Objectives

- No background assumptions
- Learn the Java programming language: syntax and semantics
- Become comfortable with object oriented programming: Learn to think in objects
- Learn the essentials of the Java class library, and learn how to learn about other parts of the library when you need them.
- Practicalities left to prácticas

References

- Deitel, Java How to Program, Third Edition (1999)
- Web Resources (reference manuals, etc...)

Introduction to Java

(Deitel ch.1, Eckel ch.1)

Plan

1. Basic Concepts
2. History of Java
3. Java Class Libraries
4. Basics of a Typical Java Environment
5. Thinking About Objects
6. Interfaces
7. Services
8. The Hidden Implementation
9. Inheritance
10. Polymorphism

Basic Concepts

- Is there a program that can be written in one language for which there is really no equivalent program in one of the other languages?
  - e.g. Is there a program that can be written in Java that has no equivalent in Pascal?

- Sapir-Whorf Hypothesis:
  - the languages we speak directly influence the way in which we view the world.
Basic Concepts

Church's thesis

• Any computable function can be computed by a Turing machine.

  – Why do we need all these different languages?

Basic Concepts

• Compiler
  – Compiler is a translator
  – Translates a high-level language into an array of machine language.

• Interpreter
  – Execute high-level language programs without compilation
    1. Gets next instruction from memory
    2. Decodes the instruction into primitive machine operations
    3. Executes the operations

Basic Concepts

• Machine language
  – +130004774; +1400593419; +1200274027
  – “Natural language” of computer component
    – Machine dependent

• Assembly language
  – LOAD BASEPAY; ADD OVERPAY; STORE TOTAL
  – English-like abbreviations represent computer operations
    – Translator programs convert to machine language

• High-level language
  – total = basePay + overTimePay
  – Allows for writing more “English-like” instructions
    – Compiler convert to machine language

• O-O language (O-O Paradigm)

Basic Concepts

• Java
  – Originally for intelligent consumer-electronic devices
  – Then used for creating Web pages with dynamic content
    – Now also used for:
      • Develop large-scale enterprise applications
      • Enhance WWW server functionality
      • Provide applications for consumer devices (cell phones, etc.)

Java Class Libraries

• Java Programs consist of pieces called classes

• Classes
  – Include methods that perform tasks
    • Return information after task completion
    • Used to build Java programs

• You can program each class or make use of Java class libraries

• Java class libraries
  – Known as Java APIs (Application Programming Interfaces)
Notes

Avoid reinventing the wheel. Use existing pieces

Typical Java building blocks
  Classes from class libraries
  Your own classes and methods
  Other people’s classes and methods

Class libraries can improve program performance

Class Libraries can also improve portability

Basics of a Typical Java Environment

• Java System: environment, language, API
• Java programs normally undergo five phases
  – Edit
    • Programmer writes program (and stores program on disk)
  – Compile
    • Compiler creates bytecodes from program
  – Load
    • Class loader stores bytecodes in memory
  – Verify
    • Verifier ensures bytecodes do not violate security requirements
  – Execute
    • Interpreter translates bytecodes into machine language

Notes

Write programs as simple as possible

Simply writing programs in Java does not guarantee portability

Interpreter vs Compiler

Thinking About Objects

• Objects
  – Software components that model real-world items
  – Look all around you
    • People, animals, plants, cars, etc.
  – Attributes
    • Size, shape, color, weight, etc.
  – Behaviors
    • Babies cry, crawl, sleep, etc.
  – Objects that are identical except for their state during a program’s execution are grouped together into classes

An object has an interface

• How do you get an object to do useful work for you?
• The requests you can make of an object are defined by its interface

```java
Light lt = new Light();
lt.on();
```
An object provides services

- Think of an object as a service provider
- “I could magically pull them out of a hat, what objects would solve my problem right away?”
- **High cohesion** means that the various aspects of a software component (such as an object) “fit together” well.
- One problem people have when designing objects is cramming too much functionality into one object.

The hidden implementation

- **Class creators and client programmers**
- Class creator exposes only what’s necessary to the client programmer and keeps everything else hidden
- Creator can change the hidden portion at will without worrying about the impact on anyone else.
- The hidden portion could easily be corrupted by a careless or uninformed client
- Hiding the implementation reduces program bugs
- In Java
  - public
  - private
  - protected

Inheritance: reusing the interface

- It seems a pity to create a class and then be forced to create a brand new one that might have **similar functionality**
- take the existing class, **clone it**, and then make additions and modifications to the clone → **inheritance**
- **base class** or **superclass** or **parent class**
- **derived class** or **inherited class** or **subclass** or **child class**
- if base class is changed, derived class also changed

Interchangeable objects with polymorphism

- Treat object not as the specific type, but instead as the base type
- Allows to write code that doesn’t depend on specific types
- In the shape example, methods manipulate generic shapes without respect to whether they’re circles, squares, etc.

Polymorphism

A method in Java:

```java
void doStuff(Shape s) {
    s.erase();
    // ...
    s.draw();
}
```

This method speaks to any **Shape**, so it is independent of the specific type of object

```java
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

calls to **doStuff** automatically work correctly, regardless of the exact type of the object.

**doStuff()** → **Circle** is being passed into a method that’s expecting a **Shape**.
Polymorphism

```
s.erase()
N...
s.draw();
```

Notice that it doesn't say "If you're a Circle, do this, if you're a Square, do that, etc."

Often, you don't want to create an object of the base class, only its upcast to an abstract keyword.