

Zaawansowane programowanie obiektowe

Lecture 1

Szymon Grabowski
sgrabow@kis.p.lodz.pl
<http://szgrabowski.kis.p.lodz.pl/zpo18/>

Łódź, 2018



Recommended literature / resources

<http://download.oracle.com/javase/tutorial/index.html>

Cay S. Horstmann, Gary Cornell, *Core Java: Volume I – Fundamentals* (Sept. 2018, 11th Ed.),
Volume II – Advanced Features (Sept. 2018, 11th Ed). Prentice Hall.

<http://horstmann.com/corejava/corejava.zip>

In Polish: Wyd. X, 2016-17 (Helion).

Cay S. Horstmann, *Core Java for the Impatient*.
Addison-Wesley 2015.

Cay S. Horstmann, *Java SE 9 for the Really Impatient*.
Addison-Wesley 2017.

Joshua Bloch, *Effective Java*. Addison-Wesley, 2018 (3rd Ed.).
<https://github.com/jbloch/effective-java-3e-source-code>

Bruce Eckel, *Thinking in Java*. Prentice Hall, 2006 (4th Ed.).

Paul Deitel, Harvey Deitel, *Java for Programmers*. Prentice Hall, 2011.

A little bit of history

Developed by Sun in 1995–96. Based on C++.

Versions: 1.0.x, 1.1.x etc. up to SE 11 now (since Sept., 2018).

Three editions:

standard (SE), enterprise (Jakarta EE) and micro (ME).

For example, J2ME is for PDA's (e.g. cell phones) – lighter.

Android SDK: new(ish) platform, old (Java) language.

Design goals: portability, high-level, rich std library.

What makes Java safe

Garbage collector.

(No *delete* for objects, arrays...

Objects no longer used will be deallocated automatically.)

Exceptions everywhere.

(In C++ they were optional.

Here they are mandatory.)

Applets – sandbox model.

(Some operations not allowed, unless perhaps the applet's digital signature is approved.)

References, no (explicit) pointers.

Strong typing. No implicit conversions (except some safe cases, see later), no `int` → `boolean` conversion.

Formatted output

[<http://mcsp.wartburg.edu/zelle/python/ccsc-html/mgp00005.html>]

The March of Progress (Cay Horstmann)

- C | Pascal

```
printf("%10.2f", x); | write(x:10:2)
```

- C++

```
cout << setw(10) << setprecision(2)  
    << showpoint << x;
```

- Java

```
java.text.NumberFormat formatter  
    = java.text.NumberFormat.getNumberInstance();  
formatter.setMinimumFractionDigits(2);  
formatter.setMaximumFractionDigits(2);  
String s = formatter.format(x);  
for (int i = s.length(); i < 10; i++)  
    System.out.print(' ');  
System.out.print(s);
```

No longer needed
since J2SE 5.0 !

Use
`System.out.printf
("%10.2f", x);`

jshell (Java 9+)

```
f:\szkola\zpo2018>jshell  
| Welcome to JShell -- Version 9.0.1  
| For an introduction type: /help intro
```

```
jshell> String s = "hello"  
s ==> "hello"
```

```
jshell> s.e  
endsWith(      equals(      equalsIgnoreCase(
```

```
jshell> s.endsWith("lo")  
$2 ==> true
```

```
jshell> List<Integer> li = List.of(3, 5, 1);  
li ==> [3, 5, 1]
```

```
jshell> /exit  
| Goodbye
```

Java-specific math operators

& and | for booleans mean almost the same as && and ||, except for full evaluation.

In Java, bitwise &, | and ^ (xor) still exist though.

E.g. `int n = 10 & 6 | 5; // n==7`

Java specific bitwise ops: `>>>`, `>>>=`

`>>>` is unsigned shift right:

zeroes are inserted at the higher-order bits.

But the signed right shift `>>` uses sign extension:

if the value is positive, 0s are inserted at the higher-order bits;

if the value is negative, 1s are inserted at the higher-order bits.

```
int a = -11;  
PrintStream so = System.out;  
so.println(a >> 1); // -6  
so.println(a >>> 1); // 2147483642
```

```
a = 11;  
so.println(a >> 1); // 5  
so.println(a >>> 1); // 5
```

8 primitive data types

char (single characters in **Unicode**, **2 bytes!**, from `'\u0000'` to `'\uffff'`) – formally it's an (unsigned) integer type

boolean (true or false; 1 byte)

integer types (**all signed**):

byte (8 bits)

short (16 bits)

int (32 bits)

long (64 bits)

floating point types:

float (32 bits)

double (64 bits)

Examples:

```
float x = 3.5f, y = -5.92F, z;
```

```
char c = '\u1f00', euroSign = '\u20ac';
```

```
long theKeyToUniverse = 42353265264254L;
```

```
int hexVar = 0x1A, octVar = 021; // 26, 17
```

```
int binVar = 0b1011;
```

Optional underscores:

```
int n = 2_000_000; // also possible in floats
```

```
But not: int n = _124;
```

```
// _124 could be an identifier
```


Wrapper classes (Boolean, Integer, Character, ...)

Wrappers are classes that **wrap up primitive values in classes** that offer utility methods to manipulate the values.

Necessary for collections (e.g. ArrayList<int> impossible!).

Objects of wrapper classes are immutable.

Byte, Double, Float, Integer and Short extend the abstract Number.

All 8 wrapper classes are **public final** (i.e., cannot be extended).

```
Integer i = new Integer(20); // deprecated in Java 9, use Integer.valueOf
```

```
byte b = i.byteValue();
```

```
double d = Double.parseDouble("3.14");
```

```
int i = Integer.parseInt("10011110", 2);
```

```
String s = Integer.toHexString(130).toUpperCase();
```

Finite or infinite loop?

```
for (int i = Integer.MIN_VALUE; i <= Integer.MAX_VALUE; i++)  
    doSomething(i);
```

Immutability

Immutable means that the
object state cannot be changed.
(But we can change the reference
(if non-final) to the object,
i.e. point to another object.)

What does it print?

```
Integer i = Integer.valueOf(5);  
Integer b = i; // b references same integer as i  
i = 6; // modify i  
System.out.println(i + "!=" + b);
```

Careful with wrapper classes

```
int x = 5;  
int y = 5;
```

```
Integer x1 = Integer.valueOf(5);  
Integer y1 = Integer.valueOf(5);
```

```
Integer x2 = new Integer(5);  
Integer y2 = new Integer(5);
```

```
System.out.println(x == y); // true  
System.out.println(x1 == y1); // true (!)  
System.out.println(x2 == y2); // false
```

```
private Boolean flag; // a class field
```

```
...
```

```
if (flag == true)  
    System.out.println("Flag is set");  
else  
    System.out.println("Flag is not set");
```

java.lang.NullPointerException

(Auto)boxing

You can't put an int (or other primitive value) into a collection. You must convert it to Integer first. It is called **boxing** a primitive value into a wrapper class instance.

Or the other way round: when you take an object out of the collection, you get e.g. an Integer; if you need a primitive you must **unbox** the Integer using the *intValue* method.

Autoboxing magically eliminates the extra effort.

```
HashMap<String, Integer> hm = new HashMap<String, Integer>();  
hm.put("speed", 20);  
int value = hm.get("speed");  
// i.e. the compiler invisibly translates the line above into  
// int value = hm.get("speed").intValue();
```

Autoboxing, warnings [Core Java, vol. 1, ch. 5; <http://www.informit.com/articles/article.aspx?p=1021579&seqNum=4>]

```
Integer n = 5;  
n++; // it works!
```

But the **illusion** that primitive types and their wrappers are the same thing is **not complete**:

after

```
Integer a = 1000; Integer b = 1000;
```

you shouldn't write: `if (a == b) ...`

but of course: `if (a.equals(b)) ...`

Object can be null, primitives cannot be null
(unboxing a null results in a `NullPointerException`).

`x.equals(y)` won't compile if `x` is a primitive.

Unsigned arithmetic (Java 8+)

`Byte.toUnsignedInt(b)` returns an int value between 0 and 255
(0 -> 0, 1 -> 1, ..., 127 -> 127, -128 -> 128, ..., -1 -> 255).

`1000000000 * 3` evaluates to -1294967296 (int type!),
yet
`Math.multiplyExact(1000000000, 3)` will throw `ArithmeticException`.

Similar methods (with int and long arguments): `addExact`,
`subtractExact`, `incrementExact`, `decrementExact`, `negateExact`.

Data conversion

What is *b*:

```
double b = 1 / 3; // ?
```

Just like in C/C++: integer division, result: 0.
To obtain 0.3333333... use 1.0 / 3.0.

And now, what is *c*?

```
int c = 1.0 / 3.0;
```

Compile error.

Java wants to make sure you know
you'd lose fractional information.

Solution:

```
int c = (int)(1.0 / 3.0);
```

Type conversions (arithmetic)

int i;

int2int

short sh;

sh = 5; // OK; 5 is int but is constant and
// is small enough (no information loss)

i = sh; // expanding, ok

sh = (short)i; // OK – with the explicit cast

double d = 5.12;

float2int

i = d; // WRONG! Conversion must be explicit

// but:

i = (int)d; // OK. The fraction is cut off

float width = 5.5; // compile error

Integer arithmetic

Java performs **integer arithmetic at the int level**, so e.g. $b+c$, where b and c are of type short, returns an int.

The sum must therefore be **cast** to a short before an assignment to a , because a is a short:

```
short a, b, c;  
c = 21; b = 9;  
a = (short)(b + c);  
// without casting – a compile error
```

Similar example:

```
byte a = 4, b = -1;  
short c = (short)(a+b);
```

java.math.BigInteger

A class to represent **arbitrarily large integers (signed)**.

Supports add, substr, mul, div etc. –
and less standard operations:

GCD calculation, primality testing, prime generation,
bit manipulation and more.

```
BigInteger b1 = new BigInteger(new String("12111301"), 4);  
// in base 4; it's 25969  
BigInteger b2 = new BigInteger(new String("59")); // base 10  
BigInteger b3 = b1.divide(b2);  
// 25969 / 59 = 440 (integer division)  
BigInteger b4 = BigInteger.valueOf(350L);  
int i = b4.intValue(); // there are: doubleValue(), longValue()...  
System.out.println(b4 + " = " + i);
```

BigInteger example [Core Java, vol. 1, ch. 3,

http://authors.phptr.com/corejava/downloads/coreJava6v1_0130471771.zip]

```
import javax.swing.*;
import java.math.*;

public class BigIntegerTest {
    public static void main(String[] args) {
        String input = JOptionPane.showInputDialog
            ("How many numbers do you need to draw?");
        int k = Integer.parseInt(input);
        input = JOptionPane.showInputDialog
            ("what is the highest number you can draw?");
        int n = Integer.parseInt(input);

        /*
         compute binomial coefficient
         n * (n - 1) * (n - 2) * . . . * (n - k + 1)
         -----
         1 * 2 * 3 * . . . * k
        */
        BigInteger lotteryOdds = BigInteger.valueOf(1);

        for (int i = 1; i <= k; i++)
            lotteryOdds = lotteryOdds
                .multiply(BigInteger.valueOf(n - i + 1))
                .divide(BigInteger.valueOf(i));

        System.out.println("Your odds are 1 in " + lotteryOdds); +
        System.exit(0);
    }
}
```

java.math.BigDecimal

[J. Bloch, *Effective Java*, 2001, Chap. 7]

```
// Broken - uses floating point for monetary calculation!
public static void howManyCandies1() {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = .10; funds >= price; price += .10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

Bad!

```
public static void howManyCandies2() {
    final BigDecimal TEN_CENTS = new BigDecimal( ".10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
        funds.compareTo(price) >= 0;
        price = price.add(TEN_CENTS)) {
        itemsBought++;
        funds = funds.subtract(price);
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}
```

OK

Arrays

```
int[][] tab = new int[8][3]; // 8 rows, 3 columns
```

```
int[][] c = { {5, 8},  
              {1, 2, 9},  
              {7, 4, 2}  
            };
```

```
// What is c[1][2] ?
```

```
// What is c.length ?
```

```
int[] a = {1, 2, 3};
```

```
int[] b = a;
```

```
b[0] = 10; // a[0] also changed
```

Create and initialize:

```
int[] n = {10, 20, 30, 40};
```

```
// or: ... = new int[]{10, 20, 30, 40};
```

Coping with array copying

There is a **copy method in java.lang.System:**

`arraycopy(sourceArr, src_pos, destArr, dest_pos, length);`

Example: `System.arraycopy(b, 0, a, 0, b.length);`

Exceptions possible: `NullPointerException` (if `sourceArray` or `destArray` null), `IndexOutOfBoundsException` (e.g. `src_pos` negative), or `ArrayStoreException` (element type mismatch).

```
class ArrayCopyDemo {  
    public static void main(String[] args) {  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                             'i', 'n', 'a', 't', 'e', 'd' };  
        char[] copyTo = new char[7];  
  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        System.out.println(new String(copyTo));  
    }  
}
```

Output:
caffein

Array copy in a modern way (since v1.6)

```
int[] arr = {1, 3, 5, 7};  
int[] arrCopy = Arrays.copyOf(arr, arr.length);  
// similar versions to other primitive types,  
// plus a generic version which returns T[]
```

Why usually **better than System.arraycopy**?

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

The destination (to) array must have sufficient space to hold the copied elements.

java.util.Arrays.copyOf(...), general use:
copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

Loops and selection – (mostly) from C/C++

Loops:

for(exp1; exp2; exp3) ...

for(T x: array_of_type_T) ... // for-each

while(expr) ...

do ... while(expr);

Selection:

```
switch (expr) {  
    case var1: inst1;  
    case var2: instr2;  
    ...  
    default: instr_def;  
}
```

```
Scanner sc = new Scanner(System.in);  
String s = sc.next();  
switch (s.toLowerCase()) {  
    case "y":  
    case "yes":  
        System.out.println("You said yes."); break;  
    case "n":  
    case "no":  
        System.out.println("You said no."); break;  
    default:  
        System.out.println("What did you say?"); break;  
}
```

Java 7

Not quite C/C++ constructs

break and *continue* made more flexible.

```
int i = 0, j = 0;
outerloop: // label
while (i < 100)
{
    i++;
    while (true)
    {
        j++;
        if (i + j > 10)
            break outerloop; // escapes to (*)
    }
} // (*)
```

A reasonable
compromise between
stiff *break / continue*
in C/C++
and most flexible but
messy *goto*
(in many languages).

Enhanced *for* (also called for-each)

The *for* loop can **iterate over collections and arrays**.

```
for (Object o: someCollection) { System.out.println(o); }  
for (String s: someStringArray) { System.out.println(s); }
```

```
void cancelAll(Collection<TimerTask> c) {  
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )  
        i.next().cancel();  
}
```



```
void cancelAll(Collection<TimerTask> c) {  
    for (TimerTask t : c)  
        t.cancel();  
}
```

Old *for*, find a bug!

[<http://download.oracle.com/javase/1.5.0/docs/guide/language/foreach.html>]

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// BROKEN - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));

// FIXED
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    Suit suit = (Suit) i.next();
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(suit, j.next()));
}
```

Modern style:

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

Enhanced *for*, final words

[<http://download.oracle.com/javase/1.5.0/docs/guide/language/foreach.html>]

Unfortunately, **you cannot use the for-each loop everywhere.**

Can't use it when your code accesses to the iterator (e.g., in order to remove the current element).

The for-each loop hides the iterator, so you **cannot call remove.**

It is **not usable** for loops where you need **to replace** elements in a list or array as you traverse it.

Finally, it is **not usable** for loops that must iterate **over multiple collections in parallel.**

var keyword (Java 10)

Type inference is a feature known from many languages, e.g. Python (`x = 3`; and Python 'knows' `x` is of type `int`), C++ (keyword `auto`), C# (`var`).

Now in Java: **local-variable type inference**.

```
var x = 5; // use with initialization!  
var y = 2;  
var z = application.sum(x, y);
```

```
public int sum(int x, int y) {  
    return x + y;  
}
```

```
var numbers = List.of(3, 2, 7);  
for (var nr: numbers)  
    System.out.print(nr + " ");
```

Fields, method signatures, and catch clauses still require manual type declaration!

var – use carefully!

var hides the type of the right-hand-side (RHS) expression (e.g., invoked function).

```
var res = getResult(); // what type is it?
```

With var you won't declare (=use on the LHS) a supertype.

```
var myMap = new HashMap<String, Integer>();
```

means:

```
HashMap<String, Integer> myMap = new HashMap<>();
```

which is not always good if you want to deal with the Map interface, not its implementation.

Legal assignments, casts

[members.tripod.com/~psdin/comlang/basicJava.PDF]

refOfSuper = refOfSub is valid

// refOfSub = refOfSuper is not valid: compile error
(even when the refOfSuper really refers to the sub object)

refOfSub = (Sub) refOfSuper is valid

may cause **ClassCastException** at run-time if

- refOfSuper actually refers to a super object,
- refOfSuper refers to object of another subclass of that super class

Reference assignment rules

[members.tripod.com/~psdin/comlang/basicJava.PDF]

(Enforced at compile-time.)

```
SourceType srcRef;  
DestinationType destRef = srcRef;
```

If SourceType is a **class** type, then

- either DestinationType is a superclass of SourceType;
- or DestinationType is an interface type which is implemented by the class SourceType.

Reference assignment rules, cont'd

[members.tripod.com/~psdin/comlang/basicJava.PDF]

If SourceType is an **interface** type, then

- either DestinationType is Object;
- or DestinationType is a superinterface of subinterface SourceType.

If SourceType is an **array type**, then

- either DestinationType is Object;
- or DestinationType is an array type, where the element type of SourceType can be converted to the element type of DestinationType.

Assignments and casts – example

[members.tripod.com/~psdin/comlang/basicJava.PDF]

```
class A{ }      class B extends A{ }      class C{ }
```

```
public class Class1 {  
    public static void main(String[] args) {  
        A a; B b; C c;  
        a = new A();  
        b = new B();  
        // b = (B)a; // ok? WRONG. Will throw java.lang.ClassCastException.  
        a = b; // ok? OK. a ref to subclass B object.  
        // b = a; // ok? WRONG. Compile error; can't implicitly convert.  
        // c = (C)a; // ok? WRONG. Compile error; can't convert A to C.  
        b = (B)a;  
        a = new A();  
        // b = (B)a; // ok? WRONG. java.lang.ClassCastException.
```

Long division

[J. Bloch, N. Gafter, *Java Puzzlers*]

```
public class LongDiv
{
    public static void main(String[] args)
    {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY); // guess what?
    }
}
```

Output: 5

Correction: ... = 24L * ...

(only in MICROS_PER_DAY is enough)

Passing arguments to methods

Passing variables in Java always **by value** – a copy is made. No change of the variable in the called method affects its value when back in the caller.

Aren't some variables passed **by reference** though??

No. The **reference's (memory location's) copy is passed** to the second method.

This works for objects (incl. arrays).

References

Declaring an object is in fact only declaring a reference to it (no memory allocation for the object itself).

This is different than with simple types (e.g. int)!

```
Button b; // declaring a reference to a Button object
```

```
b = new Button(); // now we create the object
```

(Kind of) a special case: String objects.

```
String s = "George Bush"; // !!
```

What about

```
String s = new String("George Bush"); // ??
```

It's correct too, but it's beating around the bush.

Little trap?

```
String s1 = "abc";  
String s2 = "ab";  
s2 += "c";  
boolean result = (s1 == s2); // true?
```

False. Different objects (i.e., their references point to different memory locations). Content irrelevant.

How to compare content then?

if (s1.equals(s2)) ... or if (s1.equalsIgnoreCase(s2)) ...

```
String s1 = "abc"; String s2; s2 = s1; // copy?
```

No, just another reference.

...But write then the line: s1 += "d";

What is s2 now?

Concatenating strings and numbers

```
String s1 = "Johnny has drunk ";
```

```
int count1 = 2, count2 = 1;
```

```
String s2 = " beers.";
```

```
String s3 = s1 + count1 + s2;
```

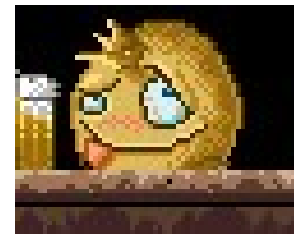
```
String s4 = s1 + (count1 + count2) + s2;
```

```
String s5 = s1 + count1 + count2 + s2;
```

```
String s6 = count1 + count2 + s2;
```

What do s3, s4, s5, s6 store?

Remember: evaluation **from left to right**.



String literals are “interned” by the JVM

```
...  
static final String fs1 = "abc";  
static final String fs2 = "abc";  
  
public static void main(String[] args)  
{  
    String s1 = new String("Krowa");  
    String s2 = "Krowa";  
    String s3 = "Krowa";  
    System.out.println("s1 equals s2 ? " + (s1.equals(s2)));  
    System.out.println("s1 == s2 ? " + (s1 == s2));  
    System.out.println("s2 == s3 ? " + (s2 == s3));  
    System.out.println("fs1 == fs2 ? " + (fs1 == fs2));  
}
```

s1 equals s2 ? true

s1 == s2 ? false

s2 == s3 ? true

fs1 == fs2 ? true

More on strings (examples)

```
String c = "Abcde ".substring(2,4);
if (c.startsWith("b")) System.err.println("What the hell?!");
c += " ";
String d = c.trim();      // leading and trailing whitespace removed
float f = 5.001f;
String s = String.valueOf(f); // s.equals("5.001") == true now
c = c.toUpperCase(); // what if we write just: c.toUpperCase(); ?
System.out.println(c.length());

String s = "Abcde ".substring(2, 8); // StringIndexOutOfBoundsException
String s = "Abcde ".substring(2, Math.min(8, "Abcde ".length())); // correct!

String s = "abracadabra";
System.out.print(s.lastIndexOf("ab")); // 7
System.out.print(s.lastIndexOf("abb")); // -1, not found

String message = String.join("-", "This", "is", "it");
// returns "This-is-it"
```

String vs StringBuffer

Objects of class String are **immutable** in Java.

```
String s = "test";  
s[0] = 'b'; // compile-error: array required, but java.lang.String found
```

There is `charAt(int i)` method in String, but only to read.

So, how can we e.g. append a character to a String?!

Short answer: we cannot. ☹️

We need to create another String:

```
String s1 = "ice-cream";  
s1 = s1 + "s"; // SLOW esp. if s1 is long; creates a temp object
```

From Java 8 spec (§15.18.1):

*To increase the performance of repeated string concatenation, a **Java compiler may use the StringBuffer** class or a similar technique to reduce the number of intermediate String objects that are created by evaluation of an expression.*

String vs StringBuffer, cont'd

[<http://java.sun.com/javase/6/docs/api/java/lang/StringBuffer.html>]

Class **StringBuffer** – smth like String but its **objects can be modified**.

```
StringBuffer sb = new StringBuffer(new String("start"));
```

```
sb.append("le"); // "startle"
```

or:

```
sb.insert(4, "le"); // start → starlet
```

Why String immutable?

- thread-safe. Two threads can both work on an immutable object at the same time without any possibility of conflict;
- immutable objects are much better suited to be Hashtable/HashMap etc. keys.

If you change the value of an object that is used as a hash table key without removing it and re-adding it, you lose the mapping.

java.util.StringJoiner, joining Collector (Java 8)

StringJoiner is used to construct a sequence of characters separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix.

The String "[George:Sally:Fred]" may be constructed as follows:

```
StringJoiner sj = new StringJoiner(":", "[", ""]);  
sj.add("George").add("Sally").add("Fred");  
String desiredString = sj.toString();
```

```
List<Person> list = Arrays.asList(  
    new Person("John", "Smith"),  
    new Person("Anna", "Martinez"),  
    new Person("Paul", "Watson ")  
);
```

```
String joinedFirstNames = list.stream()  
    .map(Person::getFirstName)  
    .collect(Collectors.joining(", ")); // "John, Anna, Paul"
```

Scanner class

Breaks its input into tokens using a delimiter pattern (by default: whitespace). Convenient nextXxxx() methods for type conversion. When scanner closed → can't read, IllegalStateException.

Examples:

```
Scanner sc = new Scanner(new File("myNumbers"));
while(sc.hasNextLong()) long aLong = sc.nextLong();
sc.close();
```

```
String s = "Name: Jim Sex: M Hobby: Music Age: 28";
Scanner sc = new Scanner(s);
String[] fields = {"Name:", "Hobby:", "Age:"};
for(String f: fields) {
    sc.findInLine(f);
    if (sc.hasNext())
        System.out.println(f + " " + sc.next());
    else {
        System.out.println("Error!");
        break;
    }
}
```

See also:
Scanner sc = new Scanner(input).
useDelimiter(";");
// e.g. for parsing CSV

Variable length argument lists

A feature borrowed from C/C++, same notation (ellipsis).

Argument list: **type ... identifier**. The type can even be an array.

Each method can only have a single type as a variable argument and it must come last in the parameter list.

```
int sum(int ... tab_int)
{
    int s = 0;
    for(int i=0; i<tab_int.length; i++)
    // or: for(int i: tab_int)
    {
        s += tab_int[i];
    }
    return s;
}

public static void main(String[] args)
{
    TestVarArg t = new TestVarArg();
    System.out.println(t.sum(1,3,5,2,2,10));
}
```

Previous example,
can we have `int sum(int[])` signature
and still use e.g. `sum(5, 10, -30)`?

Answer: no, the compiler reports an error.
You can use `sum(new int[] {5, 10, -30})` instead.

Actually, the ellipsis is just syntactic sugar for having the compiler create and initialize an array of same-typed values and pass that array's reference to a method.

When the ellipsis can be used?
E.g., for computing the average of a list (of e.g. floats),
concatenating a list of strings into a single string, finding the
max/min values in lists of floats.

Varargs, if at least one argument required

(J.Bloch's presentation, S48–49,

<http://files.meetup.com/1381525/still-effective.ppt.pdf>)

```
// The WRONG way to require one or more arguments!
```

```
static int min(int... args) {  
    if (args.length == 0)  
        throw new IllegalArgumentException(  
            "Too few arguments");  
    int min = args[0];  
    for (int i = 1; i < args.length; i++)  
        if (args[i] < min)  
            min = args[i];  
    return min;  
}
```

No args – fails at
runtime.

```
static int min(int firstArg, int... remainingArgs) {  
    int min = firstArg;  
    for (int arg : remainingArgs)  
        if (arg < min)  
            min = arg;  
    return min;  
}
```

OK!
No args –
compile error.

Classes

*Object-oriented programming
is an exceptionally bad idea
which could only have
originated in California.
/ Edsger Dijkstra /*

Class is an object type.

```
class Book {  
    ... // data & methods  
    Book(String author, final String title) // constructor  
    {  
        Objects.requireNonNull(author,  
            "author cannot be null");  
        this.author = author;  
        ...  
    }  
} // no ; as opposed to C++
```

Default (=no-arg) constructor

```
class X {  
    private int i, j;  
    void setState(int a, int b) { this.i = a; j = b; }  
    void showState() { ... }  
}  
...  
X x = new X(); // OK (default constructor launched)
```

But if we had

```
class X {  
    ...  
    X(int a, int b) { ... } // "standard" constructor  
}  
then  
X x = new X(); // compile error
```

this in a constructor

```
public class Date {  
    private int year, month, day;  
  
    public Date(int d, int m, int y) // one constructor  
    {  
        day = d; month = m; year = y;  
    }  
  
    public Date(int y) { this(1, 1, y); } // another  
  
    public Date( ) // yet another constructor  
    {  
        this(2018);  
    }  
}
```

Based on an example from

http://www.cs.fiu.edu/~weiss/cop3338_f00/lectures/Objects/objects.pdf

this specifics

```
// From 'Thinking in Java, 2nd ed.' by Bruce Eckel
// Calling constructors with "this."

public class Flower {
    int petalCount = 0;
    String s = new String("null");

    Flower(int petals) { ... }

    Flower(String ss) { ... }

    Flower(String s, int petals) {
        this(petals);
        //!    this(s); // can't call two!
        this.s = s; // Another use of "this"
        System.out.println("String & int args");
    }
    void print() {
        //!    this(11); // Not inside non-constructor!
        System.out.println(
            "petalCount = " + petalCount + " s = " + s);
    }
    ...
}
```

Hiding identifiers

Class fields can be hidden:

```
class A {  
    int a;  
    void method() {  
        int a = 0;    // local var.  
        a = a + 10;    // local var.  
        this.a++;      // field of the object  
    } ...  
}
```

You can't do it in blocks:

```
int a;  
{  
    int a;    // COMPILE ERROR  
}
```

This is wrong too:

```
int i = 0;  
for (int i = 0; i < 10; i++) { ... }
```

```
class FailedHiding {  
    public static void main(String[] args) {  
        String[] args = new String[3]; // compile error  
    }  
}
```

Method overloading

- `int fun(int x)`
- `int fun(float y)`
- `int fun(int a, float b)`

All they can appear in one class.

But not

- `int fun(int x)`
and
- `boolean fun(int y)`

Static data, static methods

```
static int n;
```

```
// all class objects “see” the same value of n
```

Static method – belongs to the class.

It can access only static fields/methods of the class.
this not accessible.

```
System.out.println(Math.sqrt(2));
```

Note however that a non-static method
can invoke a static method or access a static field.

```
Can we write like that?  
Math m = new Math();  
System.out.println(m.sqrt(2));  
?
```

No. The Math constructor
is private.

(Generally) avoid static fields; careful with static methods

Static variables represent global state.

If I create a new object, I can reason about its state within tests. If I use code which is using static variables, it could be in any state – and anything could be modifying it.

A decoupled class is generally easier to reuse.

In short: **hard to test** e.g. methods accessing static fields inside.
Hard to reuse such code.

Yet: static methods are generally fine, as long as they do not access any static data, but operate only on their arguments.

<http://stackoverflow.com/questions/7026507/why-are-static-variables-considered-evil>

<http://stackoverflow.com/questions/13453978/static-methods-and-static-variables-bad-design-practice>

Packages

Packages are class libraries (create: `package its.name;`)

Importing a package, example: `import java.util.*;`

Collisions possible:

You've created class `Vector` in package `school.my_cool_data_structures`. But you've also imported `java.util`.

Now, the line `Vector v = new Vector();` makes a collision.

Solution: use full path.

```
java.util.Vector v = new java.util.Vector();
```

or

```
school.my_cool_data_structures.Vector v =  
new school.my_cool_data_structures.Vector()
```

You can import a package more than once (no compiler complaint), you can write the `import...` line in any place in your code.

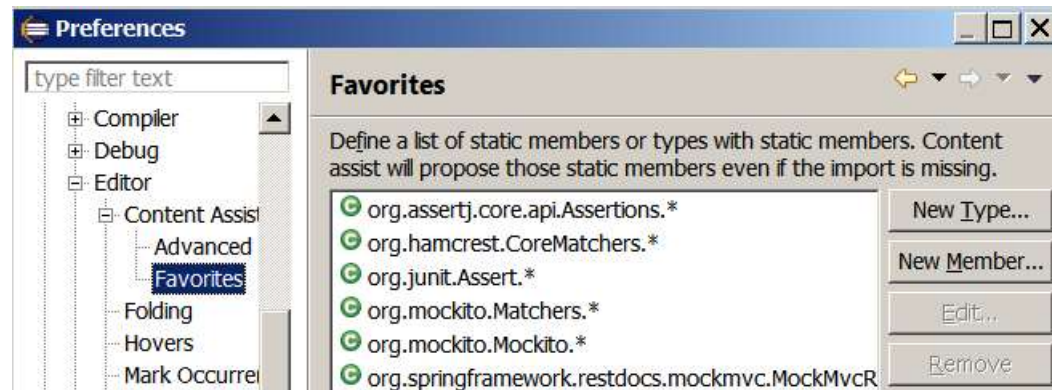
Static imports

[<http://www.agiledeveloper.com/articles/Java5FeaturesPartII.pdf>]

import statement gives a hint to the compiler as to which class you are referring to in your code. E.g. `javax.swing.JButton` -> `JButton`.

However, not only a fully qualified class name can be **omitted**, but also the **class name in front of static methods**. E.g. can use `abs()` instead of `Math.abs()`.

`import static java.lang.Math.abs;` // or even **`...Math.*;`**
`import static org.junit.jupiter.api.Assertions.assertEquals;`



Another example:

```
if (d.get(DAY_OF_WEEK) == MONDAY)
```

more readable than

```
if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
```

Access modifiers (for classes, methods, fields)

[members.tripod.com/~psdin/comlang/basicJava.PDF]

Member Accessibility	private	package	protected	public
From a subclass in the same package	yes no!	yes	yes	yes
From a non-subclass in the same package	no	yes	yes	yes
From a subclass in another package	no	no	yes	yes
From a non-subclass in another package	no	no	no	yes

If there is no direct access to a field (property),
you may provide **accessor / mutator** (aka get / set) methods.

...If really it is needed to read / write a field
from outside the class (think it over hard!).

Why private (or protected?)

To avoid accidental “spoiling” of some data.

To provide a proper class interface

(encapsulation rule: **hide unimportant details**).

Too many getter / setter calls is NOT a good OO design

Note that getters and setters provide external access to implementation details. :-/

Example: there are 1,000 calls to an *int* `getX()` method in your program, and now you want to change X's (which is private) type to long.
...1000 compile errors!

So, how to avoid getters / setters?

If you go through a design process, as opposed to just coding, you'll find hardly any accessors in your program.

The lack of getter/setter methods doesn't mean that some data doesn't flow through the system.

*Nonetheless, it's best to minimize data movement as much as possible. (...) maintainability is inversely proportionate to the amount of data that moves between objects.
(...) you can actually eliminate most of this data movement.*

By designing carefully and focusing on what you must do rather than how you'll do it, you eliminate the vast majority of getter/setter methods in your program.

*Don't ask for the information you need to do the work;
ask the object that has the information to do the work for you.*

Potential security hole

```
public static final Thing[] VALUES = { ... };  
public → private?
```

If there exists a getter to VALUES,
a client can change VALUES[0], or VALUES[1] etc.

Much better:

```
private static final Thing[] PRIVATE_VALUES = { ... };  
public static final List<Thing> VALUES =  
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Default package

```
class X { // file X.java
    int x;
    void doSomething() { ... }
}
```

```
class Y { // file Y.java in the same directory
    void someMethod() {
        X x = new X(); x.doSomething();
        // does it work? Can Y see components of X ?
    }
}
```

X's components are NOT public.
So the question is:
are Y and X in the same package?
We didn't write *package* ...

Yes!

Because they are in the same directory
and have no explicit package name.

Field initialization

Fields are initialized to ZERO (0, *false* for the boolean type, *null* for references (i.e., object types)).

BEWARE: **local variables are not initialized** (have random values)! Like in C/C++, by the way. But trying to read an uninitialized variable yields a compile error.

You **can** also **explicitly initialize a field**:

```
class Pair {  
    static String text = "Pairs";  
    static int x = -1;  
    int a = x+1;  
    int i = f();  
    int j = g(i);  
    . . . . .
```

But not:

```
.....  
int j = g(i); int i = f();  
// order matters!
```


Inheritance

All non-private methods from the parent class are present in the child class.

You can inherit from a single class (as opposed to C++).
All Java classes are derived from the class Object.

Constructors in the child class should call explicitly the constructor of the parent class.

Keyword **super**.

```
public class Vector3D extends Vector2D {  
    ....  
    public Vector3D(int x, int y, int z) {  
        super(x, y); // calls the parent constructor  
        this.z = z;  // sets a remaining field  
    }  
}
```

If `super(args)` is invoked, it must be the constructor's first instruction!
Otherwise: `super()`; implicitly called.

Terminology:
superclass,
base class,
parent class

subclass,
derived class,
extended class,
child class

Keyword *final*

`final class X { ... }` → means that X **cannot be extended**
(for example, String class is final)

`final int someMethod(...);` → means that `someMethod(...)`
cannot be overridden

Terminology: don't confuse
(method) overloading and
overriding!

Overloading a function: same name
but a different list of params.

Overriding a function: same signature but in some
descendant class.

final, cont'd

```
final int[] a = { 1, 2, 3, 4 };  
a[0]++; // OK ???
```

Yes. The reference is final (cannot be changed), not the pointed object. Therefore, of course, `a = b;` would be erroneous.

Blank finals

```
private final int i = 0; // Initialized final  
private final int j;     // Blank final
```

Value must be assigned to a blank final in every constructor. Why blank finals? Since they provide flexibility: a **final field** inside a class **can now be different for each object**.

Constant stuff

No constant methods. I.e. everything may change the object.

Method name convention:

getMember: an accessor (getter)

setMember: a mutator (setter)

Constant instance fields: with *final* keyword.

A constructor (of a given class or some ancestor class in the hierarchy) must set them to a value, and then it is fixed.

Final arguments:

e.g. void fun(**final** int j) { j++; /* ERROR! */ }

// j is read-only

Shadowed variables

A variable in a subclass **shadows** an inherited variable if both have the same name.

```
class A { int x; ... }  
class B extends A { int x; ... }
```

How to distinguish them?

Easily. In the child class: use **x** or **this.x** for x from this class,
and **super.x** or **((ParentClass)this).x** to for the shadowed x.
Outside both classes: cast an object to the appropriate type.

Polymorphism, virtual methods

A method that can be overridden = virtual method
(no extra keyword in Java).

Polymorphism: one name, many forms.

```
class Car extends Vehicle { ... }
```

```
...
```

```
Car c = new Car(...);
```

```
Vehicle v = c; // !
```

```
v.start(); // start() from Vehicle overridden in Car
```

Can the **compiler** (javac) know it should run start()
from the class Car rather than Vehicle?

Perhaps in this example, yes. 😊

But not in general.

Polymorphism, cont'd

(based on [Barteczko'04], pp. 304–305)

```
public static void main(String[] args) {  
    Car c = new Car(...);  
    Motorcycle m = new Motorcycle(...);  
    Vehicle v = args[0].equals("Jawa") ? m : c;  
    v.start(); // can the compiler know what is v ?!?!  
}
```

(Run-time) polymorphism implies **late binding**.

(Almost) all methods in Java are virtual

Methods that cannot be redefined:

- static ones;
- methods with *final* keyword (of course...);
- private methods.

Performance problems? Probably not
(with HotSpot JVM).

For virtual methods, HotSpot keeps track of whether the method has actually been overridden, and is able to perform optimizations such as inlining on the assumption that a method hasn't been overridden – until it loads a class which overrides the method, at which point it can undo (or partially undo) those optimizations.

/ Jon Skeet, Nov. 2010 /

Virtual methods, return type

The return type is not part of the signature.
However, when you override a method,
you need to keep the return type **compatible**
(not necessarily identical!).

```
public Person me() { ... }
```

Then the Student subclass can override this method as

```
public Student me() { ... }
```

We say that the two `me()` methods have
covariant return types.

Inheritance and composition

Implementation inheritance provides “is-a” relationship:
a Student is a Person, a Car is a Vehicle, etc.

Composition concerns making classes from other classes, e.g. a Car may include an Engine (field), 4 Wheels (fields), etc.

The relationship between them is: “has-a”.

Implementation inheritance is dangerous:
it **breaks encapsulation!**
(But what does it mean..?)

Inheritance breaks encapsulation

Assume the subclass relies on implementation details of the superclass.

Now, if the details change in a new version of the superclass, the subclass might (surprisingly?) break.

Note we may not have access to source code of the superclass...

Is this a good design?

```
public class Stack<T> extends ArrayList<T>{
    private int stackPointer = 0;

    public void push(T item) {
        add(stackPointer++, item);
    }

    public T pop() {
        return remove(--stackPointer);
    }

    public void pushMany(T[] items) {
        for(int i = 0; i < items.length; i++)
            push(items[i]);
    }
}
```

```
Stack<Integer> s = new Stack<>();
s.push(1); s.push(2); s.clear();
// see the problem?
```

Better solution:
use an `ArrayList<T>`
as a field in `Stack<T>`.

I.e., use composition,
not inheritance.

Abstract classes

Abstract class – cannot have instances (objects).
Methods can also be abstract (no definition).
Abstract methods – only in abstract classes.
But not all methods in an abstract class
have to be abstract.

```
public abstract class Shape { ... }
```

What for? To have a common design for inherited classes.

Can an abstract class have a constructor?

Abstract class constructor, motivation

Define a constructor in an abstract class in any of the following cases:

- you want to perform some initialization (to fields of the abstract class) before the instantiation of a subclass actually takes place,
- you have defined final fields in the abstract class but you did not initialize them in the declaration itself; in this case, you **MUST** have a constructor to initialize these fields.

You should define all your abstract class constructors **protected** (making them public is pointless).

Non-instantiable class [*Effective Java*, Item 4]

If you want your class X be non-instantiable, provide its **private constructor**. It may be empty.

Or even better:

```
private X() { throw new AssertionError(); }
```

Typical application: a utility class (e.g., `java.lang.Math`).

Using an abstract class instead is not good:

- the class can be subclassed and the subclass instantiated,
- it misleads the user into thinking the class was designed for inheritance.

java.net.InetAddress has no public constructors...

<code>byte[]</code>	<code>getAddress()</code> Returns the raw IP address of this <code>InetAddress</code> object.
<code>static InetAddress[]</code>	<code>getAllByName(String host)</code> Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.
<code>static InetAddress</code>	<code>getByAddress(byte[] addr)</code> Returns an <code>InetAddress</code> object given the raw IP address .
<code>static InetAddress</code>	<code>getByAddress(String host, byte[] addr)</code> Creates an <code>InetAddress</code> based on the provided host name and IP address.
<code>static InetAddress</code>	<code>getByName(String host)</code> Determines the IP address of a host, given the host's name.
<code>String</code>	<code>getCanonicalHostName()</code> Gets the fully qualified domain name for this IP address.
<code>String</code>	<code>getHostAddress()</code> Returns the IP address string in textual presentation.

...but its instances can be created, e.g., with
`InetAddress addr = InetAddress.getByName("127.0.0.1");`

Sealed classes in Java?

Some languages (e.g., Scala, Kotlin) allow to maintain a restricted class hierarchy, when a value (instance) can have one of the types from a limited set and this set cannot be extended.

E.g., assume a class Shape, extended by Square, Triangle and Circle; no other class can be added to extend Shape.

Shape in this example is called a **sealed** class. (Beware: sealed class in C# means smth else, it is equivalent to a final class in Java.)

Java doesn't have a language support for sealed classes. Yet, they can be obtained in a tricky way.

Sealed classes in Java, cont'd

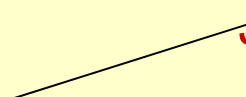
```
public abstract class A {  
    protected A() {  
        if (!(this instanceof B) && !(this instanceof C))  
            throw new UnsupportedOperationException("The class " +  
                this.getClass().getSimpleName() +  
                " cannot extend A, which is effectively a sealed class.");  
    }  
}  
  
// public class B extends A { ... }  
// public class C extends A { ... }
```

What is derived from the Object class

The class **Object** defines default versions of the following methods:

- clone()
- equals(Object obj)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()
- wait(long timeout)
- wait(long timeout, int nanos)

deprecated in
Java 9!



Every class inherits them, and can override (the non-final ones, i.e. except for notify, notifyAll, wait – all three, getClass).

equals

[based on J. Bloch, *Effective Java*, 2008, Chap. 3]

Overriding *equals* – when NOT to:

- class instances are inherently unique (e.g. Thread),
- testing for equality makes little sense (e.g. Random),
- a superclass overrides *equals* and its implementation is satisfactory (e.g. most Set implementations inherit *equals* from AbstractSet),
- the class is non-public and you are sure *equals* will never be invoked;

but you should override it then! Like this:

```
public boolean equals(Object o)
{ throw new UnsupportedOperationException(); }
```

Overriding equals, the contract

[based on J. Bloch, *Effective Java*, 2008, Chap. 3]

equals should be

- **reflexive**, i.e. `x.equals(x)`,
- **symmetric**, i.e. `x.equals(y) ↔ y.equals(x)`,
- **transitive**, i.e. `x.equals(y) && y.equals(z) → x.equals(z)`,
- **consistent**: multiple *equals* invocations on the same `x`, `y` should return the same, provided no info used in *equals* comparisons on the object is modified,
- `x != null → x.equals(null) == false`

Overriding equals, why symmetry?

[based on J. Bloch, *Effective Java*, 2008, Chap. 3]

Let's have a class for strings with case-insensitive equality test.

String is final, but we can use composition instead of inheritance:

```
public final class CaseInsensitiveString
{
    private String s;
    .... // some new cute methods, working on s
}
```

Overriding equals, why symmetry? (cont'd)

[based on J. Bloch, *Effective Java*, 2008, Chap. 3]

```
public CaseInsensitiveString(String s) {  
    if (s == null)  
        throw new NullPointerException();  
    this.s = s;  
}  
  
// broken - violates symmetry!  
public boolean equals(Object o) {  
    if (o instanceof CaseInsensitiveString)  
        return s.equalsIgnoreCase(  
            ((CaseInsensitiveString)o).s);  
    if (o instanceof String) // works in one way only!  
        return s.equalsIgnoreCase((String)o);  
    return false;  
}  
...  
}
```

Overriding equals, why symmetry? (cont'd)

[based on J. Bloch, *Effective Java*, 2008, Chap. 3]

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");  
String s = "polish";
```

cis.equals(s) returns true

s.equals(cis) returns false → no symmetry

OK, no symmetry, but so what?

Now let's have a list collection...

```
List<CaseInsensitiveString> list = new ArrayList<>();
```

...and add cis to it:

```
list.add(cis);
```

What does list.contains(s) return? NOT DEFINED!

Are the arrays equal?

```
int[] arr1 = {1, 3};  
int[] arr2 = {1, 3};  
System.out.println(arr1 == arr2); // ?  
System.out.println(arr1.equals(arr2)); // ?
```

The correct way (comparing the content):

```
import java.util.Arrays;  
System.out.println(Arrays.equals(arr1, arr2));
```

toString

[based on J. Bloch, *Effective Java*, 2008, Item 10]

By default (toString as defined in Object) the returned msg is: class name, @, unsigned hex representation of the hash code, e.g. PhoneNumber@163b91.

The toString method is automatically invoked when an object is passed to println, printf, the string concatenation operator (+), or assert, or printed by a debugger.

Example:

```
System.out.println("Failed to connect: " + phoneNumber);
```

Advice: provide programmatic access (e.g., implement accessors) to all of the information contained in the value returned by toString.

java.util.Objects: hashCode

```
public class Bar {
    private Foo foo;
    private Bar parent;

    @Override
    public int hashCode() { // old java
        int result = 17;

        result = 31 * result + (foo == null ? 0 : foo.hashCode());
        result = 31 * result + (parent == null ? 0 : parent.hashCode());

        return result;
    }
}

public class Bar {
    private Foo foo;
    private Bar parent;

    @Override
    public int hashCode() { // with objects class
        return Objects.hash(foo, parent);
    }
}
```

java.util.Objects: equals

```
public class Bar {
    private Foo foo;
    private Bar parent;

    @Override
    public boolean equals(Object obj){ // old Java
        if (obj == this)
            return true;

        if (obj instanceof Bar) {
            Bar other = (Bar) obj;

            if (foo != other.foo)
                if (foo == null || !foo.equals(other.foo))
                    return false;

            if (parent != other.parent)
                if (parent == null || !parent.equals(other.parent))
                    return false;

            return true;
        }
        return false;
    }
}
```

If you override *equals*,
you should also
override *hashCode*.

```
public class Bar {
    private Foo foo;
    private Bar parent;

    @Override
    public boolean equals(Object obj){ // with Objects class
        if (obj == this)
            return true;

        if (obj instanceof Bar) {
            Bar other = (Bar) obj;
            return Objects.equals(foo, other.foo) &&
                Objects.equals(parent, other.parent);
        }

        return false;
    }
}
```

<http://www.baptiste-wicht.com/2010/04/java-7-the-new-java-util-objects-class/>

WeakReference

Consider a map (a collection of pairs (key, value)).

What happens with a value whose key is no longer used anywhere in your program? If the last reference to a key has gone away, there is **no longer any way to refer to the value** object so it **should be removed** by the garbage collector.

But how?? Yes, we may have a **memory leak** here!

Note that the garbage collector traces live objects. As long as the map object is live, all entries in it are live and won't be reclaimed – and neither will be the values that are referenced by the entries.

If a **weak reference** is created, and then elsewhere in the code `get()` is used to get the actual object, the weak reference isn't strong enough to prevent garbage collection, so it may happen (if there are no strong references to the object) that `get()` suddenly starts returning null.

```
Reference<String> refString = new WeakReference<String>(new String("456"));
```

WeakReference, example

(<http://na-jawie.blogspot.com/2009/08/sabe-referencje-z-czym-to-sie-je.html>)

```
import java.lang.ref.Reference;
import java.lang.ref.WeakReference;
import java.util.ArrayList;

public class WeakReferenceTest {
    int[] tab = new int[1024];

    public static void main(String[] args) {
        String string = new String("123");
        Reference<String> refString = new WeakReference<String>(new String("456"));
        System.out.println(string);
        System.out.println(refString.get());

        java.util.List<WeakReferenceTest> list = new ArrayList<WeakReferenceTest>();

        for (int i = 0; i < 1000000; i++) {
            WeakReferenceTest test = new WeakReferenceTest();
            list.add(test);
            if (refString.get() == null) {
                System.out.println(string);
                System.out.println(i + " " + refString.get());
                break;
            }
        }
    }
}
```


WeakHashMap

```
Map hashMap = new HashMap();
Map weakHashMap = new WeakHashMap();

String keyHashMap = new String("keyHashMap");
String keyWeakHashMap = new String("keyWeakHashMap");

hashMap.put(keyHashMap, "helloHash");
weakHashMap.put(keyWeakHashMap, "helloWeakHash");
System.out.println("Before: HM value: " + hashMap.get("keyHashMap") +
    " and WHM value: " + weakHashMap.get("keyWeakHashMap"));

keyHashMap = null;
keyWeakHashMap = null;

System.gc();

System.out.println("After: HM value: " + hashMap.get("keyHashMap") +
    " and WHM value: " + weakHashMap.get("keyWeakHashMap"));
```

Before: hash map value: helloHash and weak hash map value: helloWeakHash

After: hash map value: helloHash and weak hash map value: null

Arrays are objects!

But of what class..?

This is implicit (“hidden” to a programmer).

E.g. the name of a 1-dim array of int's is [I,

the name of a 2-dim array of int's is [[I,

the name of a 1-dim array of double's is [D etc.

javadoc (1)

Writing documentation to large software projects:
both very tedious and very important.

javadoc is a simple tool to facilitate it. Just comment each (public) function in the classes and run javadoc – the whole documentation will be generated (as a set of html pages).

Oracle documentation for the standard Java API built with javadoc.

The documentation for each item must be placed in a comment that *precedes* the item. (This is how javadoc knows which item the comment is for.)

By default, javadoc produces html docs for public classes and public/protected members.

To produce docs for non-public classes:

`javadoc -private Hello.java`

javadoc (2)

Javadoc documentation in `/** ... */`.

Special notation allowed:

`@return`, `@param`, `@author`, `@since` ...

```
/**
 * Return the real number represented by the string s,
 * or return Double.NaN if s does not represent a legal
 * real number.
 *
 * @param s String to interpret as real number.
 * @return the real number represented by s.
 * @since version 1.1
 */
```

From
<http://www.faqs.org/docs/javap/advanced.html>

```
public static double stringToReal(String s) {
    try { return Double.parseDouble(s); }
    catch (NumberFormatException e) { return Double.NaN; }
}
```

How to create an own package

Very simply.

Just write the code of your class (in a file with the same name as the class name)

and start it with line e.g.

```
package myutils.text;
```

Your class will be added to the package myutils.text.

Of course, there can be many classes in a package. Remember: earlier in the file than the *package...* line may be only comments.

Java naming conventions

- If a name is composed of several words, then each word (except possibly the first one) begins with an uppercase letter. Examples: `setLayout`, `addLayoutComponent`.
- Names of variables, fields, and methods begin with a lowercase letter. Examples: `vehicle`, `myVehicle`.
- Names of classes and interfaces begin with an uppercase letter. Examples: `Cube`, `ColorCube`.
- Named constants (such as `final static` fields) are written entirely in uppercase, and the parts of composite names are separated by underscores (`_`). Examples: `CENTER`, `MAX_VALUE`.
- Package names are sequences of dot-separated lowercase names. Example: `java.awt.event`. For uniqueness, they are often prefixed with reverse domain names, as in `com.sun.xml.util`.

Before static imports [<http://norvig.com/java-iaq.html>]

Imagine you have a class (or several methods) where math trig functions are very frequently called. Can you save keystrokes compared to `Math.sin(...)` etc. all the time? What if old Java (<1.5) is used?

Solutions:

1. Add the sin/cos etc. to your class:

```
public static double sin(double x) { return Math.sin(x); }
```

...

2. Use a trick:

```
// Can't instantiate Math, so it must be null.
```

```
    Math m = null;
```

```
    m.sin(x);
```