Java for C++ Programmers

Technion - Israel Institute of Technology

Updated: October 2015
Why Java?

- **Object-oriented** (even though not purely ...)
- **Portable** – programs written in Java language are platform independent.
- **Simpler development** – clever compiler: strong typing, garbage collection ...
- **Familiar** – took the “best” out of C++
Java highlights

• Static typing
• Strong typing
• Encapsulation
• Reference semantics by default
• One common root object
• Single inheritance of implementation
• Multiple inheritance of interfaces
• Dynamic binding
JVM is an interpreter that translates Java bytecode into real machine language instructions that are executed on the underlying, physical machine.

A Java program needs to be compiled down to bytecode only once; it can then run on any machine that has a JVM installed.
Java Virtual Machine
// file HelloWorld.java
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}

➢ javac HelloWorld.java
    The compilation phase: This command will produce the java bytecode file HelloWorld.class

➢ java HelloWorld
    The execution phase (on the JVM): This command will produce the output “Hello World!”
The main() method

- Like C and C++, Java applications must define a main() method in order to be run.
- In Java, the main() method must follow a strict naming convention.
  - public static void main (String[] args)
- main() is always a method ("member function" in C++ terminology).
  - No global functions
Types

• There are two types of variables in Java, **primitive types** (int, long, float etc.) and **reference types** (objects).

• In an assignment statement, the **value** of a primitive typed variable is copied.

• In an assignment statement, the **pointer** of a reference typed variable is copied.
The Java programming language guarantees the size, range, and behavior of its primitive types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unicode character</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit signed integers</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed integers</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed integers</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed integers</td>
</tr>
<tr>
<td>float</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td>double</td>
<td>64-bit floating point</td>
</tr>
<tr>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>

The default value for primitive typed variables is zero pattern bit.
Reference Types

• Reference types in Java are **objects**:  
  – **Identity**: location on heap  
  – **State**: set of fields  
  – **Behavior**: set of methods  

• The default value of reference typed variables is **null**
Arrays

Animal[] arr; // Nothing yet, just a reference.
arr = new Animal[4]; // Only array of pointers
for (int i = 0; i < arr.length; ++i) {
    arr[i] = new Animal();
}
// Now we have a complete array

• Java arrays are objects, so they are declared using the new operator.
• The size of the array is fixed.
• The length of the array is available using the field length.
Multidimensional arrays

```
Animal[][][] arr; // Nothing yet, just a reference.
arr = new Animal[4][]; // Only array of array pointers
for (int i = 0; i < arr.length; ++i) {
    arr[i] = new Animal[i + 1];
    for (int j = 0; j < arr[i].length; ++j) {
        arr[i][j] = new Animal();
    }
}
// Now we have a complete array
```

- Multidimensional array is an array of arrays
- Size of inner arrays can vary.
- Add more [] for more dimensions.
  - Animal[][][][] arr3D;
Strings

• All string literals in Java programs, such as "abc", are instances of `String` class.
• Strings are immutable
  – their values cannot be changed after they are created
• Strings can be concatenated using the + operator.
• All objects can be converted to String
  – Using `toString()` method defined in Object
• The class String includes methods such as:
  – `charAt()` examines individual character
  – `compareTo()` compares strings
  – `indexOf()` Searches strings
  – `toLowerCase()` Creates a lowercase copy
Flow control

Just like C/C++:

```c
if (x == 4) {
    // act1
} else {
    // act2
}
```

```c
Do / While

int i = 5;
do {
    // act1
    i--;  
} while(i != 0);
```

```c
Switch

char c = IN.getChar();
switch (c) {
    case 'a':
        // fall through
    case 'b':
        // act1
        break;
    default:
        // act2
}
```

```c
For

int j;
for (int i = 0; i <= 9; i++) {
    j += i;
}
```
For-each loop

```java
int[] array = new int[10];
int sum = 0;

// calculate the sum of array elements
for (int element : array){
    sum += element;
}
```

- Iterates over a the elements in a collection (or array).
- Preserves type safety, while removing the clutter of conventional loops.
- The loop above reads as “for each int element in array”.
- Added to C++11 as well.
Classes in Java

• In a Java program, everything must be in a class.
  – There are no global functions or global data
• Classes have **fields** (data members) and **methods** (member functions)
• Fields can be defined as one-per-object, or one-per-class (static)
• Methods can be associated with an object, or with a class (static)
  – Anyway, methods are defined by the class for all its instances
• Access modifiers (private, protected, public) are placed on each definition for each member (not blocks of declarations like C++)
Package example;

public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
    }
    public Rectangle(Point p, int w, int h) {
        origin = p; width = w; height = h;
    }

    public void setWidth(int width) {
        this.width = width;
    }
}

"this" used to call another constructor (must be placed in the first row)
Inheritance

• It is only possible to inherit from a single class.
• All methods are virtual by default

```java
public class Base {
    void foo() { System.out.println("Base"); }
}
public class Derived extends Base {
    @Override
    void foo() { System.out.println("Derived"); }
}

public class Test {
    public static void main(String[] args) {
        Base b = new Derived();
        b.foo(); // Derived.foo() will be activated
    }
}
```
Interfaces

• Defines a protocol of communication between two objects
• Contains declarations but no implementations
  – All methods are implicitly public and abstract
  – All fields are implicitly public, static and final (constants).
• An interface can extend any number of interfaces.
• Java’s compensation for removing multiple inheritance. A class can implement many interfaces.
Interfaces - Example

**Declaration**

```java
interface Singer {
    void sing(Song);  
}
```

```java
interface Dancer {
    void dance();
}
```

**Implementation**

```java
class Actor implements Singer, Dancer {
    // overridden methods MUST be public since they were declared
    // public in super class
    @Override public void sing(Song s) { }
    @Override public void dance() { }
}
```

**Usage**

```java
Dancer d = new Actor();
d.dance();
```
Abstract Classes

• An **abstract method** means that the method does not have an implementation
  – `abstract void draw();`

• An **abstract class** is a class that is declared as being **abstract**.
  – Must be so if has at least one abstract method (a class can be abstract even if it has no abstract methods, but that’s rare).
  – An abstract class is incomplete. Some parts of it need to be defined by subclasses.
  – Can’t create an object of an incomplete class: some of its messages will not have a behavior
  – Abstract classes don’t have to implement interface functions
• **final data member**
  - Constant member

```
final class Base {
    final int[] x = new int[10];
    final void foo() {
        x = new int[9]; // Error
        x[9] = 3; // OK
    }
}
```

• **final method**
  - The method can’t be overridden.

```
class Derived extends Base {
    @Override
    void foo() {}
} // Error
```

• **final class**
  - ‘Base’ is final, thus it can’t be extended
Static Data Members

• Same data is shared between all the instances (objects) of a Class.
• Assignment performed on the first access to the Class.

class A {
    public static int x_ = 1;
};

A a = new A();
A b = new A();
System.out.println(b.x_);
a.x_ = 5; // works, but confusing
System.out.println(b.x_);
A.x_ = 10; // that’s the way to go
System.out.println(b.x_);

Output

1
5
10
class A {
    public static int[] arr = new int[4];

    static {
        for (int i = 0; i < arr.length; ++i) {
            arr[i] = i;
        }
    }

    A a = new A();
    System.out.println(A.arr[1]);
    System.out.println(A.arr[2]);
}
Java Program Organization

• **Java program**
  – One or more Java source files

• **Source file**
  – One or more class and/or interface declarations.
  – If a class/interface is public the source file must use the same (base) name
    • So, only one public class/interface per source file

• **Packages**
  – When a program is large, its classes can be organized hierarchically into packages
    • A collection of related classes and/or interfaces
    • Classes are placed in a directory with the package name
Using Packages

• Use fully qualified name
  – A qualified name of a class includes the class’ package
  – Good for one-shot uses: `p1.C1 myObj = new p1.C1();`
• Use import statement
  – at the beginning of the file, after the package statement
  – Import the package member class:
    ```java
    import p1.C1;
    ...
    C1 myObj = new C1();
    ```
• Import the entire package (may lead to name ambiguity)
  – `import p1.*;`
• classes from package `java.lang` are automatically imported into every class
• To associate a class with a package, put `package p` as the first non-comment statement in a source file.
Visibility of Classes

• A class can be declared:
  – public: visible to all packages
  – default: visible only to the same package

package P1;
public class C1 { }
class C2 { }

package P2;
class C3 { }

package P3;
import P1.*;
import P2.*;

class Do { 
  void foo() {
    C1 c1; // ok
    C2 c2; // error
    C3 c3; // error
  }
}
Visibility of Members

• A definition in a class can be declared as:
  – public
    • can be accessed from outside the package.
  – protected
    • can be accessed from derived classes and classes in the same package (different than C++).
  – private
    • can be accessed only from the current class
  – default (if no access modifier is stated)
    • also known as "Package private".
    • Can be called/modified/instantiated only from within the same package.
## Visibility of Classes

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Same class</th>
<th>Same package</th>
<th>Subclass</th>
<th>Universe</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
The Object Class

• Root of the class hierarchy
• Provides methods that are common to all objects
  - boolean equals(Object o)
  - Object clone()
  - int hashCode()
  - String toString()
  - ...
The equality operator == returns true if and only if both its operands are the same.

- Compares values of primitive types.
- Compares identities of objects:

```java
Integer i1 = new Integer("3");
Integer i2 = new Integer("3");
Integer i3 = i2;

i1 == i1; // Result is true
i1 == i2; // Result is False
i2 == i3; // Result is true
```
Object Equality

• To compare between two objects the `boolean equals(Object o)` method is used:
  – Default implementation compares using the equality operator.
  – Most Java API classes provide a specialized implementation.
  – Override this method to provide your own implementation.

```java
i1.equals(i1) // Result is true
i1 == i2; // Result is false
i1.equals(i2) // Result is true
```
public class Name {
    String firstName;
    String lastName;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name other = (Name)o;
        return firstName.equals(other.firstName) &&
                lastName.equals(other.lastName);
    }
}

More on the subtleties of equals() later in the course...
Java provides wrapper classes for each of the primitive data types. These classes "wrap" the primitive in an object.

```java
// Boxing - conversion from primitive types to their corresponding wrapper classes
Character ch = new Character('a'); // boxing example
Character ch = 'a'; // auto-boxing example

// Unboxing - conversion between wrapper classes and their corresponding primitive types
Integer n = new Integer(4);
int m = n.intValue(); // unboxing example
int k = n; // auto-unboxing example

int i = Integer.parseInt("42"); // i is 42
String s1 = n.toString(); // s1 is "4"
String s2 = "a" + n; // s2 is a4
```

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Wrapper class</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
</tbody>
</table>
Garbage Collection

- C++: delete operator releases allocated memory.
  - Not calling it means memory leaks
- Java: no delete
  - Objects are freed automatically by the garbage collector when it is clear that the program cannot access them any longer.
  - Thus, there is no "dangling reference" problem.
  - Logical memory leaks may still occur if the program holds unnecessary objects.
Handling input/output

• **Class System** provides access to the native operating system's environment through **static methods** and fields.

• It has three fields:
  – The `out` field is the standard output stream
    • Default is the same console, can be changed
    • Example: `System.out.print(“Hello”);`
  – The `err` field is the standard error output stream.
    • Used to display error messages
  – The `in` field is the standard input stream.
    • Use it to accept user keyboard input.
    • Example: `char c = (char) System.in.read();`
A collection (container in C++) is an object that groups multiple elements into a single unit.

Containers can contain only objects
  – Auto-boxing can help!

The Java Collections Framework provides:
  – Interfaces: abstract data types representing collections.
    • allow collections to be manipulated independently of the details of their representation.
    • reusable data structures.
  – Algorithms: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.
Collection Interfaces and Classes

interfaces

Abstract Classes

Complete Implementations
Map Interfaces and Classes

- Map
- SortedMap
- AbstractMap
- HashMap
- TreeMap
- LinkedHashMap
An object that implements the **Iterator** interface generates a series of elements, one at a time:

- Successive calls to the `next()` method return successive elements of the series.
- The `hasNext()` method returns true if the iteration has more elements.
- The `remove()` method removes the last element that was returned by `next()` from the underlying collection.
// instantiate a concrete set
Set<Integer> set = new HashSet<Integer>();

set.add(1); // insert an elements. note the auto-boxing
int n = set.size(); // get size
if (set.contains(1)) {...} // check membership

// iterate through the set using iterator
Iterator iter = set.iterator();
while (iter.hasNext()) {
    int number = iter.next(); // note the auto-unboxing
    // do work
}

// iterate through the set using enhanced for-each loop
for (int number : set) {
    // do work
}
• Define a collection of continuous intervals of integers:
  – define an iterator class that iterates through all the integers in the interval.

```java
class Interval implements Iterable<Integer> {
    final private int start, stop, step;
    Interval(int start, int stop, int step) {
        this.start = start;
        this.stop = stop;
        this.step = step;
    }
    @Override Iterator<Integer> iterator() {
        return new IntervalIterator(start, stop, step);
    }
}
```
class IntervalIterator implements Iterator<Integer>{
    //start stepping through the array from the beginning
    private int next; private int stop; private int step;

    IntervalIterator(int start, int stop, int step){
        this.next = start; this.stop = stop; this.step = step;
    }
    @Override public boolean hasNext() {
        //check if a current number is the last in the interval
        return (next <= stop);
    }
    @Override public Integer next() {
        int retval = next; next += step; return retval;
    }
    // implement remove as well
}

for (int i : new Interval(0, 10, 2)) {
    System.out.println(i);
}
Class Collections

• Provides static methods for manipulating collections
  – `binarySearch()` searches a sorted list
  – `copy()` copies list
  – `fill()` replaces all list elements with a specified value
  – `indexOfSubList()` – looks for a specified sublist within a source list
  – `max()` returns the maximum element of a collection
  – `sort()` sorts a list

• These methods receive collections as parameters
Class Arrays

• Provides static methods for manipulating arrays
  – binarySearch() searches a sorted array
  – equals() compares arrays
  – fill() places values into an array
  – sort() sorts an array

• These methods receive arrays as parameters
Resources

• Java Tutorial -
  http://docs.oracle.com/javase/tutorial/index.html

• Java 7 API Spec -
  http://docs.oracle.com/javase/7/docs/api/