# CS 240
# JAVA TUTORIAL

Vlado Keselj

University of Waterloo

October 1, 1998

## References:

- books (the course textbook etc.)
- http://java.sun.com/ site:
  - starter.html
    — Java starter
  - docs/books/tutorial/index.html
    — Java tutorial
  - docs/white/langenv/
    — Java Reference (white paper)
  - products/jdk/1.1/docs/api/packages.html
    — Java API Specification 1.1

## $\boxed{\textbf{Goal:}}$

- CS240 assignment oriented
- environment:
  - $\boxed{\text{application}}$
    *run as a "normal" program in Unix environment*
  - applet
    *embedded in a home page and run in a client's browser*
  - more specialized
    *e.g., smart cards, appliances...*

## Assumed prior knowledge or skills:

- assumed familiarity with OOP
- knowledge of C: useful
- knowledge of C++: even better

# Hello world!

```
> emacs hello.java
```

*(or vi, or some other editor)*

emacs

```
public class hello {
  public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

hello.java

```
> javac hello.java
```

*(hello.class is created)*

```
> java hello
Hello world!
> _
```

## Hello-world points

- application: set of files of the form ∗.java
- a class **hello**:
  - is defined in file **hello.java**
  - declaration form:
    **public class hello { . . . }**
  - byte code in file **hello.class**
    (`javac *.java`)
- execution starts by calling a method **main** in a class; method's signature:
  **public static void main(String[]** *args***)**
  *args* — command line arguments
- print to standard output:
  `System.out.print(...)`
  `System.out.println(...)`
- strings: `"Hello "` ⇔ `Hello `
  `"Hi, \"world\""` ⇔ `Hi, "world"`
  `"Hi\nworld"` ⇔
  ```
  Hi
  world
  ```

# Comments:

```
// comment 1 . . .

/* comment 2 . . .
    . . . */

/**
   comment 3 . . .
     @author . . .
   or
     @param . . .
     @return . . .
 */
```

## Primitive types

**boolean:** `true` or `false`

**char:** 2-byte Unicode character, e.g. 'A'

**byte:** 1-byte signed integer (-128...127)

**short:** 2-byte signed integer (-32768...32767)

**int:** 4-byte signed integer (approx. $\pm 2 * 10^9$)

**long:** 8-byte signed integer (approx. $\pm 10^{19}$)

**float:** 4-byte floating-point number

**double:** 8-byte floating-point number

## Declaring a variable:

```
int i;
boolean b = false;
double d = e * 0.45;

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

- don't have to be declared at the beginning of a block
- run-time initialization value allowed

## Operators:

- assignment: `a = a + 1;`
- arithmetic: `+`, `-`, `*`, `/`, `%` (modulo)
- cast: `d = ((double) i ) / j;`
- comparison and equality: `<`, `<=`, `>`, `>=`, `==`, `!=`
- boolean: `&&` (and), `||` (or), `!` (not)
- bitwise: `&`, `|`, `~` (neg), `^` (xor), `<<` (left shift), `>>` (right shift, sign ext.), `>>>` (right shift, no sign ext.)
- conditional: `max = ( (a>b) ? a : b )`
- increment and decrement: `++i`, `i++`, `--i`, `i--`
- assign.-op. comb.: `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `>>>=`, `&=`, `^=`, `|=`
- `[]`, `.`, `new`, `instanceof`

# Control structures

```
if (i==10) { i = 0; j = j + 1;}
else if (i == 9) i = i + 1;
else i += 2;

switch (state) {
case 0: state = 3; break;
case 1: state = 0; break;
default: state = 2;
}

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

while (noEnd) {...}

do {...} while (!over);
```

- `break`, `continue`, no `goto`

## Reference Data Types

- arrays and objects

## Arrays

```
public class arrays {
  public static final String[] days =
  {"Mon","Tue","Wed","Thu","Fri",
                              "Sat","Sun"};
  public static void main(String[] args) {
    for (int i=0; i<days.length; ++i)
      System.out.print(days[i] + " ");
    System.out.println();
    String[] tmp = days;
    tmp[0] = "XYZ";
    for (int i=0; i<days.length; ++i)
      System.out.print(days[i] + " ");
    System.out.println();
  }
}
```

. . . output:

```
Mon Tue Wed Thu Fri Sat Sun
XYZ Tue Wed Thu Fri Sat Sun
```

- declaring arrays:

```
int[] a; double[][][] triple;
```

- initialization, assignment:

```
int[][] a = null;
int[] b = new int[3];
String[] s = {"jan","feb","mar"};
double[][][] d = new double[12][][];
int[][] l = {{0, 1}, {0, 1, 2}, null};
```

- bounds: `0` and a`.length-1`,
  i.e., from a`[0]` to a`[`a`.length-1]`

## Strings

- strings are objects
- string concatenation:

  ```
  "Hello " + "world"
  "o = " + o + "i + j = " + (i + j)
  ```

  (*o is an object and i and j are integers*)
- string equality: `s1.equals(s2)`
  (`s1==s2` is **not** usually what you want)
- string comparison: `s1.compareTo(s2)`
  $< 0$ if `s1<s2`, $= 0$ if `s1=s2`, $> 0$ if `s1>s2`
- a string cannot be changed, an instance of StringBuffer should be used instead

## Objects (Linked List example)

```
public class ListI {
  public int i;
  public ListI next = null;

  public ListI(int i) {
    this.i = i;
  }
  public ListI(int i, ListI next) {
    this.i = i;
    this.next = next;
  }
  public void insert(int i) {
    next = new ListI(i, next);
  }
}
```

# Objects

- construction: `ListI head = new ListI();`
- field access, method invocation:
  `head.i = 5; head.insert(4);`
- garbage collector: no worries about deallocation
- all object variables are references, i.e. pointers in a sense (Java does not have pointers)
  In

  ```
  MyObject a = b;
  a.changeYourself();
  ```

  Is the object represented by `b` changed?
  — Yes.
- method arguments and return values are passed by value (which is a pointer value for objects and arrays)
- `a = b.clone();`
  creates a separated copy of an object

## Classes, static fields, static methods

- classes form a tree hierarchy with the class `Object` being the root
- no multiple inheritance (interfaces can be used, instead)
- static field values are unique for a class, and not related to any specific object
  e.g., `System.out`, `System.in`, `Math.PI`, `Math.E`
- static methods are associated with a class, but not with any specific object—instance of that class; e.g.,

```
x = Math.abs(y);
x5 = Math.pow(x,5);
f = Math.log(x);
i = Math.floor(d);
max = Math.max(a,b);
k = Math.sin(alpha);
beta = Math.atan(coeff);
```

## Standard Java packages

A package provides a set of useful classes.

- java.applet
- java.awt ...
- java.beans
- java.io — file system I/O, and more
- java.lang — always implicitly used
- java.lang.reflect
- java.math
- java.net
- java.rmi ...
- java.security ...
- java.sql
- java.text
- java.util — contains useful data structures, e.g. Vector, Hashtable...
- java.util.zip

## How to use the standard packages

- read API documentation on the Web (or a reference book)
- java.lang can be used directly, e.g.

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

- use the full name of a class, e.g.

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

- use the directive `import`, e.g.

```
import java.util.*;
public class MyClass {
...
  Vector v = new Vector();
...
}
```

# Vector

- basically an object array, but more flexible

```
Vector v = new Vector();

v.addElement("first");
v.addElement("Second");

v.setElementAt("First", 0);

String s = (String) (v.elementAt(1));

v.removeElementAt(0);

int i = v.size();

for (Enumeration e = v.elements();
     e.hasMoreElements() ;)
  System.out.println(e.nextElement());
```

```
/* Example: sketching a graph ADT */
import java.util.*;
public class Graph {
  private int numOfVertices = 0;
  private int numOfEdges = 0;
  private Vector[] neighbors = null;

  public Graph(int n) {
    numOfVertices = n;
    neighbors = new Vector[n];
    for (int i=0; i<n; ++i)
      neighbors[i] = new Vector();
  }
  public boolean areAdjacent(int v1,
                             int v2) {
    if (v1==v2) return false;
    ...
  }
  public boolean addEdge(int v1,int v2) {
  ...
  neighbors[v1].addElement(new
                         Integer(v2));
```

# Primitive type $\Leftrightarrow$ String conversion

- $\Rightarrow$

```
int i = 5;
double d = 3.14;
boolean b = true;
MyObject o = new MyObject();
String si = "" + i;
si = "" + d;
si = "" + b;
si = "" + o;
```

  (`toString` method is used for objects)

- $\Leftarrow$

```
int i = Integer.valueOf(s).intValue();
double d =
 Double.valueOf(args[0]).doubleValue();
```

- primitive data types are not objects, but they can be easily transformed into objects of the standard associated types: Integer, Boolean, Double, Character, ...

## Getting standard input

```
import java.io.*;
public class input {
  public static void main(String[] args)
  throws Exception {
    int i = 0; boolean eof=false;
    StreamTokenizer in =
      new StreamTokenizer
      (new BufferedReader
      (new InputStreamReader(System.in)));
    while (!eof) {
      in.nextToken();
      switch(in.ttype) {
      case StreamTokenizer.TT_NUMBER:
        i += (int)(in.nval); break;
      case StreamTokenizer.TT_EOF:
        eof=true;
    } }
    System.out.println("sum = " + i);
} }
```

## “Wrapping” classes

- any primitive type can be wrapped in an object of the associated “wrapping” class
- Integer, Boolean, Character, Double, ...
- wrapping:
  `o = new Integer(i);`
- unwrapping:
  `i = o.intValue();`
- Class Number is superclass of: Byte, Double, Float, Integer, Long, and Short.
- useful constants end methods: e.g.,
  `Integer.MIN_VALUE`,
  `Character.isLetter(c)`
- converting from a string: e.g.,
  `Integer.valueOf(s)`

## More about Class Hierarchy

- Object (`java.lang.Object`) is the root of the tree hierarchy
- hierarchy is pretty shallow
- Java is "self-describing" in many ways: Class, ClassLoader, Method, Field, Exception, Error…are classes
- classes Exception and Error are subclasses of Throwable
- Exception instances should be handled by programmer
  They do not have to be, but Java compiler forces programmer to think about them
- Error instances usually cause program termination, and you do not have to think about those problems (e.g., no more free memory)

## Class hierarchy: example

In file Vertex.java:

```
import java.util.*;

public class Vertex {
  private String name = null;
  private Vector[] neighbors = null;
  public Vertex() {
    System.out.println("vertex: "+
        "default constructor");
  }
  public Vertex(String name) {
    System.out.println("vertex: "+
        "constructor1");
    this.name = name;
  }
  public String hi() {
    return "Hi, I am vertex " +
           name + "!";
} }
```

In file ColoredVertex.java:

```
public class ColoredVertex
extends Vertex {
  String color = null;
  public ColoredVertex(String name) {
    System.out.println("colored vertex:"
        + " constructor1");
    this.name = name;
  }
}
```

After running `javac *.java`:

```
Variable name in class ColoredVertex
not accessible from class ColoredVertex.
```

In Vertex.java replace the row:

```
private String name = null;
```

...by row:

```
protected String name = null;
```

Compilation successful.

In file vertices.java:

```
public class vertices {
 public static void main(String[] args){
    Vertex v = new ColoredVertex("A");
    System.out.println(v.hi());
  }
}
```

Output (`java vertices`):

```
vertex: default constructor
colored vertex: constructor1
Hi, I am vertex A!
```

## Reserved word: `super`

In ColoredVertex.java replace the old constructor with:

```
public ColoredVertex(String name) {
  super(name);
  System.out.println("colored vertex:"
         + " constructor1");
}
```

Output after compiling and running:

```
vertex: constructor1
colored vertex: constructor1
Hi, I am vertex A!
```

## Method overriding

In file ColoredVertex.java add the following method:

```
public String hi() {
  return "Hi, I am colored vertex " +
         name + "!";
}
```

Output:

```
vertex: constructor1
colored vertex: constructor1
Hi, I am colored vertex A!
```

Modify vertices.java:

```
public class vertices {
  public static void main(String[] args) {
    Vertex v = new ColoredVertex("A");
    System.out.println(v.hi());
    System.out.println( ((Vertex)v).hi());
  }
}
```

Output

```
vertex: constructor1
colored vertex: constructor1
Hi, I am colored vertex A!
Hi, I am colored vertex A!
```

In file ColoredVertex.java replace the old method **hi** with:

```
public String hi() {
  return super.hi() + "\n" +
    "Hi, I am colored vertex "
    + name + "!";
  }
}
```

Output:

```
vertex: constructor1
colored vertex: constructor1
Hi, I am vertex A!
Hi, I am colored vertex A!
...
```

## **Method:** `toString()`

<u>In file Vertex.java</u> remove the method **`hi`** and add method:

```
public String toString() {
  return "vertex " + name;
}
```

<u>In file ColoredVertex.java</u> remove the method **`hi`** and add method:

```
  public String toString () {
    return "colored " + super.toString();
  }
```

Change file vertices.java to

```
public class vertices {
  public static void main(String[] args) {
    Vertex v = new ColoredVertex("A");
    System.out.println(v);
  }
}
```

Compile: `javac vertices.java`
Run: `java vertices`

```
vertex: constructor1
colored vertex: constructor1
colored vertex A
```

## Method overloading

Update vertices.java:

```
public class vertices {
  public static void main(String[] args) {
    Vertex v = new ColoredVertex("A");
    f(v); f(-1);
  }
  public static void f(Vertex v) {
    System.out.println("f(" + v + ")");
  }
  public static void f(int i) {
    System.out.println("f(" + i + ")");
  }
}
```

Output (`java vertices`):

```
vertex: constructor1
colored vertex: constructor1
f(colored vertex A)
f(-1)
```

vertices.java:

```
public class vertices {
  public static void main(String[] args) {
    Vertex v = new ColoredVertex("A");
    ColoredVertex v1 =
                 new ColoredVertex("B");
    f(v); f(v1);
  }
  public static void f(Vertex v) {
    System.out.println("f(" + v + ")");
  }
  public static void f(ColoredVertex v) {
    System.out.println("f(" + v + ") col");
  }
}
```

Output:

```
...
f(colored vertex A)
...
f(colored vertex B) col
```

But, if you use:

```
f( (ColoredVertex) v );
```

the same method is called in both cases:

```
...
f(colored vertex A) col
...
f(colored vertex B) col
```

If method `f(ColoredVertex)` is deleted, then

```
...
f(colored vertex A)
...
f(colored vertex B)
```

If method `f(Vertex)` is deleted, then explicit cast to ColoredVertex is required.

## Cloning

Update Vertex.java:

```
public class Vertex implements Cloneable {
...
  public void setName(String name) {
    this.name = name;
  }
  public Object clone()
  throws CloneNotSupportedException {
    return super.clone();
  }
```

Update ColoredVertex.java

```
public class ColoredVertex extends Vertex
implements Cloneable {
...
```

## File vertices.java

```
public class vertices {
  public static void main(String[] args)
  throws CloneNotSupportedException {
    Vertex v = new ColoredVertex("A");
    Vertex v1 = (Vertex)(v.clone());
    v1.setName("B");
    System.out.println(v +"\n" + v1);
  }
}
```

```
vertex: constructor1
colored vertex: constructor1
colored vertex A
colored vertex B
```

## Exceptions

- exceptions and errors
- exception generation
- exceptions form hierarchy:
  Throwable—Exception—IOException…
- passing or catching exception

Example (the simplest way to deal with exceptions):

```
...
  public static void main(String[] args)
  throws Exception {
    int i = 0; boolean eof=false;
    StreamTokenizer in =
      new StreamTokenizer
      (new BufferedReader
      (new InputStreamReader(System.in)));
```

## Catching an exception

```
try {
  /*... normal execution */
} catch (IOException e) {
  /*... handle exception e */
} catch (MyException e) {
  /*... handle exception e */
} finally {
  /*... guaranteed to be executed*/
}
```

## Throwing an exception

```
throw new MyException("something's wrong");
```

## Creating exception class

```
public class MyException
extends Exception {
...
}
```

## Class modifiers

- **abstract** classes cannot be instantiated
- **final** classes cannot be "subclasses" (extended); example:

```
public abstract class Graph {
```

## Field modifiers

- **static** = class variable
- **final** field cannot change value
- other: **transient**, **volatile**

## Method modifiers

- **abstract** methods do not have body; e.g.,

```
public abstract boolean
      areAdjacent(int v, int u);
```

- **static** = class method
- **final** method cannot be overridden
- other: **native**, **synchronized**

## Interfaces

- a kind of multiple inheritance
- includes constants and abstract methods
- can extend several interfaces
- similarily to a class, one interface goes into one file with the same base name

Example:

```
public interface Drawable {
  public void paint();
  public void repaint(int top, int left,
       int width, int height);
}

public class ColoredVertex
extends Vertex
implements Drawable, Enumeration {...
```

## Some other general Java issues. . .

- threads. . .
- object serialization. . .