Managing Data on the World Wide-Web

Java overview

Elad Kravi,
Based on Roy Levin's presentation
Overview

1 Why Java?

2 Essentials
Wisdom of the Crowd (1)
Wisdom of the Crowd (2)

Rank of top languages on GitHub.com over time

Source: GitHub.com
Java is Design to be

- Simple, object-oriented and familiar
  - Has a similar (but incompatible) syntax to C++
- Robust and secure
  - Compile/run-time checking, no pointers
- Architecture-neutral and portable (platform independent)
- Execute with “high performance"
- Interpreted and threaded

http://www.oracle.com/technetwork/java/intro-141325.html
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
Java Vs. C++ (Machine Independence)

- The compiler generates `.class files` (not executable!)
- They contain "byte-code" which is interpreted within a Java Virtual Machine (JVM)
- Primitive types have specified sizes which are machine independent (for portability)
Java Vs. C++ (Language)

- Everything **must** be in a class
  - static methods/variables within classes instead of globals
  - no structs/unions, only **classes**
    - *enumerations have been also added*
- Built-in support for documentation – **Javadoc**
- Similar Syntax
  - loops, conditional statements, methods
  - both kinds of comments (*like C++*)
- Unlike C++, Java does not support operator overloading
Java Vs. C++ (Language)

- Generics are since java 5 (resemble templates in C++)
- No preprocessor
  - the `import` keyword allows using classes from other libraries
  - no preprocessor-like macros
- Singly-rooted hierarchy
  - all objects inherit from the root class `Object`
- No multiple inheritance
  - support multiple interfaces instead
Java Vs. C++ (Garbage Collection)

- **Pros:**
  - Frees the programmer from manually dealing with memory deallocation
  - Eliminates dangling-pointer and double-free bugs

- **Cons:**
  - may cause sudden periodic slowdown effect
  - still need to handle resource leaks
    - for instance, file reading must be **explicitly closed**
      - for this, the `finalize` method can be overridden
      - Another option – `finally` on try-catch block
Java Vs. C++ (Standard Libraries)

- Standard libraries for solving many different tasks.
- For instance:
  - collections, parsing, networking, concurrency *(all build-in)*
  - database connectivity *(via JDBC)*
  - many open source libraries
- While C++ tends to rely on non-standard third-party libraries for similar tasks.
Overview

1. Why Java?

2. Essentials
public interface Shape {
    public int area();
}

public abstract class AbstractShape implements Shape {
    protected AbstractShape(int id) { this.id = id; }  // ctor
    public abstract String name();  // abstract method
    public int id;  // member
}

public class Rectangle extends AbstractShape {
    public Rectangle(int id, String name) { super(id); this.name = name; }
    public String name() { ... }
    public int area() { ... }
    public String name;
}
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.
Each public class/interface resides in a file of its own
  - the name of the file is the same as that of the class
  - a class can contain inner classes
The top of the file contains the name of the package
Many classes can belong to a single package
  - a package is similar to a namespace in C++
When referring to classes in other packages, use the import keyword
An API is a set of packages designed to provide a well defined programmatic service.

For instance, see the Java Platform API.

It is a specification for the Java Platform 8 Standard Edition.

It contains a complete listing for packages, interfaces, classes, fields, and methods.
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

$ ls
HelloWorld.java
$ javac HelloWorld.java
$ ls
HelloWorld.java HelloWorld.class

When more classes are involved a classpath is required

main() is a member function

Standard output stream

$ ls
HelloWorld.java
$ java HelloWorld
Hello World!
There are two levels of access control:

- At the top level - public, or *package-private* (no explicit modifier)
- At the member level - public, private, protected, or *package-private* (no explicit modifier)

There is no equivalent to friend in Java.
The following table shows the access to members permitted by each modifier:

<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>
Primitive data types include everything from C
- `int, long, double, boolean, char, byte, short, …`

Declarations look like C, for example
- `double x, y; int count = 0;`

Primitive types are not objects.

Each has a corresponding object type
- `Integer, Long, Double, Boolean, Character, Byte, …`

No pointers!
Java arrays are objects, hence, declared using the new operator

- `int[] myArray = new int[5];`
- `int[] myArray = {1, 4, 9, 16, 25};`
- `String[] languages = {"Prolog", "Java"};`
- `Shape[] shapes = {new Rectangle();};`

The length of the array is available using the field length.
• Classes are arranged in a hierarchy
• The root (topmost) class is the *Object* class
• Every other class has at least one superclass
• A class may have subclasses (exception next slide).
• Each class *inherits* all the fields and methods of its (possibly numerous) superclasses
• Except for primitive types and strings, every object must be created using the *new* keyword
  – Shape s = new Rectangle(…);
Final

- **final data member**
  Constant member

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

- **final class**
  ‘Base’ is final, thus it can’t be extended

```java
final class Base {
    final int i = 5;
    final void foo() {}
}

class Derived extends Base { // Error
    // another foo ...
    void foo() {}
}
```

- For references to objects, final ensures that the reference will never change.
  - It makes no guarantees regarding the values inside the object being referred to
The Collection Hierarchy

- Interfaces
- Abstract classes
- Complete Implementations
The Map Hierarchy

- Map
  - AbstractMap
    - HashMap
    - LinkedHashMap
  - SortedMap
    - TreeMap
Basic *Object* Methods

- int i1 = 1, i2 = 1;
  - i1 == i2?
  - i1.equals(i2)?
- Integer in1 = 1, in2 = 1;
  - in1 == in2?
  - in1.equals(in2)?
- String str1 = "abc", str2 = "abc";
  - Str1 == str2?
  - Str1.equals(str2)?
- String str1 = new String("abc"), str2 = new String("abc");
  - Str1 == str2?
  - Str1.equals(str2)?

For your own classes you should consider overriding
- *equals*
- *hashCode*
- *toString*
Expressions are Like in C

- Assignment statements
  - =, +=, *=, /= etc.
- Arithmetic uses the familiar + - * / %
- i++, i--, ++i, --i
- Boolean operators && || !
- Comparisons < <= == != >= >
- Even logical operations i&0x1, i|0xf, i>>
Control Statements are Like in C

• if (x < y) smaller = x;
• if (x < y) { smaller=x; sum += x; } else { smaller = y; sum += y; }
• while (x < y) { y = y - x; }
• do { y = y - x; } while (x < y);
• for, Switch...

• int [] array = new int[10];
  // calculate the sum of array elements
  for (int curr: array){sum += curr;}  // for-each loop
• Java is **case-sensitive** – i.e., *maxval*, *maxVal*, and *MaxVal* are three different names
• Class names begin with a capital letter
  – *public class Rectangle { ...*
• All other names begin with a lowercase letter
• Subsequent words are capitalized
  – *int bigOne = 1;*
• Underscores are **NOT** used in names
• These are **very strong** conventions!
public class MathUtil {
    public static int distance(Point p1, Point p2) {
        ...
    }

    public static final double pi = 3.14159;
}

• Exceptions are events that can be triggered during a program run
• These events disrupt normal program flow
  – never use exceptions for returning special values
• There is an hierarchy of exception types
• The root class is:
  – RuntimeException – for exceptions that are not declared
  – Exception – for exceptions that are declared
int main( ...) {
  try {
    foo();
  }
  catch (IOException e) {
    ...
  }
  catch (MyException e) {
    ...
  }
}

int foo() throws MyException {
  ...
  bar();
  ...
  return 5;
}

void bar() throws MyException {
  ...
  if (some_condition)
    throw new MyException();
  ...
}
void myFunc() throws IOException {
    InputStream is = null;
    try {
        is = new FileInputStream("tmp.txt");
        // do something with ‘is’
    }
    finally {
        is.close();
    }
}
Small Example

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Gets called no matter what. Typically used to avoid resource leaks.
• reflection - the ability of a program to examine and modify its own structure and state at runtime
• A Class object is associated for every loaded class by the JVM
• The Class object reflects the class it represents
  
  ```java
  Object o = new Integer(5);
  System.out.println(o.getClass().getName());
  ```
• Dynamic instantiation is possible
• Major performance impact
public class MyObject {
    public Integer foo(Object o) {
        ...
        return new Integer(5);
    }
}

MyObject o = new MyObject();
Integer i = o.foo(new Integer(5));

MyObject o = new MyObject();
int i = o.foo(5);

Boxing supports primitive to corresponding object conversion (and vice-versa)
Generics

Collection<String> lst = new ArrayList<String>();
lst.add("sad");       // ok
lst.add(1);           // compilation error

// how about the following:
Collection<int> lst = new ArrayList<int>();

Will not compile – int is not an object!
The resulting HTML from running Javadoc is:

```java
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */

public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
- url - an absolute URL giving the base location of the image
- name - the location of the image, relative to the url argument

Returns:
- the image at the specified URL

See Also:
- Image
Java 7
Noticable features

- **Diamond Operator** – auto infer of generic collections type
  
  ```java
  Map<String, List<Trade>> trades = new TreeMap<>();
  ```

- **Strings in switch statements**

- **Improved exception handling**
  
  ```java
  try {
      methodThatThrowsThreeExceptions();
  } catch (ExceptionOne | ExceptionTwo | ExceptionThree e) {
      // log and deal with all Exceptions
  }
  ```
Java7
Noticeable features

• Automatic resource management – implements java.lang.AutoCloseable interface

```java
try(File f = new File("c:/myFile.txt"){
    // do something with f

} catch (IOException e){
    ...
}
```
• close() is called when exiting the try block
• No need to use finally
Java7
Noticeable features

• **New file system API**
  – New interfaces like Path, Paths, FileSystem, FileSystems and others
    ```java
    Path path = Paths.get("c:\Temp\temp");
    Files.delete(path);
    Files.deleteIfExists(path);
    ```

Java 8 (Since March 2014)

- Lambda expressions
- Streams
- Method and constructor references
- Default methods
- Recommendation for extra reading:
  - http://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html
  - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html
Lambada Expressions (informally, "closures" or "anonymous methods")

Today:

```java
collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

A single method interface is a very common convention in Java.

- `java.lang.Runnable` - void run()
- `java.util.concurrent.Callable` - V call() throws Exception
- `java.util.Comparator` - int compare(T o1, T o2)
- `java.awt.event.ActionListener` - void actionPerformed(ActionEvent e)

These are called **Functional Interface**
Lambada Expressions

Today:
```java
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

**Some problems (anonymous inner classes):**
- Bulky syntax
- Confusion surrounding the meaning of names and this
- Inflexible class-loading and instance-creation semantics

**Lambada expressions:**
```java
(int x, int y) -> x + y
```

parameters

body

From open JDK project
Today:
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});

With Lambada expression:
Collections.sort(people,
    (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));

From open JDK project