JAVA

Java vs. Java

- Java Language Specification
  - http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
  - Java programming language is compiled into java byte code
- Java Virtual Machine Specification
  - VM behavior
  - Java class file format
  - http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Terms

- Java programming language is a *strongly typed* language
  - type of every variable and every expression is known at compile time
  - http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
- 2 kinds of types
  - primitive types (`boolean` and numeric types)
  - reference types (classes, interfaces, arrays)
- Classes
  - `java.lang.Object` is superclass of all **classes**
- Objects
  - building blocks of a Java-application
  - class instance or array

Java language features

- Single-inheritance of classes
- Support of multiple interfaces in a class
- Access control
  - private: only in defining class
  - protected: from subclasses
  - public: full accessible
  - final: no subclassing
  - http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#104285
- abstract classes can not instanciated
  - An abstract method has no implementation
- static methods callable without an instance

Private Restriction

```
public class A{
   private int x=1;
   public  int y=x+1;
}

public class B extends A {
   public int z = x;          // compile error
   public int w = y;
}
```

```
B.java:2: x has private access in A
    public int z = x;
                   ^
1 error
```


Protected Restriction

```
package a;
public class A{
   protected int x;
}
package a;
public class B extends a.A{
   int y=x+1;
   public void foo(a.A o){
      o.x++;
   }
}
```

- Access permited within the package containing the class

Protected Restriction

```
package a;
public class A{
   protected int x;
}
package b;
public class B extends a.A{
   protected int y=x+1;
   public void foo(a.A o){
      o.x++;      // compile error
   }
}
```


Final Restriction

```
public class A{
    final void do(){ }
}

public class B extends A{
   public void do() {}          // Error
   public void do(int i) {}     // Ok
}
```

Abstract class

```
public abstract class A{
   public void foo(){ }
}

public class B{
   A a=new A();    // cannot be instantiated
}

public class C extends A{
    public void bar() {
       foo();                    //  Ok
    }
}
```


Abstract methods

```
public abstract class A{
   abstract void foo();
}

public class B extends A{
   public void foo() {}   //compile error
}
```

- Inherited abstract methods must be overriden

Exceptions

- Runtime problems are represented as Exception objects
- If an exception is thrown
  - Code stops immediately
  - Call stack „unwinds“ until the exception is caught
- „throws“ clause lists possible exceptions
  - Must be caught or thrown by the caller

Exceptions

```
public void foo() throws Exception {}

public void bar()  throws Exception {
   foo(); // compile error
}

public void foobar(){
   try{
     bar();    // compile error
   }catch(Exception e){}
}
```

Coding Conventions

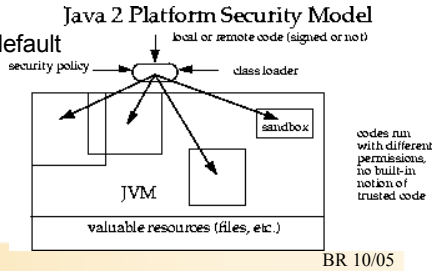
- Why? (Sun Java Coding Conventions)
  - 80% of the lifetime cost of a piece of software goes to maintenance.
  - Hardly any software is maintained for its whole life by the original author.
  - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
  - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
- http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Coding Conventions

- Public classes/interfaces must be declared in a file with the same name
- Class names should be nouns, in mixed case with the first letter of each internal word capitalized
  - e.g. MyFooBar
- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized
  - e.g. doStuff
- Javadoc for source code documentation

Application vs. Applet

- Application
  - Starts with "**public static void** main(String[] args)" of first called class
  - Local code runs with "liberal" restrictions by default (JDK 1.2)
- Applet
  - Runs in a sandbox
  - No access to local resources by default
  - Extends `java.applet.Applet`



BR 10/05

Packages

- Organize programs as sets of packages
  - prevent name conflicts
  - hierarchical naming structure
- A package can be stored in a file system or in a database
- A package contains subpackages and compilation units
  - e.g. filesystem
  - packages are directories
  - compilations units are class-files
- Keywords: package & import

BR 10/05

Location

package java.lang;
public class Object { … }

- Path: java/lang/Object.class
  - relative to the Classpath
- Class name: java.lang.Object

Demo (create package, compile)

JAR Files

- Plattform-independent file format that aggregates many files to one
- Combined packages
- JAR-file specification
  - http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html
- ZIP-file with meta data
  - META-INF directory contains MANIFEST.MF
- Manifest
  - name: value pairs (inspired by RFC822)
- Reading JAR file content
  - java.util.jar.JarFile

JAR-Metadata

- Determine entrypoint of a JAR-file
  - Manifest m=JarFile.getManifest()
  - Attributes at=m.getMainAttributes()
  - at.getValue(Attributes.Name.MAIN_CLASS)
- Additional references
  - Attributes.Name.CLASS_PATH

ClassLoader

- Responsible for loading classes (`java.lang.Class`)
- Every class-object contains a reference to its ClassLoader (`Class.getClassLoader()`)
- Search order for classes
  - Bootstrab classes : rt.jar, i18n.jar
  - installed extensions: JAR-files in lib/ext of JRE directory (java.home)
  - ClassPath:
    - JVM-Property java.class.path,
    - Class-Path im Manifest von JAR-Dateien
    - CLASSPATH Umgebungsvariable

ClassLoader

- (Almost) all ClassLoaders hava a Parent-ClassLoader
  - Default is the System ClassLoader `getSystemClassLoader()`
  - can be overwritten: `ClassLoader(ClassLoader parent)`
- Bootstrap ClassLoader
  - basic runtime-classes
  - Extensions in lib/ext
  - `Class.getClassLoader()` returns null
- System ClassLoader
  - Default ClassLoader of applications and custom ClassLoader
  - e.g. `sun.misc.Launcher$AppClassLoader`
  - `getSystemClassLoader()` returns always System-ClassLoader

ClassLoader-API

- findClass(String name)
  - Loads a class
  - Full qualified names java.lang.Object
- defineClass(String name,byte[],int off, int len)
  - Converts an array of bytes into an instance of class java.lang.Class
- resolveClass(Class c)
  - Loads all referenced classes of the class object
  - Linking process
- loadClass(String name)
  - called by Class.forName()

ClassLoader

- Load a class with the System-ClassLoader
  - Class.forName(Name)
- Use a custom ClassLoader
  - Class.forName(Name,true,ClassLoader);
- ClassLoader.load(Name) search order
  - findLoadedClass(Name)
  - delegate search to parent ClassLoader.load()
  - findClass()

Reflection

- Classes and interfaces for obtaining reflective information about classes and objects
- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.

Reflection

- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

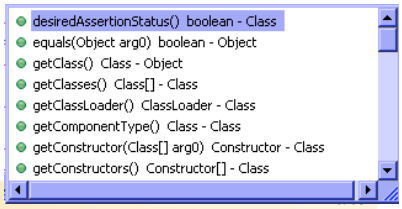
Reflection-API

java.lang.Class
- Class.getDeclaredMethods() , all implemented/declared methods
- Class.getConstructors()
- Class.getMethods() , only public methods
- Class.getFields()
- Class.getInterfaces()
- …

- java.lang.reflect
  - Array
  - Construtor
  - Field
  - Method
  - …

Demo – Discover Class Methods

- Get class information like in eclipse „Intellisense“
- Use Reflection API to get signatures of all methods
- Class.getDeclaredMethods() of all superclasses
- Modifier.isXXX to get access restrictions
- Method.getParameterTypes() returns array of Class objects
- Method.getReturnType()



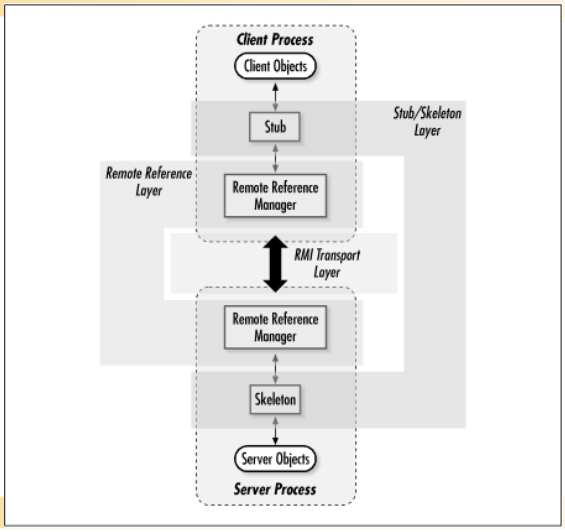
Reflection

- Call a method with Reflection-API
- Get Method object from the class object
  - Method m=Class.getMethod(„Name“,parametertypes)
- Create Class instance if required
  - Object obj=Class.newInstance() or
  - Object obj=Constructor(Object[] args)
- Method.invoke(Object obj, Object[] args)
  - Call invoke on the Method object
  - Static methods with obj=null

RMI

- Remote Method Invocation-API
- Introduced with JDK 1.1
- RMI Architecture and Functional Specification
  - http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html
- Distributed (java) object system
- Transparent access to remote Java objects
  - Stub/Skeleton objects

RMI-Architecture



„Java Enterprise in a Nutshell“

RMI-Layers

- Stub/Skeleton Layer
  - Interface that client and server applications objects use to interact with each other
- Remote Reference Layer
  - Creation and management of remote object references
  - JRMP (Java Remote Method Protocol)
  - RMI/IIOP (Internet Inter-ORB Protocol, CORBA)
- RMI Transport Layer
  - Binary protocol for remote object requests over the wire
  - TCP/IP

Un/Marshalling

- Marshalling
  - convert local objects into a portable format
- Unmarshalling
  - convert portable format into local objects
- Noneremote arguments are passed by copy
  - Must implement `java.io.Serializable` interface
  - Serialized byte stream is reconstitued into a copy of the local object
- Remote objects are passed by reference
  - Must implement `java.rmi.Remote` interface
  - Reference is used as marshalled data
- Otherwise
  - `java.rmi.MarshallException` is thrown

Stub/Skeleton

- Stub: Clientside proxy for the remote object
  - Marshalling parameters
  - Initiate the call
  - Unmarshall the returned values
- Skeleteon: Serverside proxy for the hosted object
  - Unmarshall arguments for the call
  - Making the up-call to the local object
  - Marshall the return value(s) of the call
- rmic
  - generates Stub/Skeleton classes from the remote object implementation
  - `<RemoteObjectImplementation>_Stub.class`
  - `<RemoteObjectImplementation>_Skel.class`

Implementation

- RMI-objects are accessible through interfaces
  - Must implement the `java.rmi.Remote` interface
  - All methods of an RMI-interface must throw a `java.rmi.RemoteException`
  - Interface:

```
public interface Hello extends java.rmi.Remote{
  public String sayHello(String msg) throws
                          java.rmi.RemoteException;
}
```

Implementation (II)

- Must implement RMI-Interface

```
import java.rmi.server.UnicastRemoteServer;
import java.rmi.RemoteException;

public class HelloImpl extends UnicastRemoteServer
implements Hello{
  public HelloImpl() throws RemoteException {
          super();
  }
  public String sayHello(String msg) throws
   RemoteException{
            return _Hello _ + msg;
  }
}
```

- Generate Stub/Skeleton: `rmic HelloImpl`

How to find a RMI-Object?

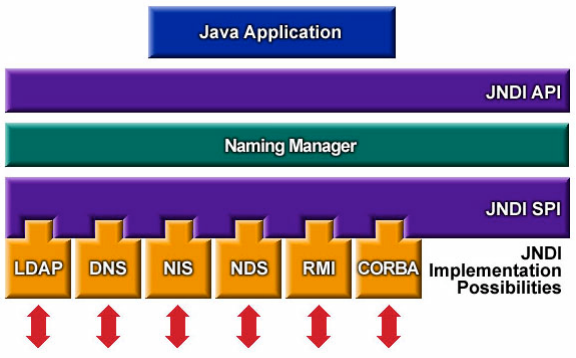
- RMI registry Naming service
  - rmiregistry tool
  - default port 1099
  - Or `LocateRegistry.createRegistry(1099)`
- On the server
  - Remote object is identified with a name
  - `Naming.rebind(_Name_ ,Remoteobj);`
- At the Client
  - `Naming.lookup(_Name_)`
  - IHelloWorldrmiobj=(IHelloWorld)Naming.lookup(_URL_to_registry/ “+ “Name “);
- Access remote registry
  - `LocateRegistry.getRegistry(URL)`

Other features

- Dynamic loading of sub and skeleton classes
  - RMIClassLoader loads classes from a web server
  - `java -Djava.rmi.server.codebase=http://…`
  - Requires a SecurityManager e.g. RMISecurityManager
  - `-Djava.security.policy=policyfile`
- Remote Object Activation
  - Extend from `java.rmi.activation.Activable`
- rmid
  - Activation system daemon that allows objects to be registered and activated in JVM on demand

JNDI

- Java Naming and Directory Interface



„Sun JNDI tutorial“

JNDI

- Assosiate names with objects and provide access
- Integral part of J2EE standard framework
- Service Provider Interface (SPI) for various services
  - LDAP
  - DNS
  - RMI
  - …
- A naming-service is connected with a context
  - set of name-to-object bindings

JNDI-API

- javax.naming
- javax.naming.direcotry
- Context with Name-Value pairs
- Factory class
  - Context.INITIAL_CONTEXT_FACTORY
- Server Url
  - Context.PROVIDER_URL

JNDI example

- Access RMI registry

```
Properties p=new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
  "com.sun.jndi.rmi.registry.RegistryContextFactory");
p.put(Context.PROVIDER_URL,"rmi://localhost");
Context ctx=new InitialContext(p);

Object o=ctx.lookup("hello");
```


JDBC

- „Java Database Connectivity“
- platform-neutral SQL interface to database systems
- Database dependend drivers
  - load with Class.forName()
  - e.g. Class.forName(„sun.jdbc.odbc.JdbcOdbcDriver“)
- Establish a database connection
  - DriverManager.getConnection(URL,…)
  - URL is driver dependent: `jdbc:driver:databasename`
  - e.g. `java:oracle:thin:@site:port:database`
  - DriverManager queries all loaded database driver
  - Driver must recognize its own URL

JDBC

- Packages
  - java.sql
  - javax.sql
- Create SQL-statements at the Connection object

Connection con=DriverManager.getConnection(„jdbc:….“);
Statement stmt=con.createStatement();

- Result rows as ResultSet

  ResultSet rs=stmt.executeQuery(„Select * from foo“);
- API allows
  - Prepared statement, parameterized query
  - Batch jobs
  - Transactions
  - …

References

- Sun J2SDK 1.4.2 documentation & Guides
- http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
- http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html
- http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html
- Java Enterprise in a Nutshell