Architecture of Java Systems

Java's goals:

- Platform-independence
- Security
- Mobility

all in the context of an OO language similar to C++.

Architectural components:

- Java source language
- Java Libraries API
- Java Virtual Machine (JVM) specification
- Java Class File specification

Many kinds of JVM implementations are possible:

- interpreters
- compilers
- hardware

in many contexts:

- in browser ("applets")
- stand-alone ("applications")
- embedded

Classes

All Java data and function definitions are organized into **classes**.

Each class is a subclass of exactly one other class, except for the predefined class `Object`, which serves as the root of the object hierarchy.

An **instance** of a class is a record-like value with associated **members**: **fields** (variables) and **methods** (functions). As usual, a class inherits the fields and methods of its parent class and of its interfaces, and can override the methods of its parent. Instance methods take the instance itself as an implicit parameter, which can be referred to via `this`.

Abstract classes have specifications but no associated code. In addition, classes can **implement** one or more abstract **interfaces** (with methods but no mutable fields), effectively allowing multiple inheritance of specifications.

Classes have special **constructor** methods for creating new instances; as in C++, the class name serves as the constructor name.

Important Java Language Features

Essentially a simplified version of C++

- Class-based object-oriented language.
- Type-safe.
- Garbage collected.
- No explicit pointers.
- Single inheritance of classes ("implementation inheritance").
- Inheritance from (multiple) interfaces ("specification inheritance").
- No templates or generics.
- Built-in arrays and strings.
- Multi-level break; no `goto` statement.
- Exceptions.
- Concurrent threads.
- Dynamic linking.

More on Classes

Classes can also have **static** (per-class) variables and methods; this is important, since every function has to be a method within some class or other.

Method and constructor names can be **overloaded**, so long as they can be distinguished by the number and types of their parameters (their **signatures**).

Class and member definitions have a variety of **modifiers** controlling things like access. For example, a class can be declared **final** to indicate that it cannot be sub-classed; a member can be declared **private** to indicate that it can only be accessed by method code within its own class.

Example:

```
class MyClass {
  int i;
  public MyClass() {i = 0;}
  public MyClass(int k) {i = k;}
  public int addtome(int i) {return this.i + i;}
  public static int add(Myclass c,int j)
      {return c.i + j;}
}
MyClass m1 = new MyClass();
MyClass m2 = new MyClass(1);
int j1 = m1.addtome(2);
int j2 = MyClass.add(m2,3);
```

Packages

At the very top level, class definitions are organized into **packages**, each with a separate internal name space.

Packages consist of class and interface definitions, and possibly subpackages.

In particular, the standard API is presented in the following packages (which always exist):

```
java.lang
java.util
java.io
```

Simple applications can just define clases in the "unnamed" package.

Package names are prepended to class or interface names to produce **fully qualified names** which are what appear in class files.

Types

Each class declaration essentially defines a type (called a **reference type**, because class instances are handled by reference).

There are also built-in **primitive types** (boolean, char, byte, short, int, long, float, double) whose values are **not** class instances; there are standard library wrapper classes for these types.

Variables (class or instance fields, parameters, or local variables within methods) and expressions are always **typed** with a primitive, class or interface name. The basic typing rule is that it is valid to assign an expression of type `C` to a variable of type `D` if and only if `C` inherits from `D`.

It is also possible to explicitly cast an expression of type `D` to some subtype `C`; this is only legal if the expression actually evaluates at runtime to an object of type `C`, which requires a runtime check.

Objects = Class Instances

Class instances are essentially like records. They are always heap-allocated and manipulated via an (implicit) pointer.

Java doesn't have explicit pointers.

Instances are created by calling a constructor. Instances are removed by an automatic **garbage collector** at some point after they cease to be referenced by the program. There is no explicit deallocation.

The `Object` class has a **finalize** method which is guaranteed to be called before an object instance is garbage-collected; this can be overridden

Types Example

```
class C extends D {}
class B extends D {}

C x;
D y;
E z;

y = x; // ok
y = z; // ok
x = y; // not ok; compile time error
x = (C)y; // ok with cast, but may throw
          // ClassCastException at runtime
y = z; // never ok; compile time error
```

Any reference variable can take on the special value `null`. Attempts to dereference a null value are also checked at runtime. If a runtime check fails, an exception is raised.

Arrays and Strings

Java **arrays** are a special sort of object (belonging to the root class `Object`), which are created dynamically.

Arrays can contain either primitive or reference types.

Arrays are single-dimensional; multi-dimensional arrays can be generated as arrays of arrays.

All references to arrays are subject to runtime **bounds checking**. There is no way to point directly into an array.

Java **strings** are **not** just arrays of characters; `String` is a special built-in class. Strings are **immutable**; their contents never change.

Java has special support for:

- defining string literals using the ordinary `""` mechanism;
- appending strings, using the built-in `+` operator;
- converting arbitrary types to string form.

There is also a class of mutable `StringBuffer`s.

Strings are encoded in Unicode.

Exceptions

Exceptions indicate abnormal conditions in code.

Java exceptions can be **thrown** by the internal runtime system (e.g., by memory access violations, division by zero, etc.) or explicitly by the user program using the `throw` statement.

The exception itself is represented as an object belong to (a subclass of) class `Throwable`.

Exceptions can be **caught** by enclosing the statements whose execution led to the exception in a `try...catch` construct. Exception handlers can be restricted to handle only certain subclasses of exceptions. When an exception is raised, control passes to the nearest **dynamically** enclosing handler.

Multi-level break

Java has **no** general-purpose `goto` statement.

But it provides a restricted `goto` by allowing labels on loops, which can be referenced by `break` or `continue` statements.

Example:

```
float [][] Matrix;
boolean found = false;

search:
  for (y = 0; y < Matrix.length; y++) {
    for (x = 0; x < Matrix[y].length; x++) {
      if (Matrix[x][y] == 0.0) {
        found = true;
        break search;
      }
    }
  }
  if (found)
    ...
  else
    ...
```

Exceptions Example

```
// class Exception extends Throwable {}
class NotFoundException extends Exception {}

class Dictionary {
  public int lookup(String key) {
     // if found then
          return value;
     // else
          throw new NotFoundException();
  }
  ...
}

void lookshow(Dictionary dict, String key) {
  try {
     System.out.println(dictionary.lookup(key))
  } catch (NotFoundException e) {
     System.out.println("Not found")
  }
}
```

Finally Clauses

The `finally` construct can be used to clean up at the end of a block that might be exited abnormally.

Example:

```
void openlookshow(String dict, String key) {
  Dictionary d = Dictionary.open(dict);
  try {
     System.out.println(d.lookup(key))
  } finally {
     d.close()
  }
}
```

This is essentially equivalent to:

```
void openlookshow(String dict, String key) {
  Dictionary d = Dictionary.open(dict);
  try {
     System.out.println(d.lookup(key));
     d.close()
  } catch (Throwable e) {
     d.close();
     raise e
  }
}
```

But note the duplicated code in this version.

Dynamic linking

The language specification includes detailed rules for how inter-class linking occurs. These are described with reference to the Java Virtual Machine (JVM) and to a set of standard functions in the `java.lang` API.

In particular, the specification insists that binaries generated by Java compilers use **symbolic names** for all references from the code of one class to the members of other classes and interfaces. These names are not **resolved** until the external member is actually needed at runtime. (More precisely, **errors** arising from attempts to resolve symbolic names must not be reflected until then.)

Loading and resolution are provided by a runtime system class called `ClassLoader`. The default, built-in class loader obtains binaries from the local host system (in some implementation-dependent way). User code may provide subclasses of `ClassLoader` that obtain binaries in some other way. For example, net browsers might get binaries over the network using a specialized `ClassLoader`.

Once a class has been (explicitly) loaded with a specialized instance of `ClassLoader`, any classes references are automatically loaded using the same `ClassLoader` instance. Thus, `ClassLoader` instances effectively provide a top-level partitioning of the symbolic name space.

Threads

Java has integral support for multi-threaded programming using per-object locks. Locks can be handled using **monitor** protocols (executing an object method requires holding the object's lock) or explicitly.

Threads are initiated by invoking the `start` method of an object inheriting from class `Thread`; this in turn invokes the object's `run` method.

Monitor-like semantics for a method are specified by using the modifier `synchronized` on the method definition.

Locks can be obtained explicitly by using the `synchronized` statement.

The `wait` and `notify` statements are used to to coordinate access to shared objects among threads.

To obtain sensible semantics for non-synchronized access to shared variables, those variables may be declared `volatile`.

More Dynamic linking

Specialized `ClassLoader` instances can be used to load new code into a running Java program and jump into it. However, it is apparently not possible to replace existing class definitions in a running program.

Also, Java 1.1 contains **reflection** facilities for examining class definitions at runtime.

Despite these facilities, Java provides only imprecise control over **versioning** of components, because symbolic names don't contain any **global** identification information, and the precise time when class loading occurs is still rather flexible.

Java API

The language specification insists on the existence of a basic set of library facilities, which must be present (with the specified semantics) in every Java runtime environment.

Many of these may be implemented in Java, but some require access to underlying facilities of the host system (e.g., for IO) or of the java execution engine itself (e.g., class loading).

`java.lang` contains: `Object` – the root of the object hierarchy
wrapper classes for primitive types
`String` and `StringBuffer`
`ClassLoader`
`Class` – represents class and interface defns at run-time
`Process`,`Runtime`,`System` – basic RTS stuff
`SecurityManager`
`Thread` and `ThreadGroup`
`Throwable` and its subclasses

`java.util` contains useful ADTs, from `Date` to `Dictionary`.

`java.io` is the basic I/O library.