Java

from the Perspective of a Long-Time C/C++ Hack

Bruno R. Preiss
Electrical and Computer Engineering
University of Waterloo

http://www.pads.uwaterloo.ca/Bruno.Preiss/talks/1997/padsgroup/slides.ps

Even Experts Get it Wrong Occasionally

From [**?**], page 1:

```
class HelloWorld {
    public static void main(String[] args) {
        Sytem.out.println("Hello, world");
    }
}
```

Java: "A Blue Collar Language" [?]

- object-oriented flavour
  - derived from Simula, C/C++, Objective C, Cedar/Mesa, Modula and Smalltalk
- completely specified expression semantics
  - what "2+2" means
- has a "systems programming flavour"
  - whatever that means
- distributed nature
  - no boundaries between machines

Primitive Types

`boolean` `true` or `false`

`byte` -128 to $+127$

`short` $-32\,768$ to $+32\,767$

`int` $-2\,147\,483\,648$ to $+2\,147\,483\,647$

`long` $-9\,223\,372\,036\,854\,775\,808$ to $+9\,223\,372\,036\,854\,775\,807$

`char` `\u0000` to `\uffff`

`float` IEEE 754 single-precision (32-bit)

`double` IEEE 754 double-precision (64-bit)

- an `int` is an `int`!
- 16-bit `char`! (Unicode)
- no `signed` or `unsigned`
- `long` and `short` are types, not modifiers

Unicode is Great!

```
class Ω
{
    double α;
    double β;

    public double Γ ()
        { return α*α+β*β; }
}

Ω τ;
double z = τ.Γ();
```

User-defined Types

- no `typedef`
- no `struct`
- no `enum`
- no `union`
- Do it with `class` and `interface`

Variables, Objects and References

- two kinds of data values that can be stored in variables:
  - primitive types
  - reference types
- three kinds of reference types
  - class types
  - interface types
  - array types
- an object is a
  - class instance
  - array instance

No Addresses

- no pointers
- no address-valued expressions
- no unary `&` operator
- no unary `*` operator
- no address arithmetic

Parameter Passing

- primitive types: call-by-value
  - formal parameter is initialized using the value of the actual argument expression

- reference types: call-by-reference
  - formal parameter is initialized as a reference to the same object to which the actual parameter refers
  - actual parameter must be a reference variable (not an expression)

Example of a Class

```
public class Foo
{
    private int x;
    protected int y;
    public int z;
    public static int a;
    public static final int b = 6;

    public Foo ()
        { ... }
    public void F ()
        { ... }
    public static void G ()
        { ... }
}
```

More About References

- all objects must be created explicitly

```
Foo f;
Foo g = new Foo ();
f = g;
g = null;
f = null;
```

- appropriate constructor is automatically invoked

- there are no destructors

- there is no `delete`

- storage is recovered by garbage collection

Java Programs

- Everything is declared inside a class (or interface).
- No `.h` (header) files
- No forward declarations (allowed or required)

Typical Programming Environment

- only one `public` class per file

- the name of the class matches the name of the file: public class `Foo` must be defined in `Foo.java`

- compiler translates Java to "bytecode" in a `.class` file

- a separate "bytecode" file is produced for each class/interface
  - one source file can generate several `.class` files

Derivation and Inheritance

```
public class B
{
    protected int x;
    protected int y;

    public void F ()
        { ... }
    public void G ()
        { ... }
}

public class D extends B
{
    protected int y;
    protected int z;

    public void G ()
        { ... }
    public void H ()
        { ... }
}
```

Derivation and Inheritance

- only single-inheritance (derivation) allowed
- all (non-static) member functions are *virtual*
- member variable accesses and member function calls are resolved in fundamentally different ways
- a class that does not explicitly extend another implicitly extends class `Object`

Object Class

```
public class Object
{
    public final Class getClass () { ... }
    public final String toString () { ... }
    public boolean equals (Object object) { ... }
    public int hashCode () { ... }
    protected Object clone ()
        throws CloneNotSupportedException { ... }
    public final void wait ()
        throws IllegalMonitorStateException { ... }
    public final void wait (long millis)
        throws IllegalMonitorStateException,
        InterruptedException { ... }
    public final void wait (long millis, long nanos)
        throws IllegalMonitorStateException,
        InterruptedException { ... }
    public final void notify ()
        throws IllegalMonitorStateException { ... }
    public final void notifyAll ()
        throws IllegalMonitorStateException { ... }
    protected void finalize ()
        throws Throwable { ... }

}
```

Classes

- access control

`public` anyone can instantiate it

default only instantiatable in its package

- modifiers

`abstract` has abstract (pure virtual) functions

`final` cannot be derived from

default instantiatable and derivable

An Abstract Class

```
public abstract class Foo
{
    protected int x;
    protected int y;

    public void F ()
        { ... }
    public abstract void G ();
}
```

Member Variables—access control

`public` anyone can access

`protected` access only from the same package or a derived class

`private` access from class only

default access only from the same package

Member Variables—modifiers

`static` only one instance

`final` a constant (like `const`)

`transient` not persistent (a hook)

`volatile` may change by itself

default just a plain old member variable

Member Functions—access control

`public` anyone can access

`protected` access only from the same package or a derived class

`private` access from class only

default access only from the same package

Member Functions—modifiers

`abstract` a pure virtual function

`static` called without using dot operator

`final` cannot be overridden in a derived class

`native` implementation not in Java

`synchronized` a monitor entry routine

default just a plain old member function

What is `this`?

- `this` a reference to self
- not a pointer to self
- `this (...)` is a call to self's constructor
  - can only be called from another constructor
  - must be the first statement

Who is `super`?

- `super` is a reference to the part of self that is an instance of the superclass (the class from which self's class is derived)

- a way to get at inherited functionality that may have been overridden in self's class
  - `super.x` (a variable)
  - `super.Foo ()` (a function)

- `super (...)` is a call to a constructor of the superclass
  - can only be called from another constructor
  - must be the first statement
  - if you leave it out, `super ()` is called by default
- `super` makes sense because of single-inheritance

What is an interface?

```
public interface Vocal
{
    public void Yell ();
    public void Scream ();
}

public interface Physical
    extends SomethingElse
{
    public void Kick ();
    public void Punch ();
}

public class Foo
    extends Bar
    implements Vocal, Physical
{
    ...
}
```

The Cool Thing About Interfaces

- program for the capabilities of a class not for its implementation

```
public interface Comparable
{
    public boolean LT (Comparable object);
}

public interface Sorter
{
    public void Sort (Comparable[] array);
}

public class BubbleSorter
    implements Sorter
{
    ...
}
```

```
class Foo
{
    public static void DoSomething (Sorter s)
    {
        ...
        s.Sort (array);
        ...
    }

    public static void main (String[] args)
    {
        DoSomething (new BubbleSorter ());
        DoSomething (new QuickSorter ());
    }
}
```

Cool Features

- dynamic, incremental linking/loading of applications
  - recompile a module and run the app (no link step)
  - possible to link a module into a running app
  - possible to determine during run time what module to load and link
  - possible to link physically remote modules into an app
  - possible to turn a sequence of bytes into bytecodes that can be linked into the app

The Class of a Class

```
public final class Class
{
    public String toString () { ... };
    public String getName () { ... };
    public boolean isInterface () { ... };
    public Class getSuperclass () { ... };
    public Class[] getInterfaces () { ... };
    public Object newInstance ()
        throws InstantiationException,
        IllegalAccessException
        { ... };
    public ClassLoader getClassLoader () { ... };
    public static Class forName (String className)
        throws ClassNotFoundException { ... };
}
```

An Example

```
class Example
{
    public static void main (String[] args)
    {
        for (int i = 0; i < args .length; ++i)
        {
            Class cls = Class.forName (args [i]);
            Object obj = cls.newInstance ();
            System.out.println (obj);
        }
    }
}
```

Exceptions

- `Exception` class is superclass of all exceptions
- `throw` statements
- `catch` blocks
- `throws` declarations
- checked vs. unchecked exceptions
  - `RuntimeException` is a subclass of `Exception`
  - all subclasses of `Exception` except subclasses of `RuntimeException` are checked at compile time

Other things

- Object Classes `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Char`, `Float` and `Double`

- a standard `String` class

No Operator Overloading (Almost)

- The plus operator `+` concatenates `String` instances
  - if one argument is a `String` and the other is an object that is not a string that object is converted to a string by calling its `toString()` member function
  - e.g., `System.out.println ("x=" + x);`

Arrays are Objects

- syntax is fixed:

```
int[] x;
int[][] y;
String[] args;
```

- must be allocated explicitly

```
x = new int[10];
y = new int[100][100];
args = new String[32];
```

- Note `args` $\neq$ `null` but `args[0]` $=$ `null`

- Also `char[]` $\neq$ `String`

Concurrency

- built-in concurrency support
  - threads and a standard `Thread` class
  - all objects are monitors
  - `synchronized` keyword used to ensure mutually exclusive access to critical sections
  - all objects can serve as condition variables via `wait` and `notify`

Packages

- built-in support for control of namespace pollution

```
package ThisIsMyNamespace;
import java.io.*;
import SomeOtherGuys.Bar;

class Foo
{
    ...
};
```

- `import` statement used to get at contents of other package
  - import piecemeal
  - or holus bolus