CS 565 September 1, 1997
Jens Palsberg Readings: Any good book on Java.

Lecture Note 2:

Java for Ace Programmers

Contents

1 A quick sip of Java **2**

1.1 The History of Java 3
1.2 Java's Type System 4
1.3 Java vs. C++ 6
1.4 Java is Buzzword Compliant 8

2 The Java Language **9**

2.1 "Hello World" in Java 9
2.2 Interfaces in Java 10
2.3 Objects and Classes in Java 16
2.4 Inheritance and Super 17
2.5 Class Variables and Class Methods 18
2.6 Constructors and Finalizers 19
2.7 Visibility Modifiers in Java 20
2.8 Packages in Java 21
2.9 Exception Handling in Java 24

1 A quick sip of Java

Java is a new programming language.

$$\text{Java} \quad = \quad \text{C++ ++ −−}$$

If you know C++, good!
If you don't, even better!
Java is a better programming language than C++.

My personal perspective: they took C++, removed the stuff I am uncomfortable with, and put in some stuff I like.

1.1 The History of Java

- Java was designed in the 1990s at Sun Microsystems.
- The design of Java was initiated by Bill Joy and completed by Jim Gosling.
- Java was first intended for the smart consumer electronics market. In 1993 the design project was redirected to address the settop box market. In 1994 the Web came along, and the Java designers realized that Java was a good fit for Web programming.
- Java is popular among programming language researchers. At the conference on Principles on Programming Languages in January 1997, at least two papers are on Java.
- Guy Steele has done a good job on writing the specification of Java.
- Java runs on many platforms.
- Java is compiled to byte-code. The byte-code is interpreted by a byte-code interpreter. Thus, to port Java programs to a new platform, simply port the interpreter!

1.2 Java's Type System

$$
\begin{array}{rcl}
\mathsf{Type} & ::= & \mathsf{PrimitiveType} \;|\; \mathsf{ClassName} \;|\; \mathsf{InterfaceName} \\
 & | & \mathsf{Type}\,[\,] \qquad \text{;; array type}
\end{array}
$$

$$
\mathsf{ReturnType} \;::=\; \mathsf{Type} \;|\; \text{void}
$$

$$
\begin{array}{l}
\mathsf{PrimitiveType} \\
\qquad \begin{array}{cll}
::= & \text{boolean} & \text{;; 1-bit} \\
| & \text{char} & \text{;; 2 unsigned bytes, Unicode} \\
| & \text{byte} & \text{;; 1 signed byte (two's complement)} \\
| & \text{short} & \text{;; 2 bytes signed (two's complement)} \\
| & \text{int} & \text{;; 4 bytes signed (two's complement)} \\
| & \text{long} & \text{;; 8 bytes signed (two's complement)} \\
| & \text{float} & \text{;; 4 bytes, IEEE 754} \\
| & \text{double} & \text{;; 8 bytes, IEEE 754}
\end{array}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{ClassName} & ::= & \text{Identifier} \\
\mathsf{InterfaceName} & ::= & \text{Identifier}
\end{array}
$$

- Strings and arrays are objects.
- Casting: only between numeric types, between subclasses and superclasses, and between interfaces.
- Type correctness is checked at compile-time.

Java's Type System

The primitive types form a partial order which is a disjoint union of two trees:

$$\begin{tikzcd} & \text{byte} \arrow[r] & \text{short} \arrow[dr] & & & & \\ \text{boolean} & & & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\ & & \text{char} \arrow[ur] & & & & \end{tikzcd}$$

Subtyping in Java is a partial order of the types which extends the order of the primitive types in two ways:

1. if a class D explicitly extends a class C, then the classname type D is a subtype of the classname type C, and
2. if an interface J explicitly extends I, then the type J is a subtype of the type I.

Thus, an interface can be a subtype of an interface with fewer fields, but only if this is explicitly declared.

Interfaces can be mutually recursive, although an interface is not identified with its infinite unfolding.

1.3 Java vs. C++

Here is a list of C++ features which are *not* in Java:

- multiple inheritance
- operator overloading
- templates
- pointers
- undefined or architecture dependent constructs
- goto
- #define
- typedef
- enum
- union
- struct
- global variables (every variable is inside a class)
- global functions (every function must be a method)
- friend functions (in a package, “we are all friends”)
- virtual functions (all methods are by default virtual).

Java vs. C++, continued

Here is a list of Java features which are *not* in C++:

- interfaces
- packages
- multithreading
- garbage collection
- array-bounds checks

The basic control structures of Java have C syntax:

- if, else, for, while, do while, switch

There is exception handling in Java (a la C++).

The expression syntax of Java is essentially that of C.

Parameter passing:

- Objects, including strings and arrays, are passed by reference.
- Data of primitive type are passed by value.

1.4 Java is Buzzword Compliant

In “The Java Language: A White Paper,” Sun describes Java as follows:

“Java: A

- simple,
- object-oriented,
- distributed,
- interpreted,
- robust,
- secure,
- architecture neutral,
- portable,
- high-performance,
- multithreaded, and
- dynamic language.”

2 The Java Language

2.1 "Hello World" in Java

```
public class HelloWorld {

   public static void main(String[] args) {
       System.out.println("Hello world!");
   }
}
```

- Store the program in a file with the same name as the public class plus a `.java` extension.
- Compile with `javac`. (It is in `/usr/local/java/`.)
- Example: `javac HelloWorld.java`.
  This produces the file `HelloWorld.class`.
- To execute, run the interpreter `java` (it is also in `/usr/local/java/`).
- Example: `java HelloWorld`.
  This prints `Hello World!`.

2.2 Interfaces in Java

- An interface can contain
  - method headers, and
  - constants (static final).
- A class can implement an interface.
- An interface can extend other interfaces.
- Subtyping is by name, not structural.
- No overriding of method headers.

Interface Example

```
interface Driveable {
    boolean startEngine();
    void stopEngine();
    float accelerate( float acc );
    boolean turn( Direction dir );
}
class Automobile implements Driveable() {
    ...
    boolean startEngine() {
        if ( notTooCold )
           engineRunning = true;
        ...
    }
    void stopEngine() {
        engineRunning = false
    }
    float accelerate( float acc ) {
        ...
    }
    boolean turn( Direction dir ) {
        ...
    }
}
```

Interface Example, continued

```
Automobile auto = new Automobile();
Lawnmover mower = new Lawnmover();
Driveable vehicle;

vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();
...
vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

Another Interface Example: Finite Functions

Here is an interface for finite functions with strings as domain and integers as range.

```
interface FF {
  void empty_ff();
  int apply_ff(String key);
  void extend_ff(String key, int val);
}
```

Variables in Interfaces

```
interface Scaleable {
    static final int BIG = 0,
                  MEDIUM = 1,
                   SMALL = 2;

    void setScale( int size );
}

class Box implements Scaleable {
    void setScale( int size ) {
        switch( size ) {
            case BIG:
                ...
            case MEDIUM:
                 ...
            case SMALL:
                 ...
        }
    }
    ...
}
```

Interface Extension

```
interface I extends I1, I2, ..., In {...}
```

- Note: `I1, I2, ..., In` must declare disjoint sets of names.

```
interface DynamicallyScaleable
          extends Scaleable {

    void changeScale( int size );
}
```

2.3 Objects and Classes in Java

```
public class Circle {
    public double x,y;    // the center
    public double r;      // the radius
    public double circumference() {
        return 2 * 3.14159 * r;
    }
    public double area() {
        return 3.14159 * r * r;
    }
}
```

We can use this class as follows.

```
Circle c = new Circle();
```

Accessing instance variables:

```
Circle c = new Circle();
c.x = 2.0; c.y = 2.0; c.r = 1.0;
```

Invoking methods:

```
double a;
a = c.area();
```

2.4 Inheritance and Super

- Single inheritance!
- One can override methods and shadow variables in superclasses. The old version can be accessed with `super`.

```
class Animal {
    ...
    void eat( FoodItem f ) {
    }
    ...
}

class Carnivore extends Animal {
    ...
    void eat( FoodItem f ) {
        if ( f instanceof Plant )
            // Complain profusely
        else
            super.eat( f );
    }
    ...
}
```

2.5 Class Variables and Class Methods

- A class variable is declared with the keyword `static`.
- A class variable is "global" for all instances of the class.
- Same story for class methods.

```
public class Circle {
    static int num_circles = 0;
             // class variable
    public double x,y,r;
             // instance variables
    ...
}
```

2.6 Constructors and Finalizers

Constructors:

- To create an object of class `Circle`, we can use the default constructor and write:

  ```
  new Circle();
  ```

- But we can also program constructors ourselves.
- Every constructor must have the name of the class.
- No return type can be specified, the return type is implicitly the class itself.

Finalizers:

- If an object contains a `finalize()` method, then it will be invoked just before the object is garbage collected.

2.7 Visibility Modifiers in Java

Modifier	**Visibility**
`private`	None
`private protected`	Subclasses only,
	inside or outside the package
none (default)	Classes in the package
`protected`	Classes in package and subclasses
	inside or outside the package
`public`	All classes

One can *increase* visibility in a subclass. (But one cannot decrease visibility.)

2.8 Packages in Java

- A package is a module. It creates a scope.
- The space of packagenames is flat.
- A java program can be divided into files. A file is a sequence of class and interface definitions. One can start a file with

  ```
  package Id
  ```

  This declares that the content of the file belongs to the package `Id`.
- If one omits the package declaration, then the contents becomes part of a default unnamed package.

The Import Statement

- One can write abbreviations for class and interface names with the `import` statement.

  ```
  package mypackage;

  import mytools.text.TextEditor;

  class MyClass {
      TextEditor editor;
      ...
  }
  ```

  The notation with "." is not about nesting.

The Import Statement, continued

- There are three forms of the `import` statement.

```
import packageId;
import packageId.classId;
import packageId.*;
```

- The first one allows the specified package to be known by the name of its last component (this is where "`.`" plays a role). For example,

```
import java.awt.image;
```

  allows the class `java.awt.image.ImageFilter` to be called `image.ImageFilter`.

- The second one allows the specified class in the specified package to be known by its class name alone. For example,

```
import java.util.Hashtable;
```

  allows the class `java.util.Hashtable` to be called `Hashtable`.

- The third form makes all classes in a package available by the class name.

2.9 Exception Handling in Java

Exceptions = instances of the class `java.lang.Exception`.

An exception is thrown from one point in the code and caught by another.

```
try {
    readFromFile("foo");
    ...
}
catch ( Exception e ) {
    // Handle error
    System.out.println(
      "Exception while reading file: " + e );
    ...
}
```

Exception Handling in Java

A try statement can have multiple catch clauses that specify different specific types (subclasses) of `Exception`. The catch clauses are evaluated in order, and the first possible (assignable) match is taken. At most one catch clause is executed.

```
try {
    readFromFile("foo");
    ...
}
catch ( FileNotFoundException e ) {
    // Handle file not found
    ...
}
catch ( IOException e ) {
    // Handle read error
    ...
}
catch ( Exception e ) {
    // Handle all other errors
    ...
}
```

Exception Handling in Java

Uncaught exceptions are, in turn, thrown from the method in which it appeared.

```
void sendRequest() throws IOException {
  write( ... ); // can throw an IOException
}
```

If an exception never gets caught, the program terminates with a run-time error message.

We can throw our own exceptions:

```
throw new MyException();
```

Exception Handling in Java

Exceptions are divided into two categories:

General exceptions Exceptions a reasonable application should try to handle gracefully. Examples: Missing files, unavailable hosts.

Run-time exceptions Exceptions from which we would not normally expect our software to try to recover. Examples: Out of memory, array index out of bounds.

It is not a compile-time error to ignore the possibility of subclasses of `RuntimeException` to be thrown; methods do not have to declare that they can throw them.

By the way, multithreading in Java is not covered here; take Parallel Programming.