Announcements

- Still more tutorials on monday and tuesday
- Tentative date for the midterm: March 1st

Statements

- Variable declaration

```
type identifier;
```

- Assignment

```
variable = expression;
```

- User Interface: output

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

- User Interface: input

```
variable = Keyboard.readType();
```

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

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

- Operator precedence: the order in which operators get evaluated
  - Highest precedence: unary +, unary -
  - Mid precedence: *, /, %;
  - Low precedence: +, -

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	

Syntax of arithmetic expressions

- An *arithmetic expression* e is either
  - a numeric value (constant)
  - or a variable of numeric type
  - or an expression of the form

    $$e_1 \textit{ binop } e_2$$

    where e_1 and e_2 are *arithmetic expressions* and *binop* is one of the binary arithmetic operators: +, -, *, /, or %
  - or an expression of the form

    $$\textit{unop } e'$$

    where e' is an *arithmetic expression* and *unop* is one of the unary arithmetic operators: + or -.
  - or an expression of the form

    $$(e')$$

    where e' is an *arithmetic expression*.

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 expressions

- 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 usefule for the programmer to give variables meaningful names

Class name and files

- A class definition like:

```
public class MyProgram
{
  public statc void main(String[] args)
  {
    //...
  }
}
```

Must be in a file named

```
MyProgram.java
```

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

- Casting expressions (not a statement)

  ```
  (type)expression
  ```

- Examples:

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

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 widening conversion

- If `expression` has type `t`, and `t` requires less memory than type `s`, then `(s)expression` is a widening conversion

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
```