Advanced Java Programming

Ohad Barzilay,
Tel-Aviv University
Spring '06

Java Syntax & Structures

Identifiers, Keywords, Literals
Separators and Operators

Java Syntax Basics

- The java syntax is case-sensitive.
- Semi-colon (;) is used as line terminator.
- Curly braces ({,}) are used for block structure .
- There are several reserved keywords.

Language Syntax Symbols Examples

- Literals (numbers, strings etc.):
  - `219 "hello"`
- Operators (arithmetic, boolean, etc.):
  - `+ - * / & |`
- Identifiers (names for variables, methods, classes etc.):
  - `result`
- Keywords & Separators:
  - `for if while { } ( ) ;`

Keywords

- The following keyword are reserved and cannot be used as identifiers:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Literals

- A literal is a source code representation for a value of:
  - a primitive type (Integer, Floating-Point, Boolean, Character)
  - the String type,
- Literals are the representation of simple data values in our language: mainly numbers and text.
- Reserved literal words: `null`, `true`, and `false`

Integer Literals

- An integer literal may be expressed in:
  - decimal (base 10),
  - hexadecimal (base 16), using the prefix 0x
  - octal (base 8), using the prefix 0

Floating-Point Literal Parts

- a whole-number part: -22
- a decimal point (represented by an ASCII period character): -22.
- a fractional part: -22.58
- an exponent (indicated by the letter e or E followed by an optionally signed integer): -22.58e3
- and a type suffix: -22.58e3f

$= -22.58 * 10^3$

Character Literals

- A character literal is expressed as a character or an escape sequence, enclosed in single quotes (').
- The single-quote, or apostrophe, character is \u0027
- A character literal is always of type char:

```
'a' '%'  '\t'   '\\'
  '\u03a9' '\uFFFF'  '\177'
```


String Literals

- A string literal consists of zero or more characters enclosed in double quotes. A string literal is always of type String:

```
""                  // the empty string
"This is a string"  // a string containing 16 characters
"This is a " +      // a string-valued constant expression,
"two-line string"   // formed from two string literals
```


String Conversions

Type	To String	From String
boolean	*String.ValueOf(boolean)*	*new Boolean(String).booleanValue()* **OR** *Boolean.parseBoolean(String)*
int	*String.valueOf(int)*	*Integer.ParseInt(String,int base)* **OR** *new Integer(String).intValue();*
long	*String.ValueOf(long)*	*Long.ParseLong(String,int base)*
float	*String.ValueOf(float)*	*new Float(String).floatValue()*
double	*String.ValueOf(double)*	*new Double(String).doubleValue()*

String Conversions

- Converting primitive to Strings can also be done using the + operator:

```
int  i = 4;
String s = "" + i;
```

- Converting strings to primitives can be done either by using the `static` wrapper methods, or by creating a temp wrapper object and apply the appropriate instance method

Boolean Literals

- The `boolean` type has two values, represented by the literals true and false.
- A Boolean literal is always of type `boolean`.

Separators

- The following nine ASCII characters are the separators (punctuators):
  - `( ) { } [ ] ; , .`
- Separators have special meanings in our language and so cannot be used freely (e.g. as part of an identifier).

Operators

Operators	Associative
++ -- + *unary* - *unary* ~ ! (<data_type>)	R to L
* / %	L to R
+ -	L to R
<< >> >>>	L to R
< > <= >= instanceof	L to R
== !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
<boolean_expr> ? <expr1> : <expr2>	R to L
= *= /= %= += -= <<= >>= >>>= &= ^= |=	R to L

Arithmetic operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
++	Increment
--	Decrement

Boolean Operators

Operator	*Use*	*Returns true if*
&	*exp1 & exp2*	*exp1 and exp2 are both true*
&&	*exp1 && exp2*	*exp1 and exp2 are both true, conditionally evaluates exp2*
|	*exp1 | exp2*	*either exp1 or exp2 is true*
||	*exp1 || exp2*	*either exp1 or exp2 is true, conditionally evaluates exp2*
^	*exp1 ^ exp2*	*if exp1 or exp2 is true but not both*
!	*!exp*	*exp is false*

Bit-wise Operators

- Perform the operation on each bit

Operator	***Use***	***Evaluates***
~	*~exp1*	*Flips all the bits of exp1*
&	*exp1 & exp2*	*Always evaluates exp1 and exp2*
|	*exp1 | exp2*	*Always evaluates exp1 and exp2*

- `~10000101 == 01111010`
- `10000101 | 10110001 == 10110101`
- `10000101 & 10110001 == 10000001`

Strong Typing

- The Java programming language is a strongly typed language.
- Strongly typed - every variable and every expression has a type that is known at compile time (as opposed to runtime).

Typing Features

- Limit the values
  - That a variable can hold or
  - That an expression can produce
- Limit the operations supported on those values
- Determine the meaning of the operations

Strong typing helps detect errors at compile time

Java Types

- The data types of the Java programming language are divided into two categories:
  - primitive types
  - reference types

Reference Type

- Enables a mechanism to create and add new types using classes and objects.
- (Much) more later with classes…

Primitive Types

- A primitive type is predefined by the Java programming language and named by its reserved keyword.
- The primitive types are:
  - numeric
  - boolean
  - char

Java Numeric Types

- The integer types represent whole number:

  `byte, int, short, long`

- The floating-point types represent real numbers:

  `float, double`

Integer Type Ranges

- The four separate integer primitive data types ranges and number of bits are:

Type	Storage	Min Value	Max Value
`byte`	8 bits	-128	127
`short`	16 bits	-32,768	32,767
`int`	32 bits	-2,147,483,648	2,147,483,647
`long`	64 bits	-2^{63} or < -9 x 10^{18}	2^{63}-1or > 9 x 10^{18}

Floating-point Type Ranges

- The two floating-point primitive data types ranges and number of bits are:

Type	Storage	Approximate Min Value	Approximate Max Value
`float`	32 bits	-3.4 x 10^{38}	3.4 x 10^{38}
`double`	64 bits	-1.7 x 10^{308}	1.7 x 10^{308}

- Only a very small sub-set of real numbers!

Unicode Character Type

- A char value stores a single character from the Unicode character set.
- The values range is between 0 and 65535.
- The amount of memory it requires: 16 bit or 2 bytes.
- The Unicode character set uses 16 bits per character.

Constants

- A constant is an identifier that is similar to a variable except that its value cannot be changed . You cannot assign to a constant!
- The final modifier is used for constant declaration:
  - `final <type> <Variable_name> = <value>;`
- Example:
  - `final int MAX_GRADE = 100;`

Statements

- Statements are the basic building block of any program – they are the 'instructions' that tell the computer what to do.
- "Print the results of the math":

Signifies 'end of statement'

```
System.out.println(4075+219);
```

One Statement

Print to the screen! Do the math!

Statements in Methods*

- A method contains one or more statements.
  - System.out.println("How are you?");

* Method == Function

The `main` Method

- The main method is the 'entry point' to the program. It contains the code to be executed when the program starts.
- There can be only one main method per file.
- The main method has a standard syntax:

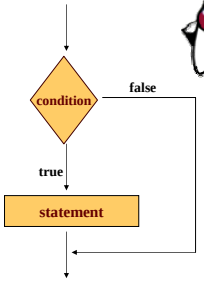
```
public static void main(String[] args){
   // your code...
}
```


Control Structures and statements

If Statement

Syntax:

```
if (condition) {
   statement;
}
```



if – else Statement

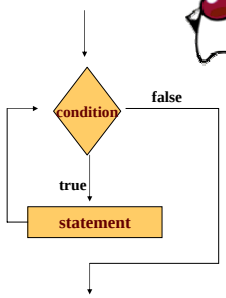
Syntax:

```
if (condition) {
      statement;
}
else {
  statement;
}
```


while Statement

Syntax:

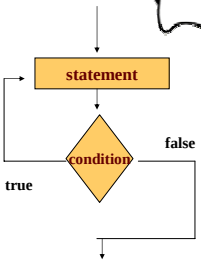
```
while (condition) {
      statement;
}
```



do - while Statement

Syntax:

```
do {
      statement;
} while (condition)
```

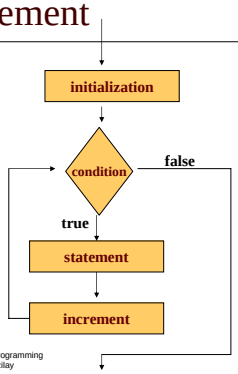


for Statement

Syntax:

```
for (initialization;  condition; increment) {
  statement;
}
```

for Statement



`switch` Statement

- The `switch` statement is a choice between doing several things (usually more then two things).
- The `switch` statement evaluates an expression, then attempts to match the result to one of a series of values.
- Execution transfers to statement list associated with the first value that matches.

`Switch` Syntax

```
switch(exp){
case value1:
     statement1;
     break;
  …
case valueN:
     statementN;
     break;
default:
     defaultStatement;
     break;
}
```


The `Break` Statement

- The `break` statement is usually used to terminate the statement list of each case. This causes the control to jump to the end of the `switch` statement and continue.
- Note: if the `break` statement is omitted execution continues (“falls”) to the next case!

Example

```
switch(letter) {
case ‘a’:
     System.out.println(“The letter was a”);
     add();
     break;
case ‘d’:
     System.out.println(“The letter was d”);
     delete();
     break;
default:
     System.out.println(“Illegal input”);
     break;
}
```

`break` inside a Loop

- The `break` statement, (already used within the `switch` statement), can also be used inside a loop
- When the `break` statement is executed, control jumps to the statement after the loop (the condition is not evaluated again)

`continue` inside a Loop

- A similar statement to the `break` is `continue` inside loops.
- When the `continue` statement is executed, control jumps to the end of the loop and the condition is evaluated (possibly entering the loop statements again).

Labeled `continue` and `break`

- In nested loops we might want to `break`/`continue` with respect to the outer loop
- As there is no `goto` in java – labels serves the missing feature

Labeled `break`

```
outer:
do {
   statement1;
   do {
            statement2;
            if ( condition ) {
                  break outer;
            }
            statement3;
   } while ( test_expr );

   statement4;

} while ( test_expr );
```


Labeled `continue`

```
test:
   do {
      statement1;
      do {
            statement2;
            if ( condition ) {
                  continue test;
            }
            statement3;
      } while ( test_expr );

      statement4;

   } while ( test_expr );
```


Packages and Import statements

Java API (Packages)

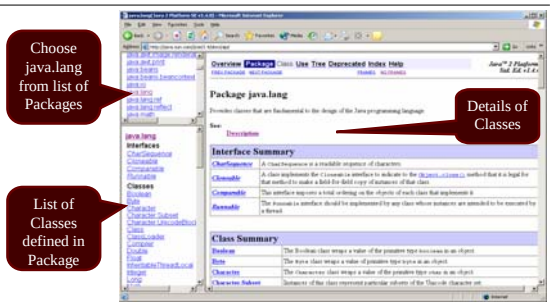
- Java comes with 3,000+ pre-designed components.
- The Java API is the library of classes supplied by Java.
- The classes in the Java API is separated into packages. Each package contains a set of classes that are related in some way.

The Java API Packages

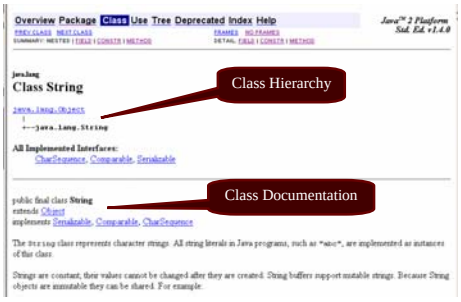
```
java.applet     java.net
java.awt        java.rmi
java.beans      java.security
java.io         java.sql
java.lang       java.text
java.math       java.util
...
```


Documentation: http://java.sun.com/j2se/1.5.0/docs/api/

java.lang



`String` Class



`String` Methods

Importing Packages

- Using a class from the Java API can be accomplished by using its fully qualified name:

```
java.util.Random random =
      new java.util.Random();
```

- Or the class can be imported once with the import statement at the top of the file:

```
import java.util.Random;
. . . Random random = new Random();
```


Importing Packages

- You can also import all the classes in a given package with a single import statement:

```
import java.util.*;
```

- The `java.lang` package is automatically imported into every Java program.

Arrays

Arrays

- An array is an object that can be used to store a list of values.
- All array elements are of the same type (primitive or objects).
- Arrays have fixed sizes, set when the array is created.

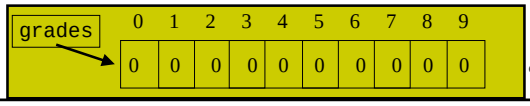
Array Elements

- A particular value in an array is referenced using the array name followed by the index in brackets.
- As in C, a java Array of size n is indexed from 0 to n-1.
- The Java interpreter will throw an exception if an array index is out of bounds.

Creating Arrays

- Array elements are initialized with there default values

```
int[] grades = new int[10];
grades[3] = 70;
grades[7] = 87;
int i = 5;
grades [i/2] = 39;
```



Initialization list

- You can declare, construct, and initialize the array all in one statement:

```
int[] primes = {2,3,5,7,11,13,17,19};
```

- This declares an array of type `int`, constructs an array of 8 slots, and assigns the designated values into the array.

```
int[] [] values = {{1,2,5}, {4,3,2,1}, {11}};
```