

## Lecture 20

### A tiny introduction to Java

We shouldn't need to introduce yet another language... but it is just too popular, and just used too widely in year 2 and 3 development that we must say a few words about it.

#### References:

- A short Java tutorial is available here: <http://java.sun.com/docs/books/tutorial/>
- Java docs: <http://java.sun.com/j2se/1.4/docs/index.html>

PMOOP(0396A)

PMOOP(0396A)-20.1

The word Java is overloaded with 2 meanings:

- The Java programming **language**: how your programs look like.
- The Java **virtual machine**: how an implementation of Java should compile, and run your program, and what is provided by the environment.

The target: **one should be able to run the same program in different platforms, without sacrificing too much efficiency.**

I.e., combine some efficiency of C++ with some machine independence of Python. How to achieve that?

- Programs are **compile into Java bytecode** instead of machine code. You can see this as Java defining a common assembly language, so that programs compile into that language.
- Java bytecode is then **interpreted by a java interpreter** in the VM. Some interpreters have "just in time" compilation which turn some byte-code into real machine code.

### Java: comparison with C++ and Python

Attribute	C++	Java	Python
Speed	Fast	Medium	Slow
Compiled	Yes		No
Code runs in	CPU	Interpreter/CPU	Interpreter
Object model	Value based	Reference based	
GC	External or none	Built-in	
Var allocation	Global, auto, dynamic	Static for built-in types, dynamic for classes	Dynamic
Abstractions	Functions/types	Classes	Objects
Templates	Yes	Not yet	Not needed
Low-level ops	Direct	Through C/C++ functions	

Rather than "a better C++", I'd call it "a faster Python".

PMOOP(0396A)-20.2

PMOOP(0396A)-20.3

### Hello-world in Java

```
// Must be written in a file Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!\n");
    }
}
```

- The program is a class. **The primary abstraction of Java is class**, not function. If you want a "normal" function, define a static class function.
- Both classes and functions have access specifier, here "public". Unlike Python, it supports full encapsulation.
- Since primary abstraction is class, quite clumsy to write the println: find the class (System), look for the field (out), and call the method (println). In Java, data member is called "field", member function is called "method".
- Each public class is stored in a **separate source file**.

### Compiling and running

To compile the Java program:

```
javac Hello.java
```

This produce a file Hello.class. To run it, type

```
java Hello
```

- Why needing to compile? To check syntax problems, and to avoid having to parse the program everytime we run it.
- Each class of the program (public or not) will create one class file. The Java run-time environment **load** these classes to memory for running.
- The Java interpreter also needs to load the **built-in classes** like System we used in our program. They are stored in an archive called `rt.jar`, installed when you install the Java interpreter. (E.g., `/usr/java/jre1.3.1/lib`). `rt` stands for run-time, so it is the run-time library.

PMOOP(0396A)-20.4

PMOOP(0396A)-20.5

### Everything is a class!

A class can contain everything we want. E.g.,

```
public class MyClass {
    public final int myConst = 10; // Constants
    private int myField;
    private static int myVar; // "global variable"
    public MyClass() { // No initializer
        myField = 0;
    }
    public int myMethod() {
        return myField; // No need for self, like C++
    }
    public static void myFunc() {
        System.out.println(myVar);
    }
    public final int myMethod2(int x) { // Can't be overridden
        return myField += x;
    }
}
```

## All the (primitive) types

Java has an unextensible set of primitive types:

- Integer types: **byte**, **short**, **int** and **long**, to represent 8-bit, 16-bit, 32-bit and 64-bit integers. No signed and unsigned differentiation.
- Floating point types: **float**, **double**, to represent 32-bit and 64-bit IEEE floating point numbers.
- Representation types: **char**, **boolean**, to represent characters and boolean. Unlike C++, they are not numbers, and can't be assigned to and from numbers. And unlike C++, **char** is 16-bits in Unicode.

They have the properties that **value semantics** is used, similar to C++. So if you write "**int** x;" in a function, you have allocated an integer.

Seems obvious. But read on...

Note that even *String* is not a primitive type—although you can write strings like "*my string*". So Java *String* is a "class with language support".

PMOOP(0396A)-20.6

## Objects

- Non-primitive type objects are created **in free store**, using the **new** operator. They are cleaned up by garbage collector automatically. Conversely, you can't **new** a primitive type.

- So if you write a class *MyClass*, you can create a variable using "*MyClass* x;", but in fact it is just a pointer (or **reference**, in terminology of Java). You still need to allocate it, e.g., "**x = new MyClass()**;".

- Like C++, there is a *null* pointer (but written in lower case).

- **Array types are non-primitive**, so you also need to **new** them. E.g.,

```
int[] x = new int[10];
```

- All classes are derived from the *Object* type. This includes even the array types.  
So it works like **void** in C++, except that type-safe casts can be used on it.

- One can also has an array of references, like *MyClass[]*, *int[][]*, etc.

PMOOP(0396A)-20.7

## Similarity with C++

**Superficially, Java looks very like C/C++.** Every C constructs that is not related with objects and pointers works the same way in Java.

E.g., a Fibonacci number printing program:

```
public class Fibo {
    public static void main(String[] args) {
        int last = 0, curr = 1;
        for (int i = 0; i < 10; ++i) {
            System.out.println(curr);
            int temp = last;
            last = curr;
            curr += temp;
        }
    }
}
```

PMOOP(0396A)-20.8

## Argument passing

Function arguments are **always passed by value**. To allow changing content, one **must** use non-primitive types. E.g.,

```
public class ArgPassing {
    // Pass by value!
    static void TestFunc
        (int x) {
        ++x;
    }
    public static void main
        (String[] args) {
        int x = 10;
        TestFunc(x);
        System.out.println(x);
    }
}

class IntArg { // For argument passing
    int val;
}
public class ArgRPassing {
    static void TestFunc (IntArg x) {
        ++x.val;
    }
    public static void main
        (String[] args) {
        IntArg x = new IntArg();
        x.val = 10;
        TestFunc(x);
        System.out.println(x.val);
    }
}
```

PMOOP(0396A)-20.9

## Inheritance model

Java allows inheritance. Every class implicitly inherits *Object* (directly or indirectly). If you want to inherit from another class, use **extends**:

```
public class MyExtended extends MyClass { // Inherit from MyClass
    public MyExtended() {
        super(); // Chaining. Must be first stmt.
    }
    public int myMethod() { // Non-final functions can be overridden
        super.myMethod(); // Chaining
        return 10;
    }
}
```

- Again, the compiler allow you to put a derived type whenever the base type is expected, and automatic casting is done.

- The reverse can't be done, and need **casting**. The casting operator looks like C-style casts: *(MyExtended)my\_value*.  
but it acts like C++ *dynamic\_cast*, and return *null* if failed.

PMOOP(0396A)-20.10

## Multiple inheritance??

- There is **no** concept of multiple inheritance in Java.

- But there is a concept of **interface implementation**. A class can implement multiple interface.

- Interface is written this way:

```
interface CanTeach {
    public TCourse[] GetCourseTaught();
    public void TeachCourse(TCourse);
}
```

- What's so different between interface and class? Two differences: an interface **can't have any implementation or field**, and it is **implemented**, not **extended**, by a class.

- The implementing class must **provide all the functions** defined by the interface.  
No implementation reuse!! But we can do aggregation.

PMOOP(0396A)-20.11

## Member class

We can nest a class in another, i.e., define **member classes**.

```
public class Outer {
    // Note the static!
    static class Inner {
        int myFunc() {
            return ifield;
            // Can't use ofield!
        }
        private int ifield;
    }
    private int ofield;
    Inner GetInner() {
        return new Inner();
    }
}

public class Outer {
    // No static, can use fields of Outer!
    class Inner {
        int myFunc() {
            return ifield + ofield;
        }
        private int ifield;
    }
    private int ofield;
    Inner GetInner() {
        // Can be written as new Inner()
        return this.new Inner();
    }
}
```

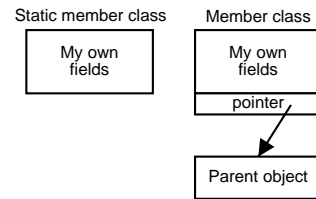
We can also nest classes in C++, and the effect is a nested class.  
Note the strange syntax for new. Unluckily, this is not the only strange syntax.

PMOOP(0396A)-20.12

## Object structure

How the member class can **access data members of its enclosing class**?

The object looks like this in the memory...



This explains why we need to provide an outer class object when we new a member class object.

And shows what you will do if you need to do it in C++.

PMOOP(0396A)-20.13

## Exception model

Java exception mechanism is similar with C++, but...

- There is **no "resources acquisition is initialization" technique**, so a **finally** clause frees resources, like Python.
- Most Java exceptions are "checked exceptions". Functions have to declare that it wants to throw it, and the compiler checks for it:

```
import java.io.*;
public class InputNum {
    public static void main(String[] args) throws IOException {
        int x;
        Reader in = new InputStreamReader(System.in);
        BufferedReader bin = new BufferedReader(in);
        String s = bin.readLine(); // Can throw IOException!!
        x = Integer.parseInt(s);
        System.out.println(x);
    }
}
```

PMOOP(0396A)-20.14

## I/O library model

- With no facility for generic programming, the C++ trick to have a templated char and wchar\_t streams is not possible in Java.
- With no multiple inheritance, Java can't mix input and output formatting nicely.
- The designers chooses to implement **4 sets of classes** for I/O: input and output for (8-bits) byte-streams and (16-bits) char-streams. They are called InputStream, OutputStream, Reader and Writer.
- There are different sub-classes of these basic classes to deal with different buffering strategies and I/O sources and destinations.
- **Formatting is done minimally** in these streams. They are moved to the **StringTokenizer** and **String** class instead.
- The system is **far inferior** to the C++ I/O system.

PMOOP(0396A)-20.15

## Collections in Java

- Java still got no generic programming tools, so **all collections are Object collections**.  
It seems that everything is just not bad enough to push away programmers.
- In other words, you have to **cast references back to your desired type** after you get things out of the collection.
- The framework consists of interfaces defining abstract containers, and classes defining their implementations.
- **Abstract interfaces** are list, set, sorted set, map and sorted map. They define the operations that can be done for all implementations.
- **Implementations** include linked list, vectors, hash set, tree set, hash map, hash map and weak reference hash.  
Note: deque absent! (On the other hand, hash absent in STL.)

PMOOP(0396A)-20.16

## Wrapper classes in Java

- But there is a more funny consequence of using *Object* as component type... you can't have a collection of **primitive types**!
- To deal with that, we need a non-primitive type that has the primitive type as the only element, just like the case when we do argument passing.
- Java has a class for each primitive type. E.g., *Integer* represent an integer. So you can **new Integer(6)**; to produce one.
- You can call functions like *intValue()* to recover its value.
- But the class is **immutable**, i.e., provides no method for modifying the content. If you really need a mutable type, then you have to define your own.

PMOOP(0396A)-20.17

### A few words about exam...

- Includes lecture 1–19.
- Includes 7 multi-part questions, each 10% of the marks.
- Marks of top 3 scoring questions will be double-counted.  
So it is important to get a couple of them to score completely.
- Includes both coding tasks and concept tasks.
- Include questions closely related to assignment, and include some other questions.
- You can bring 1 A4 paper into the exam hall, with any written or printed materials. But 1 A4 paper means 1 A4 paper, not 1 A4 paper with many smaller-sized paper stucked onto it.  
Last year my Algorithm course has one student doing so, and the whole sheet of paper is taken away from him.

PMOOP(0396A)-20.18

### Assignment 4, part 1

- You need to implement an object class and a weak pointer class. Both are template classes.
- The object class has a **member which is of type T**, a template parameter. A reference to it can be obtained using the Get function.
- The pointer **looks like an ordinary pointer**: using it, you can access the data within the object by dereference and by  $\rightarrow$ .
- But there is one big difference between pointers and weak pointers: when the object is deleted...
  - A normal pointer would **still be pointing to the deleted location**. Accessing it results in strange memory problem which is hard to debug.
  - A weak pointer would become NULL automatically. Accessing it results in the *bad\_dereference* exception being thrown.

PMOOP(0396A)-20.19

### How to get that effect?

- The object must keep not just the T-type data member, but some **extra information** telling what needs to be done if the object is deleted.
- In the simplest design, the object keeps a **set of weak pointers** pointing to the object.
- This set of weak pointers **must be updated** whenever a new pointer points to the object, or an old pointer pointing to it is changed to point to another object (or is deleted).
- When the object is deleted (i.e., when the destructor is executed), **all weak pointers** are changed to contain 0.
- One last hint: **don't iterate through the set to delete each element of the set**: the iterator will become invalid when you delete it.
- Instead, find the first element of the set, delete it and repeat.

PMOOP(0396A)-20.20

### part 2: Python programming

- You need to write a Python script that reads the **directory structure leading to a directory**, and show the **files** within the directory.
- You are to display the results in two **list widgets** using the Tkinter library, one for the files, one for the directory structure.
- The directory structure listing allows you to **select another directory** for showing, and display the used, remaining and total **disk space in the filesystem** hosting that directory.
- The file listing allows you to **display the type of the files**, using a standard Unix command called "file".  
It examines file data and look for "magic words" to determine the file type.
- The GUI must have a menu, allowing you to create **multiple windows** showing **different directories**.

PMOOP(0396A)-20.21

### View the following pages

#### Python:

- Look for the os module in the library reference. It tells you how to manipulate directories (os.getcwd, os.listdir, os.statvfs, os.path).
- Use the command module to execute a command and get its output.
- You probably need to play with some Python data structures like lists and strings. Look into the library doc about the string module and about built-in "mutable sequence types".

#### Tk:

- You will need to create new top-level windows. The Toplevel widget allows you to do that.
- The lists should be implemented by Listbox widgets. Check the documentation to see how to make it shrink and grow like other widgets.
- Follow the hints in the assignment sheet.

PMOOP(0396A)-20.22