JAVA for C++ Developers

Dr. Eliahu Khalastchi

2017

המסלול האקדמי
המכללה למינהל

Hello World

- The simple example:

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

- Conventions:
  - File name = class name, one class per file
  - Class's name starts with capital letter
  - Object's name start with lower case letter
  - Case sensitive

Hello World

- Will this work?

```
public class BadHelloWorld {
 public void print(){
  System.out.println("Hello World!");
 }
 pu  sta  void main(String[] args) {
  prin
 }
}
```

Hello World

- Static methods can invoke only static members

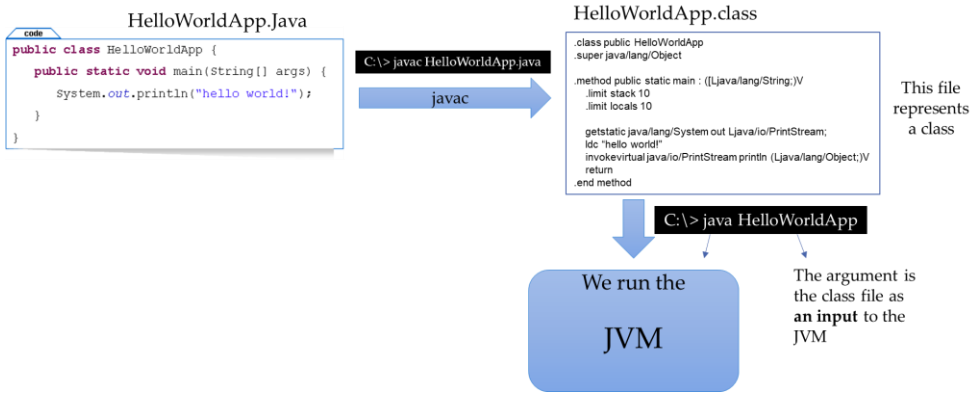
```
public class BadHelloWorld {

 public static void print(){

  System.out.println("Hello World!");

 }

 public static void main(String[] args) {

  print();

 }

}
```

Hello World

- The better, more object oriented example:

```
public class BetterHelloWorld {
 public void print(){
  System.out.println("Hello World!");
 }
 public static void main(String[] args) {
  BetterHelloWorld b=new BetterHelloWorld();
  b.print();
 }
}
```

How the main is able to create an object of the same class the main is in??



Data types

Type	Size	Min	Max	Wrapper	Defaut
boolean	-	-	-	Boolean	false
char	16-bit	0	$2^{16}-1$	Character	'\u0000'
byte	8-bit	-128	+127	Byte	(byte)0
short	16-bit	-2^{15}	$+2^{15}-1$	Short	(short)0
int	32-bit	-2^{31}	$+2^{31}-1$	Integer	0
long	64-bit	-2^{63}	$+2^{63}-1$	Long	0L
float	32-bit			Float	0.0f
double	64-bit			Double	0.0d
void	-			Void	

- On declaration all types are initialized
- The size of each type is fixed for every platform
- ***boolean*** is not a number, only true or false
- ***char*** is for letters while ***byte*** is numerical
- Each primitive type has a wrapper class
  - Sometimes we have to work with objects

Variables

- Variables must be declared before use
  - Can be declared anyplace in a scope
- Variable names must begin with a letter but can contain letters and digits
- Variable names are case sensitive
- Examples:

```
double myAge=29.5;
double herAge;
herAge=myAge-4;
char yes='Y';
Character c=new Character(yes);
Character c1=new Character('N');
```

Constant variables

- In Java, the keyword ***final*** denotes a constant
- Constant variables cannot be assigned more than once

```
final int max=5;   // ok
final int min;
min=0;             // ok
min=4;             // error
```

```
final A a=new A();
a.setValue(5);     // ? ok
a=new A();         // ? error
```

Operators

- Usual arithmetic operators + - * / are used in Java as in C
- divide / and modulus % as in C
- Increment ++ and decrement –
- There is no operator overloading in Java(!)

```
int x,y,m;        y=2*x+5;
x=20;             y+=5;
x++;              m=y%2;
```

Relational and Boolean operators

- Java uses the same relational operators as C
  - == (equal to)
  - != (not equal to)
  - <, >, <=, >= (less, greater, less or equal, greater or equal)

- Java uses the same bitwise operators as C
  - & (and)
  - | (or)
  - ^ (xor)
  - ~ (not)

Boolean expressions

- In Java the result of a Boolean expression is a ***Boolean*** type (true or false)
- This can't be converted to an int (as in C)
  - if (x == y) {…} // Result is a boolean
- Java eliminates a common C programming bug
  - if (x = y) {…} // Ok in C, won't compile in Java

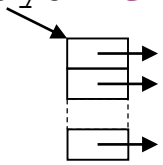
Control flow

- Java uses the same control structures as in C
- Selection (conditional) statements
  - if (..) {…}
  - if (..) {…} else if (..) {…} …. else {…}
  - switch (..) { case 1: break; … default: break; }
- Iteration statements
  - for (..) {…}
  - while (..) {…}
  - do {…} while (..);

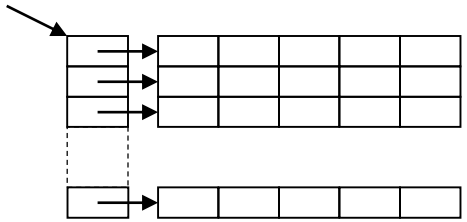
Java Arrays

Arrays

```
int[] array1=new int[20];      // 20 cells initialized to 0

int[] array2={1,2,3,4,5};      // 5 cells initialized 1-5

int[][] array3=new int[20][]; // an array of 20 un-initialized
                                 int[] pointers
```



```
int[][] array4=new int[20][5];// a matrix of 20x5 cells,
                                 its an array of 20 int[]
                                 pointers that each points
                                 to an array of 5 ints
```



```
int[][] array5={ {1,2,3} , {4,5,6} };// an initialized 2x3 matrix
```

Arrays as objects

- In Java arrays are objects
- For example they have the member ***length***:

```
int[] array1=new int[20];
for(int i=0;i<array1.length;i++)
        System.out.println(array1[i]);

int[][] array4=new int[20][5];
for(int i=0;i<array4.length;i++)
        for(int j=0;j<array4[i].length;j++)
                System.out.println(array4[i][j]);
```

Iterating an array

- A nicer syntax:

```
int[] array1=new int[20];
for(int a : array1)
        System.out.println(a);


int[][] array4=new int[20][5];
for(int[] ar : array4)
        for(int a : ar)
                System.out.println(a);
```

Copying arrays

- As Objects, arrays inherited ***.clone()*** method

```
int [] numbers = { 2, 3, 4, 5};
int [] numbersCopy = (int[]) numbers.clone();

numbersCopy[0]=7;
System.out.println(numbersCopy[0]);
System.out.println(numbers[0]);

Output
7
2
```

Copying arrays

```
int[][] array5={ {1,2,3} , {4,5,6} };// an initialized 2x3 matrix
int[][] array6=array5;    // points to the same allocated array
array6[0][0]=0;           // changed also array5[0][0] to 0
array6=(int[][]) array5.clone(); // cloning array5…
array6[0][0]=6;           // would array5[0][0] change to 6?

// creating a new copy the slow way:
array6=new int[array5.length][array5[0].length];
for(int i=0;i<array5.length;i++)
        for(int j=0;j<array5[i].length;j++)
                array6[i][j]=array5[i][j];

array6[0][0]=1;// won't affect array5
```

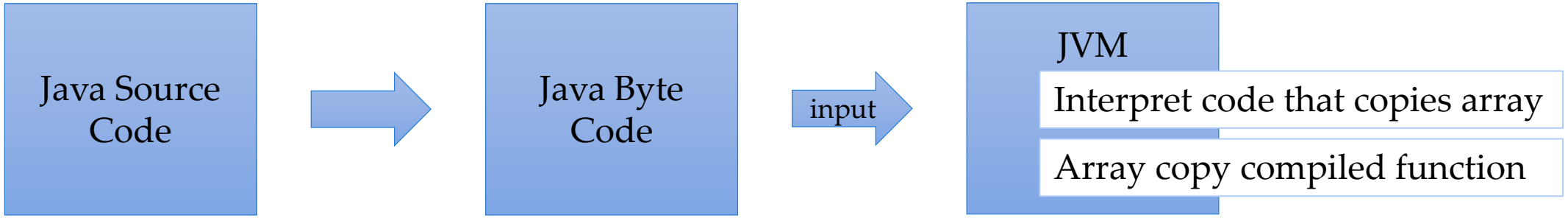
Copying arrays

- More efficient way for copying arrays is
- The function *arraycopy*
  - Provided by the System class
  - Implemented by native code, therefore faster. Defined as :
  - *public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)*

```
int [] numbers = { 2, 3, 4, 5};
int [] numbersCopy=new int[7] ;
System.arraycopy(numbers, 0, numbersCopy, 4, 3);
for(int i=0;i<7;i++)
     System.out.print(numbersCopy[i]+" ");
```

Output
0 0 0 0 2 3 4

Why is it more efficient?




```
for(int i=0;i<array1.length;i++)
        array2[i]=array1[i];

System.arraycopy(array1, 0, array2, 0, array1.length);
```

Java Strings

Strings

- Strings are sequences of characters as in C
- The standard Java library has a predefined class *String*
  - String name = “Mike”;

- Strings are *immutable* (unlike in C)
  - individual characters in the string cannot be changed
  - name[0] = ‘m’; // Not allowed!

String - concatenating

- Strings can be concatenated using the "+" operator

```
String name1 = "Israel";
String name2 = "Israeli";
String myName=name1+name2;//we will get IsraelIsraeli
```

- In Java, every object, even literals, can be automatically converted to a string

```
String p="people";

String loveJava="Java is "+4+" gr"+8+" "+p;

// Java is 4 gr8 people
```

Comparing Objects

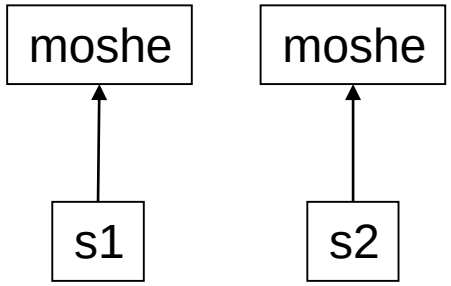
- The "==" operator compares variables values
- If the variable is of primitive type
  - "==" checks the equality of the two values

- However, if the variables are objects
  - Their value is an address
  - "==" checks the equality of the two addresses
  - In other words, checks if they are pointing to the same location = referenced to the same object

- To check if two objects are identical use the ***.equals(Object)*** function

Comparing Strings

- Strings are obviously Objects
- Do not use "==" to compare strings
- Use the "equals" function



For strings with n characters, ***.equals*** takes O(n), can it be done in O(1)?

```
String S1=new String("moshe");
String S2=new String("moshe");
if(S1==S2)
  System.out.println("equal");
else
  System.out.println("not equal");

if(S1.equals(S2))
  System.out.println("equal");
else
  System.out.println("not equal");
```

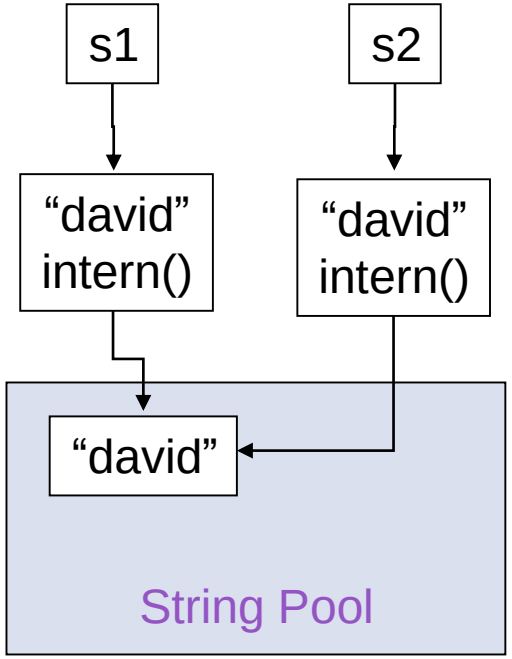
Comparing Strings

- The String Class has the method "intern"
  - The string that invoked the method is saved in a special pool (hash table)
  - The method returns the string from the pool

```
String s1=new String("david");
String s2=new String(s1);

if(s1==s2)
        System.out.println("equal");
else
        System.out.println("not equal");

if(s1.intern()==s2.intern())
        System.out.println("equal");
else
        System.out.println("not equal");

if(s1.intern()==s2.intern())
        System.out.println("equal");
else
        System.out.println("not equal");
```

O(n)

O(1)



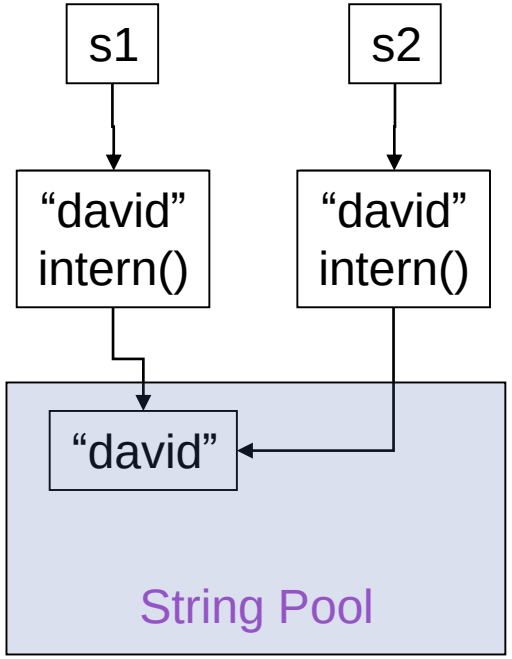
Comparing Strings

- The String Class has the method "intern"
  - The string that invoked the method is saved in a special pool (hash table)
  - The method returns the string from the pool

```
String s1=new String(input);    // "david"
String s2=new String(s1);

if(s1==s2)
        System.out.println("equal");
else
        System.out.println("not equal");

if(s1.intern()==s2.intern())
        System.out.println("equal");
else
        System.out.println("not equal");

if(s1.intern()==s2.intern())
        System.out.println("equal");
else
        System.out.println("not equal");
```

O(n)

O(1)



Quiz…

- String s1 = new String("hello");
- String s2 = "hello";
- How many objects are in memory?
- Does ?
  - s1 == s2
  - s1.intern() == s2.intern()
  - s1.intern() == s2
  - s2.intern() == s2

Intern in C#

- A similar mechanism exists in C#
  - It is called the "intern pool"
  - With each string creation the pool is checked for storing the string and a reference is kept
  - The Intern() is a static method of the String class that returns this reference

```
string s1 = "MyTest";
string s2 = new StringBuilder().Append("My").Append("Test").ToString();
string s3 = String.Intern(s2);

Console.WriteLine(s2==s1); // operator overloading for Equals – O(n)
Console.WriteLine((Object)s2==(Object)s1); // Different references -> false
Console.WriteLine((Object)s3==(Object)s1); // The same reference -> true O(1)
```

Back to Java…

String useful methods

- A *substring()* method is provided to access a substring of a larger string

```
String java="Java";
String s = java.substring(0,3);// Jav
```

- A *charAt(int n)* method returns the character at position *n* in the string

```
String java="Java";
char c = java.charAt(2);// v
```

String useful methods

- More than 50 methods were implemented
- Here is a short list of some:
  - **int** compareTo(String anotherString)
  - **int** compareToIgnoreCase(String str)
  - **boolean** startsWith(String prefix) / endWith
  - **int** indexOf(**int** ch)
  - String concat(String str)
  - String replace(**char** oldChar,**char** newChar)
  - String[] split(String regex)
  - toLowerCase / toUpperCase

Look at the Javadoc!!!

StringBuilder

- StringBuilder objects are like String object
- Except that they can be modified
- Internally, these objects are treated like variable-length array of char

```
String s=new String();
for(int i=0;i<100;i++)
        s+=i+",";// wasteful way! amortized O(n)

StringBuilder sb=new StringBuilder();
for(int i=0;i<100;i++)
        sb.append(i+",");// efficient way! amortized O(1)
```

Without StringBuilder

```
/*
 * palindrom creation without StringBuilder
 */
String palindrome = "I love java";
int len = palindrome.length();
char[] tempCharArray = palindrome.toCharArray();
char[] charArray = new char[len];

// reverse array of chars
for (int j = 0; j < len; j++)
    charArray[j] = tempCharArray[len - 1 - j];

String reversePalindrome =  new String(charArray);
System.out.println(reversePalindrome);

Output
avaj evol I
```

With StringBuilder

```
*/
* second way: with StringBuilder
/*
String s1 = "Ali Babba";

StringBuilder sb = new StringBuilder(s1);

sb.reverse();  // reverse it

System.out.println(sb);
Output
abbaB ilA
```

Java – Parameter Passing

Parameter Passing – General Idea

- In a programming language you may have
  - Value types, and Reference types
- Each can be passed *by Value* or *by Reference*
- As C++ demonstrates well:

```
void func( Point x );
void func( Point* x );
void func( Point & x );
void func( Point* & x );
```

Passing *Reference Types* by Value

RAM – Random Access Memory

Address	Content

53	rect{
	m_TopLeft,
	m_BottomRight,
	color
	}

70	p1 = 86
74	p2 = 92
78	topLeft = 86
82	bottomRight = 92
86	{ m_x=20, m_y=45 }
92	{ m_x=50, m_y=70 }

```
void main() {
  Rectangle rect;
  Point *p1 = new Point();
  Point *p2 = new Point();
  p1->Set(20, 45);
  p2->Set(50, 70);
  rect.Set(p1, p2);
  delete p1;
  delete p2;
}
```

```
// inside Rectangle class
 void Set(Point* topLeft, Point* bottomRight){
    m_TopLeft = *topLeft;
    m_BottomRight = *bottomRight;
 }
```

Passing *Reference Types* by Value

RAM – Random Access Memory

Address	Content

53	rect{
	m_TopLeft = { 20, 45 }
	m_BottomRight = { 50, 70 }
	color
	}

70	p1 = Null
74	p2 = Null
78	topLeft = 86
82	bottomRight = 92
86	{ m_x=20, m_y=45 }
92	{ m_x=50, m_y=70 }

```
void main() {
  Rectangle rect;
  Point *p1 = new Point();
  Point *p2 = new Point();
  p1->Set(20, 45);
  p2->Set(50, 70);
  rect.Set(p1, p2);
  delete p1;
  delete p2;
}
```

```
// inside Rectangle class
 void Set(Point* topLeft, Point* bottomRight){
    m_TopLeft = *topLeft;
    m_BottomRight = *bottomRight;
 }
```

Parameter passing in Java

- Recall, in Java a variable of a class type is automatically a reference type
  - Java: Point p; ≡ C++: Point* p;
- All parameters in Java are passed **"by value"**
- If it is of primitive type, changes are local
- If it is an object's reference then
  - Changing its state, has only local effect
    - e.g., an assignment like "new", "null" or another object
  - Invoking members, has a global effect obviously
    - e.g., calling some method, or setting a field

Quiz parameter passing in Java

- Consider the following *MyInt* class:

```
public class MyInt{
 int x;
 public MyInt(int x) {
  this.x=x;
 }
 public int getIntVal(){ return x;}
 public void inc(){x+=1;}
}
```

Quiz parameter passing in Java

- What would be the output of this code?

```
public static void add1(int x){x+=1;}

public static void add2(MyInt x){
 x=new MyInt(x.getIntVal() +1 );
}

public static void add3(MyInt x){
 x.inc();
}
```

```
public static void main(String[] args) {
 int x=5;
 MyInt myX=new MyInt(5);
 add1(x);
 add2(myX);
 add3(myX);
 System.out.println(x + "," + myX.getIntVal());
}
```