a && b

a	b	!b	a&&!b	!a	!a&&b	a&&!b || !a&&b
true	true	false	false	false	false	false
true	false	true	true	false	false	true
false	true	false	false	true	true	true
false	false	true	false	true	false	false

Semantics of boolean expressions

- What is the value of 4+x==9 && b < 8 || !c ?

Semantics of boolean expressions

- What is the value of 4+x==9 && b < 8 || !c ?
- It depends on the values of the relational expressions (4+x==9), (b<8) and the boolean expression c.
- These expressions depend on the values of x, b and c, which we do not know
- ...but we can consider the all the possible truth values for each subexpression:

Semantics of boolean expressions

4+x==9	b<8	c	!c	4+x==9 && b<8	(((4+x)==9)&&(b<8))||(!c)
true	true	true	false	true	true
true	true	false	true	true	true
true	false	true	false	false	false
true	false	false	true	false	true
false	true	true	false	false	false
false	true	false	true	false	true
false	false	true	false	false	false
false	false	false	true	false	true

Semantics of boolean expressions

temp > -20.0 || !windy && sunny

temp>-20.0	windy	sunny	!windy	!windy && sunny	(temp > -20.0) || (!windy && sunny)
true	true	true	false	false	true
true	true	false	false	false	true
true	false	true	true	true	true
true	false	false	true	false	true
false	true	true	false	false	false
false	true	false	false	false	false
false	false	true	true	true	true
false	false	false	true	false	false

Note about variables

- The name of a variable is just a symbolic name to make the program more readable
- The name of the variable does not give the variable any special meaning

```
double temp = -27.0;
boolean cold;
cold = temp <= -20.0;
```

- Does not mean that it is actually cold!
- It only means

```
double x = -27.0;
boolean y;
y = x <= -20.0;
```

- But it is useful for the programmer to give variables meaningful names

Data conversion

- Sometimes it is useful to look at data as if they were from a different type
- For example:
  - Adding an integer and a double
  - Obtaining the ASCII code of a character
- Forms of data conversion:
  - Implicit:
    * Assignment conversion
    * Promotion
  - Explicit: Casting

Data conversion

- Assignment conversion: A value of one type is assigned to a variable of a different type, as long as the types are compatible

```
int n = 7;
double d = n;
long k = n;
int m = d; // Wrong: compile-time error
```

- Promotion: an expression "promotes" the types of its operands to its "largest" type

```
int m = 8;
float x = 3.0f, y;
y = x + m;
```

Data conversion

- Casting expressions (not a statement)

  (*type*)*expression*

- Examples:

```
int n = 3;
double p;
p = (double)n + 4.0;

int a = 3, b = 8;
float c, d;
c = b/a;
d = (float)b/a;
System.out.println(c);  // 2.0
System.out.println(d);  // 2.666666...
```

Data conversion

```
double r = 2.41;
int a;
a = r; // Error
```

Data conversion

```
double r = 2.41;
int a;
a = (int)r;    //OK:  Narrowing casting
```

Data conversion

- There are two types of casting:
  - Narrowing conversions: from a type which requires more memory to a type that requires less
  - Widening conversions: from a type which requires less memory to a type which requires more

- If `expression` has type `t`, and `t` requires more memory than type `s`, then `(s)expression` is a narrowing conversion (e.g. `int` to `byte`, `double` to `float`, `float` to `int`, ...)

- If `expression` has type `t`, and `t` requires less memory than type `s`, then `(s)expression` is a widening conversion (e.g. `byte` to `double`, `long` to `int`, ...)

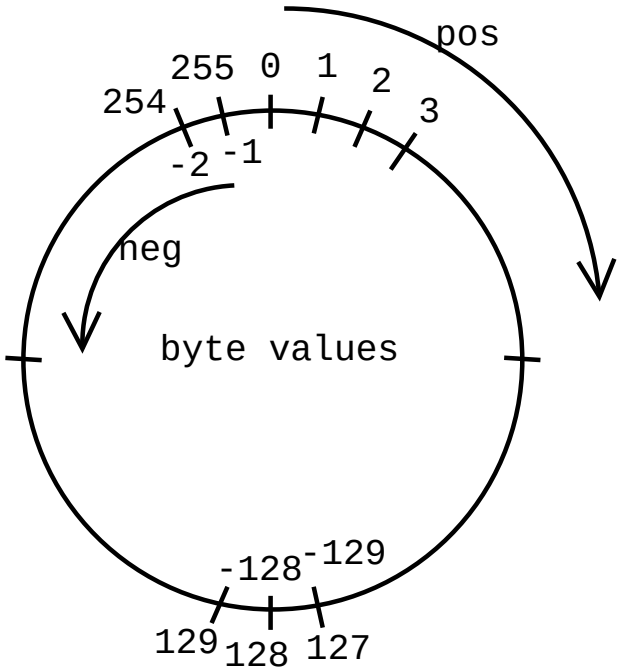
Data conversion

- Widening conversions are safe: no loss of information
- Narrowing conversions are not safe: possible loss of information

```
float x = 2.71f;
int i = (int)x;
// i == 2

int k = 130;
byte b = (byte)k;
// b = -126
```

Data conversion



128 = -128	byte b	b+$k2^8$=b
129 = -127	int i	i+$k2^{32}$=i
256 = 0	k is any integer	
257 = 1		

The end