Software and Programming II

A brief review of basic Java constructs

Keith Leonard Mannock

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk

September 30, 2014

Java History

Java Design Goals

- Safe: Can be run inside a browser and will not attack your computer
- Portable: Run on many Operating Systems (e.g., Windows, Linux, Mac OS)

Java programs are distributed as instructions for a *virtual machine*, making them platform-independent

- Virtual machines are available for most Operating Systems. [The iPhone is a notable exception]

Java Timeline

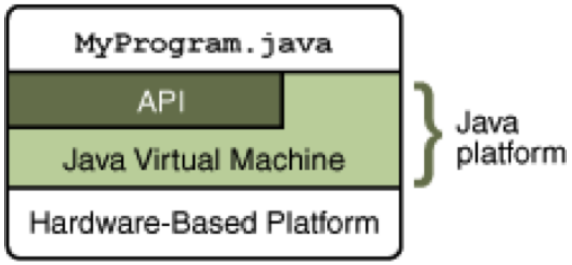
Version	Year	Important New Features
1.0	1996	
1.1	1997	Inner classes
1.2	1998	Swing, Collections framework
1.3	2000	Performance enhancements
1.4	2002	Assertions, XML support
5	2004	Generic classes, enhanced for loop, auto-boxing, enumerations, annotations
6	2006	Library improvements
7	2011	Small language changes and library improvements

Oracle purchased Sun (along with Java) in 2010

There are still quite a few references and links to Sun Microsystems which are now re-directed to Oracle

The Java API

The Java Platform consists of two parts:



1. Java Virtual Machine
2. Java API — also called libraries

The Application Programming Interface (API) is a huge collection of handy software packages that programmers can use:

Graphics, user interface, networking, sound, database, math, and many more

Structure of a Java program

A program, or project, consists of one or more packages where

$$\text{package} = \text{directory} = \text{folder}$$

Project:

- packages
- classes
- fields
- methods
- declarations
- statements

- A package contains one or more classes
- A class contains one or more fields and methods
- A method contains declarations and statements
- Classes and methods may also contain comments

We'll begin by looking at the *insides* of methods

Simple program outline

```
class MyClass {
    public static void main(String[ ] args) {
        new MyClass().run();
    }

    void run() {
        // some declarations and statements go here
        // this is the part we will talk about today
    }
}
```

- The class name (`MyClass`) must begin with a capital
- `main` and `run` are methods
- This is the form we will use for now
- Once you understand all the parts, you can vary things

Comments in Java

- Single-line comments start with `//`
- Multi-line comment start with `/*` and end with `*/`
- Documentation comments start with `/**` and end with `*/`, and are put just before the definition of a *variable*, *method*, or *class*
- Documentation comments are more heavily used in Java, and there are much better tools for working with them

Declaring variables

In Java, every variable that you use in a program must be declared (in a declaration)

- The declaration specifies the type of the variable
- The declaration may give the variable an initial value

Examples:

```
int age;
int count = 0;
double distance = 37.95;
boolean isReadOnly = true;
String greeting = "Welcome to SP2";
String outputLine;
```

Some Java data types I

Now we will look at some data types.
In Java, the most important primitive (simple) types are:

- `int` variables hold integer values
- `double` variables hold floating-point numbers (numbers containing a decimal point)
- `boolean` variables hold a `true` or `false` value

Some Java data types II

Other primitive types are

- `char` variables hold single characters
- `float` variables hold less accurate floating-point numbers
- `byte`, `short` and `long` hold integers with fewer or more digits

Some Java data types III

Another important type is the `String`

- A `String` is an `Object`, not a primitive type
- A `String` is composed of zero or more chars

Enumerated Types

Java provides an easy way to name a finite list of values that a variable can hold

- It is like declaring a new type, with a list of possible values

```
public enum FilingStatus {
 SINGLE, MARRIED,MARRIED_FILING_SEPARATELY }
```

- You can have any number of values, but you must include them all in the `enum` declaration
- You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```

- And you can use the comparison operator with them:

```
if (status == FilingStatus.SINGLE) . . .
```

Reading in numbers I

First, import the `Scanner` class:

```
import java.util.Scanner;
```

Create a scanner and assign it to a variable:

```
Scanner scanner = new Scanner(System.in);
```

The name of our scanner is `scanner`

`new Scanner(...)` says to make a new one

`System.in` says the scanner is to take input from the keyboard

Reading in numbers II

Next, its polite to tell the user what is expected:

```
System.out.print("Enter a number:  ");
```

Finally, read in the number:

```
myNumber = scanner.nextInt();
```

If you haven't previously declared the variable `myNumber`, you can do it when you read in the number:

```
int myNumber = scanner.nextInt();
```

Printing I

There are two methods you can use for printing:

- `System.out.println(something);`

  This prints something and ends the line
- `System.out.print(something);`

  This prints something and doesnt end the line (so the next thing you print will go on the same line)

Printing II

These methods will print anything, but only one thing at a time
You can concatenate values of any type with the + operator
Example:

```
System.out.println("There are " + appleCount + " apples and "
                        + orangeCount + " oranges.");
```

Program to double a number

```
import java.util.Scanner;

public class Doubler {
  public static void main(String[] args) {
    new Doubler().run();
  }

  private void run() {
    Scanner scanner;
    int number;
    int doubledNumber;
    scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    number = scanner.nextInt();
    doubledNumber = 2 * number;
    System.out.println("Twice " + number + " is " + doubledNumber);
    scanner.close();
  }
}
```

Assignment statements

- Values can be assigned to variables by assignment statements
- The syntax is: `variable = expression;`
- The expression must be of the same type as the variable
- The expression may be a simple value or it may involve computation

Examples:

```
name = "Oded";
count = count + 1;
area = (4.0 / 3.0) * 3.1416 * radius * radius;
isReadOnly = false;
```

When a variable is assigned a value, the old value is discarded and totally forgotten

Methods

A method is a named group of declarations and statements

```
void tellWhatYearItIs( ) {
    int year = 2025;
    System.out.println("Hello in " + year + "!");
}
```

We *call* or *invoke* a method by naming it in a statement:

```
tellWhatYearItIs();
```

This should print out `Hello in 2025!`

Method types and returns

- Every method definition must specify a return type
- `void` if nothing is to be returned
- Every method parameter must be typed
- Example: `double average(int[] scores) { }`
- The return type is `double`, the parameter type is `int[]`
- If a method returns `void` (nothing), you may use plain `return` statements in it
- If you reach the end of the method, it automatically returns
- If a method returns something other than `void`, you must supply return statements that specify the value to be returned
- Example: `return sum / count;`

Method calls

- A method call is a request to an object to do something, or to compute a value
- `System.out.print(expression)` is a method call; you are asking the `System.out` object to evaluate and display the expression
- When you call a method, do not specify parameter
- You must provide parameters of the type specified in the method definition
- A method call may be used as a statement
- Example: `System.out.print(2 * pi * radius);`
- Some method calls return a value, and those may be used as part of an expression
- Example: `h = Math.sqrt(a * a + b * b);`

Organisation of a class

- A class may contain data declarations and methods (and constructors, which are like methods), but not statements
- A method may contain (temporary) data declarations and statements
- A common error:

```
public class Example {
  int variable; // simple declaration is OK
  int anotherVariable = 5; // declaration with initialization is OK
  variable = 5; // statement! This is a syntax error

  void someMethod() {
    int yetAnotherVariable; // declaration is OK
    yetAnotherVariable = 5; // statement inside method is OK
  }
}
```

Arithmetic expressions

Arithmetic expressions may contain:

- `+` to indicate addition
- `-` to indicate subtraction
- `*` to indicate multiplication
- `/` to indicate division
- `%` to indicate remainder of a division (integers only)
- parentheses `( )` to indicate the order in which to do things
- An operation involving two `int`s results in an `int`
- When dividing one `int` by another, the fractional part of the result is thrown away, e.g., `14 / 5` gives `2`
- Any operation involving a double results in a double, e.g., `3.7 + 1` gives `4.7`

Boolean expressions I

Arithmetic comparisons result in a boolean value of true or false
There are six comparison operators:

<	less than
<=	less than or equals
>	greater than
>=	greater than or equals
==	equals
!=	not equals

Boolean expressions II

There are three boolean operators:

`&&`	and	true only if both operands are true
`||`	or	true if either operand is true
`!`	not	reverses the truth value of its one operand

Example:

```
(x > 0) && !(x > 99)
```

"x is greater than zero and is not greater than 99"

String concatenation

You can concatenate (join together) `Strings` with the `+` operator

Example:

```
fullName = firstName + "  " + lastName;
```

In fact, you can concatenate any value with a `String` and that value will automatically be turned into a `String`

Example:

```
System.out.println("There are " + count + " apples.");
```

String concatenation (II)

Be careful, because + also still means addition

```
int x = 3;
int y = 5;
System.out.println(x + y + " != " + x + y);
```

The above prints `8 != 35`

Addition is done left to right — use parentheses to change the order

`if` statements I

An `if` statement lets you choose whether or not to execute one statement, based on a boolean condition

Syntax:

```
if (boolean_condition) statement;
```

Example:

```
if (x < 100) x = x + 1;
// adds 1 to x, but only if x is less than 100
```

Please note: the condition must be boolean

`if` statements II

An `if` statement may have an optional `else` part, to be executed if the boolean condition is false

Syntax:

```
if (boolean_condition) statement;
 else statement;
```

Example:

```
if (x >= 0 && x < limit)
    y = x / limit;
else
    System.out.println("x is out of range: " + x);
```

Compound statements

Multiple statements can be grouped into a single statement by surrounding them with braces, `{ }`

Example:

```
if (score > 100) {
        score = 100;
        System.out.println("score has been adjusted");
}
```

Unlike other statements, there is no semicolon after a compound statement

Braces can also be used around a single statement, or no statements at all (to form an empty statement)

Compound statements (II)

It is good style to always use braces in the if part and else part of an if statement, even if the surround only a single statement Indentation and spacing should be as shown in the above example

`while` loops

A `while` loop will execute the enclosed statement as long as a boolean condition remains true

Syntax:

```
while (boolean_condition) statement;
```

Example:

```
n = 1;
while (n < 5) {
       System.out.println(n + " squared is " + (n * n));
       n = n + 1;
 }
```

Result:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

Please note: the condition must be boolean

Danger: If the condition never becomes false, the loop never exits, and the program never stops

The `do-while` loop

The syntax for the do-while is:

```
do {
    any number of statements
} while (condition) ;
```

- The `while` loop performs the test first, before executing the statement
- The `do-while` statement performs the test afterwards

As long as the test is true, the statements in the loop are executed again

The increment operator

++ adds 1 to a variable

- It can be used as a statement by itself
- It can be used as part of a larger expression, but this is very bad style (see next slide)
- It can be put before or after a variable
  - If before a variable (preincrement), it means to add one to the variable, then use the result
  - If put after a variable (postincrement), it means to use the current value of the variable, then add one to the variable
  - When used as a statement, preincrement and postincrement have identical results

Examples of ++

```
int a = 5;
a++;
// a is now 6

int b = 5;
++b;
// b is now 6

int c = 5;
int d = ++c;
// c is 6, d is 6
```

```
int e = 5;
int f = e++;
// e is 6, f is 5

int x = 10;
int y = 100;
int z = ++x + y++;
// x is 11, y is 101, z is 111
```

This last example is confusing code and therefore bad code, so this is very poor style

The decrement operator

-- subtracts 1 from a variable

It acts just like ++, and has all the same problems

The `for` loop

The `for` loop is complicated, but very handy

Syntax:

```
for (initialise ; test ; increment) statement ;
```

Notice that there is no semicolon after the increment

Execution:

- The initialise part is done first and only once
- The test is performed; as long as it is true,
- The statement is executed
- The increment is executed

Parts of the `for` loop

Initialise: In this part you define the loop variable with an assignment statement, or with a declaration and initialisation

Examples: `i = 0` `int i = 0` `i = 0, j = k + 1`

Test, or condition: A boolean condition

Just like in the other control statements we have used

Increment: An assignment to the loop variable, or an application of ++ or -- to the loop variable

Example for loops

Print the numbers 1 through 10, and their squares:

```
for (int i = 1; i < 11; i++) {
    System.out.println(i + "  " + (i * i));
}
```

Print the squares of the first 100 integers, ten per line:

```
for (int i = 1; i < 101; i++) {
    System.out.print("  " + (i * i));
    if (i % 10 == 0) System.out.println();
}
```

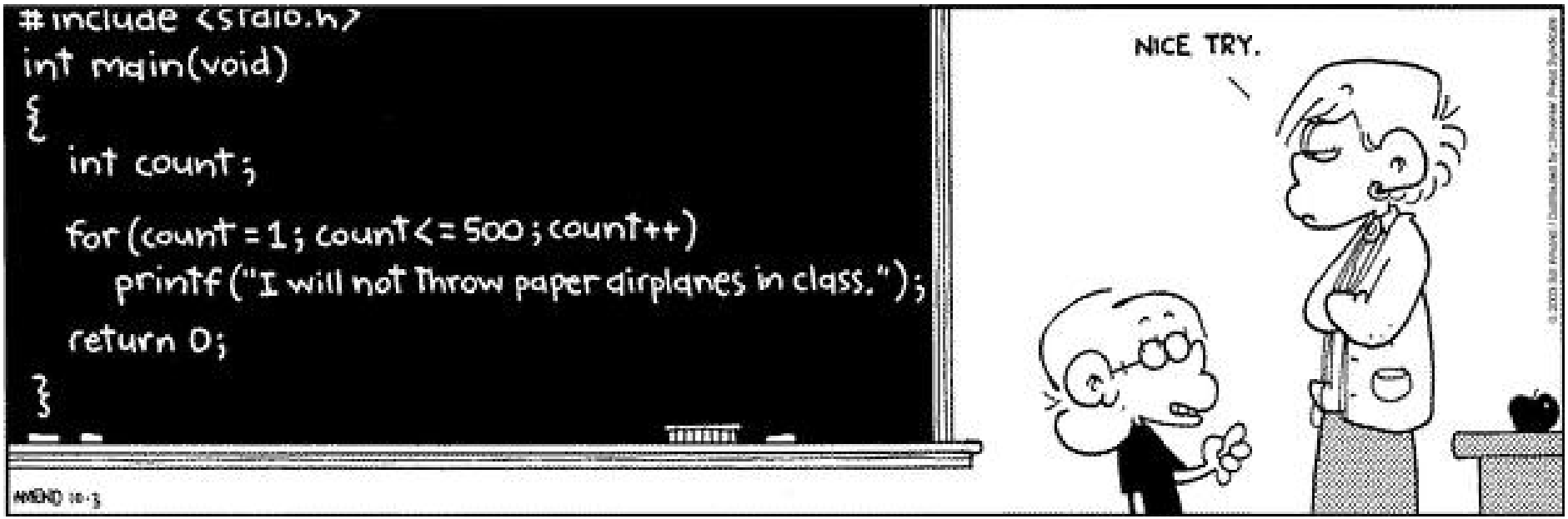
Example: Multiplication table

```
public class Multiplication {
  public static void main(String[] args) {
    for (int i = 1; i < 11; i++) {
      for (int j = 1; j < 11; j++) {
        int product = i * j;
        if (product < 10)
          System.out.print("   " + product);
        else
          System.out.print("  " + product);
      }
      System.out.println();
    }
  }
}
```

When do you use each loop?

- Use the for loop if you know ahead of time how many times you want to go through the loop
- Example: Stepping through an array
- Example: Print a 12-month calendar
- Use the while loop in almost all other cases
- Example: Compute the next step in an approximation until you get close enough
- Use the do-while loop if you must go through the loop at least once before it makes sense to do the test
- Example: Ask for the password until user gets it right

`for` loops


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf ("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.

The `break` statement

- Inside any loop, the `break` statement will immediately get you out of the loop
- If you are in nested loops, `break` gets you out of the innermost loop
- It doesnt make any sense to break out of a loop unconditionally — you should do it only as the result of an `if` test
- Example:

```
for (int i = 1; i <= 12; i++) {
    if (badEgg(i)) break;
}
```

- `break` should not be the normal way to leave a loop
- Use it when necessary, but dont overuse it

The `continue` statement

- Inside any loop, the `continue` statement will start the next pass through the loop
- In a `while` or `do-while` loop, the `continue` statement will bring you to the test
- In a `for` loop, the `continue` statement will bring you to the increment, then to the test

Multiway decisions

- The `if-else` statement chooses one of two statements, based on the value of a boolean expression
- The `switch` statement chooses one of several statements, based on the value

Syntax of the switch statement

The syntax is:

```
switch (expression) {
    case value1 :
        statements ;
        break ;
     case value2 :
        statements ;
        break ;
     ...(more cases)...
     default :
        statements ;
        break ;
}
```

Syntax of the switch statement (II)

- A switch works with the `byte`, `short`, `char`, and `int` primitive data types.
- It also works with enumerated types, the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer`
- Notice that colons (`:`) are used as well as semicolons
- The last statement in every case should be a break;
- Some programmers even like to do this in the last case (a C++ hangover)
- The `default:` case handles every value not otherwise handled

Syntax of the switch statement (III)

Example:

```
switch (cardValue) {
  case 1:
    System.out.print("Ace");
    break;
  case 11:
    System.out.print("Jack");
    break;
  case 12:
    System.out.print("Queen");
    break;
  case 13:
    System.out.print("King");
    break;
  default:
    System.out.print(cardValue);
    break;
}
```

The `assert` statement

The purpose of the `assert` statement is to document something you believe to be true

There are two forms of the assert statement:

```
assert booleanExpression;
```

- This statement tests the boolean expression
- It does nothing if the boolean expression evaluates to true
- If the boolean expression evaluates to false, this statement throws an `AssertionError`

The `assert` statement (II)

```
assert booleanExpression : expression;
```

- This form acts just like the first form
- In addition, if the boolean expression evaluates to false, the second expression is used as a detail message for the `AssertionError`
- The second expression may be of any type except `void`

Enabling assertions

- By default, Java has assertions disabled — that is, it ignores them
- This is for efficiency
- Once the program is completely debugged and given to the customer, nothing more will go wrong, so you dont need the assertions any more
- Yeah, right!

A complete program

```
// print the square roots of the numbers in the range 1 to 10
// using a while loop
public class SquareRoots {
  public static void main(String[] args) {
    int n = 1;
    while (n <= 10) {
      System.out.println(n + "\t" + Math.sqrt(n));
      n++;
    }
  }
}
```

```
1   1.0
2   1.4142135623730951
3   1.7320508075688772
4   2.0
5   2.23606797749979
etc.
```

Another complete program

```
public class LeapYear {

  public static void main(String[] args) {
    int start = 1990;
    int end = 2015;
    int year = start;
    boolean isLeapYear;

    while (year <= end) {
      isLeapYear = year % 4 == 0;
      // a leap year is a year divisible by 4...
      if (isLeapYear && year % 100 == 0) {
        // ...but not by 100...
        isLeapYear = year % 400 == 0;
        // ...unless its also divisible by 400
      }
      if (isLeapYear) {
        System.out.println(year + " is a leap year.");
      }
      year = year + 1;
    }
  }
}
```

```
1992 is a leap year.
1996 is a leap year.
2000 is a leap year.
2004 is a leap year.
2008 is a leap year.
2012 is a leap year.
```

And yet another example. . .

```
import java.util.Random;

public class RandomWalk {
  int x = 0;
  int y = 0;
  Random rand = new Random();

  public static void main(String[] args) {
    new RandomWalk().run();
  }

  void run() {
    double distance = 0;
    while (distance < 10) {
      step(3);
      System.out.println("Now at " + x + ", " + y);
      distance = getDistance();
    }
  }

  void step(int maxStep) {
    x += centreAtZero(maxStep);
    y += centreAtZero(maxStep);
  }

  int centreAtZero(int maxStep) {
    int r = rand.nextInt(2 * maxStep + 1);
    return r - maxStep;
  }

  double getDistance() {
    return Math.sqrt(x * x + y * y);
  }

}
```

Questions