Concurrent and Distributed Programming

Java for C/C++ Programmers
Lecture 1

Based on slides from Introduction to Software Engineering
Why Java?

- **Object-oriented** - Java is a single-root, single-inheritance object oriented language
- **Multithreaded** - Java has a built-in support for multithreading
- **Distributed** - Using Java RMI (remote method invocation) you can access objects on other machines almost as if they were local
- **Portable** - programs written in the Java language are platform independent
- **Simpler development** – clever compiler
JVM – Java Virtual Machine

- 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
In C++ we use the `delete` operator to release allocated memory. Not using it means memory leaks.

In Java there is no `delete` and there are no memory leaks. 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 in Java.
// 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
HelloWord.class

> java HelloWorld

The execution phase (on the JVM): This command will produce the output
"Hello World!"
Naming Conventions

- Methods and variables start with a leading lowercase letter
  - next, push(), index, etc.
- Classes start with a leading uppercase letter
  - String, StringBuffer, Vector, Calculator, etc.
Naming Conventions – Cont.

- Constants (final) are all upper-case: DEBUG, MAX_SCROLL_X, CAPACITY
  - final double PI = 3.1415926;

- Word separation in identifiers is done by capitalization (e.g. maxValue), except for constants where underscore is used (e.g. MAX_SCROLL_X)
The main() method

- Like C and C++, Java applications must define a main() method in order to be run.
- In Java code, the main() method must follow a strict naming convention. All main() methods must be declared as follows -
  - public static void main(String[] args)

Like in the example we saw -
```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World !");
    }
}
```
Variables

- 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
Primitive Types

- 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 signed integers</td>
</tr>
<tr>
<td>double</td>
<td>64-bit signed integers</td>
</tr>
<tr>
<td>void</td>
<td>-</td>
</tr>
</tbody>
</table>

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

- Reference types in Java are **objects**
- An object has a set of **data members (attributes)** and a set of **methods**
- All reference typed variables are **dynamically allocated** from **heap** at runtime (and can’t be explicitly deallocated by the programmer)
- Referenced typed variables can’t be dereferenced (no reference * or derefernce & operators)
- The default value of reference typed variables is **null**
## Reference Types

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyObject x</td>
<td>MyObject* x (not initialized !!!!)</td>
</tr>
<tr>
<td>N/A</td>
<td>MyObject x(5)</td>
</tr>
</tbody>
</table>

Since we’re handling pointers, the following is obvious:

![Diagram showing pointer operations and assignments]
Wrappers

- Java provides objects which wrap primitive types.

Example:

```java
Integer n = new Integer(4);
int m = n.intValue(); // java 1.4
int k = n; // java 1.5 - autoboxing
```
Flow control

It is like C/C++:

```java
if(x==4) {
    // act1
    i--;
} else {
    // act2
}
```

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

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

```java
int j;
for(int i=0;i<=9;i++)
{
    j+=i;
} // java 1.4
```
Arrays

- Java arrays are objects, so they are declared using the new operator
- The size of the array is fixed

```java
Animal[] arr; // nothing yet ...
arr = new Animal[4]; // array of references
for(int i=0 ; i < arr.length ; i++) {
    arr[i] = new Animal();
// now we have a complete array
```
Multidimensional arrays

- Multidimensional array is an array of arrays
- Size of arrays may not be the same

```java
// building triangular array
Animal[][] arr; // nothing yet ...
arr = new Animal[4][]; // array of references to arrays
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();
    }
}
```
for-each loop

```java
int[] arrayOfPrimitive = new int[10];
for(int k: arrayOfPrimitive){
    k+=1; // ERROR: this has no effect
    // k is passed by value
}

----BUT the following is OK-------

int[][] arrayOfObj = new int[10];
for(int i = 0; i < arrayOfObj.length; i++) {
    arrayOfObj[i] = new int[i + 1];
}
for(int[] k: arrayOfObj){
    for (int i = 0; i < k.length; i++) { k[i] = i; }
}
```
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 (functions)
- Fields and methods are defined to be one-per-object, or one-per-class (static)
- Access modifiers (private, protected, public) are placed on each definition for each member (not blocks of declarations like C++)
package example;

public class Rectangle {

    // Data members
    public int width = 0;
    public int height = 0;
    public Point origin;

    // Constructors
    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;
    }

    // A method
    public void setWidth(int width) {
        this.width = width;
    }
}
Managing Source and Class Files

- Source code is placed in a text file whose name is the *simple* name of the *single public* class or interface with the *SAME name* contained in that file and whose extension is `.java`

- Example: `Rectangle.java`

```java
package com.taranis.graphics;

public class Rectangle {
    ...
}

class Helper {
    ...
}
```
Packages

- Physically and logically bundles a group of classes
  - Classes are placed to the directory with the package name
  - Avoid naming conflicts
  - Addition access control
    - Unrestricted access between classes of the same package
    - Restricted access for classes outside the package
Creating a Package

- place a `package` statement at the top of the source file in which the class or the interface is defined.
- If you do not use a package statement, your class or interface ends up in the `default package`, which is a package that has no name.

Compilation: `javac p1/C1.java`
Invocation: `java p1.C1`

```java
package p1;
public class C1 {
    public static void main(String[]){...}
    ...
}
class C2 {...}
```
Using Packages

- Packages often used to bundle libraries which provide some specific functionality (more on this later)

- To use a class in a package
  - Use *qualified* name (only public classes are accessible outside the package)
    - A qualified name of a class includes the package that contains the class
    - Good for one-shot uses
      - Ex: `p1.C1 myObj = new p1.C1();`
  - Or...
Import statement

- An import statement should be at the beginning of the file, after the package statement

- Import the package member class (When only a few members of a package are used)
  ```java
  import p1.C1;
  ...
  C1 myObj = new C1();
  ```

- Import the entire package (may lead to name ambiguity)
  ```java
  import p1.*/;
  ...
  C1 myObj = new C1();
  ```

- Not physically including code like C++. Also, an import is not transitive.
As opposed to C++, it is possible to inherit only from ONE class.

class Base {
    Base() {}
    Base(int i) {}
    protected void foo() {...}
}

class Derived extends Base {
    Derived() {}
    protected void foo() {...}
    Derived(int i) {
        super(i);
        ...
        super.foo();
    }
}
In Java, all methods are "virtual":

class Base {
    void foo() {
        System.out.println("Base");
    }
}

class Derived extends Base {
    void foo() {
        System.out.println("Derived");
    }
}

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

- **All classes** implicitly inherit from the class `java.lang.Object`
- Root of the class hierarchy
- Provides methods that are common to all objects (including arrays)
  - `boolean equals(Object o)`
  - `Object clone()`
  - `int hashCode()`
  - `String toString()`
  - ...
The equality operator == returns true if and only if both its operands have the same value.

- Works fine for primitive types
- Only compares the values of reference variables, not the referenced objects:

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

i1 == i1 && i1 != i2 && i2 == i3
```

This expression evaluates to 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) && i1.equals(i2)
```

This expression evaluates to `true`
Example: Object Equality

```java
class Name {
    String firstName;
    String lastName;
    ...
    public boolean equals(Object o) {
        if (!(o instanceof Name)) {
            return false;
        }
        Name n = (Name)o;
        return firstName.equals(n.firstName) &&
               lastName.equals(lastName);
    }
}
```
Abstract Classes

- *abstract* method means that the method does not have an implementation
  - abstract void draw();
- *abstract* class, is a class that can not be instantiate
- There are two ways to make your class abstract:
  - Use the keyword `abstract` in the class declaration
    - abstract class Number {
  - Every class with at least one abstract method is an abstract class (no need to explicitly write the `abstract` keyword in the class declaration)
Interface

- 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).
- Whereas a class can extend only one other class, an interface can extend any number of interfaces. The list of super-interfaces is a comma-separated list of all the interfaces extended by the new interface.
- Java’s compensation for removing multiple inheritance. You can implement as many interfaces as you want.
interface Singer {
    void sing();
}

interface Dancer {
    void dance();
}

class Actor implements Singer, Dancer {
    // overridden methods MUST be public
    public void sing() { ... }
    public void dance() { ... }
}
Static

- **Data members** - same data is used for all the instances (objects) of some Class.

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

A a = new A();
A b = new A();
System.out.println(b.x_);
a.x_ = 5;
System.out.println(b.x_);
A.x_ = 10;
System.out.println(b.x_);
```

Assignment performed on the first access to the Class.
Only one instance of ‘x_’ exists in memory

Output:

```
1
5
10
```
Methods

- Static method can access *only* static members
- Static method can be called without an instance.

```java
class TeaPot {
    private static int numOfTP = 0;
    private Color myColor_;
    public TeaPot(Color c) {
        myColor_ = c;
        numOfTP++;
    }
    public static int howManyTeaPots() {
        return numOfTP;
    }

    // error :
    public static Color getColor() {
        return myColor_;  // should return Color instance
    }
}
```
Usage:

TeaPot tp1 = new TeaPot(Color.RED);
TeaPot tp2 = new TeaPot(Color.GREEN);
System.out.println("We have " +
    TeaPot.howManyTeaPots() + " Tea Pots");
**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
class Base {
    final int i = 5;
    final void foo() {
        i = 10;
    }
}

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

- public class
  - 'new' is allowed from other packages
  - Default (if public isn’t stated)
    - 'new' is allowed only from the same package

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

package P2;
class C3 { }

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

public class DO {
    void foo() {
        C1 c1 = new C1();
        C2 c2 = new C2(); // ERROR
        C3 c3 = new C3(); // ERROR
    }
}
```
Access Control – Cont.

- **public** member (function/data)
  - Can be called/modified from outside the package.

- **protected** member (function/data)
  - Can be called/modified from derived classes and other classes in the same package.

- **private** member (function/data)
  - Can be called/modified only from the current class.

- **Default (if no access modifier is stated)**
  - Usually referred to as “Friendly” or "Package access".
  - Can be called/modified/instantiated only from within the same package.

- **We didn’t talk about nested classes here**
Access Control - Cont.

Package P1

Base

Protected

Derived

Friendly

SomeClass2

Package P2

SomeClass

↑ — Usage

↑ — Inheritance
Exceptions

- Thrown upon abnormal condition occurred somewhere in the program
- Caught and handled by the caller, or thrown further
- Contributes to the code structure clarity
Checked and unchecked exceptions

- Checked exceptions are enforced to be caught by compiler. Used mostly for indicating software problems in the program logic.

- Unchecked exceptions are not checked by compiler and used usually by JVM to indicate critical errors.
Exceptions in Java
Exceptions in Java

```java
void foo(int x) throws ThreeException, FourException {
    if (x==3)
        throw new ThreeException("Problem 3");
    if (x==4)
        throw new FourException("Problem 4");
}
void foo2() {
    try {
        foo(3);
    } catch(ThreeException te) {
        System.err.println(te.getMessage());
    } catch(Exception g) {
        g.printStackTrace();
    }
}
```
class ThreeException extends Exception {
    public ThreeException() {
    }
    public ThreeException(String msg) {
        super(msg);
    }
}
Finally statement

Example:

```java
try{
    // the code opens a file
} catch (Exc1 e1) {
    ..
} catch (Exc2 e2) {
    ..
} finally {
    // We should close the file
    // no matter how we exit try-catch block
}
```
Containers – short overview

- Java is supplied with an extensive toolbox of easy-to-use data structures provided in Java distribution (see java.util package)
- They include: List, Map (hash table), Set, Vector, Queue and many many others.
- Containers can contain only objects (no primitive type containers! But can be tricked by wrappers or arrays).
- Some containers are thread safe, some are not.
- For more details look API Javadoc.
Other convenient features

- Generics: like C++ templates
- Reflection
- Type-safe enums
- Nested classes
- ... And many other useful things – now that Java 7 is finally out!