Data Externalisation

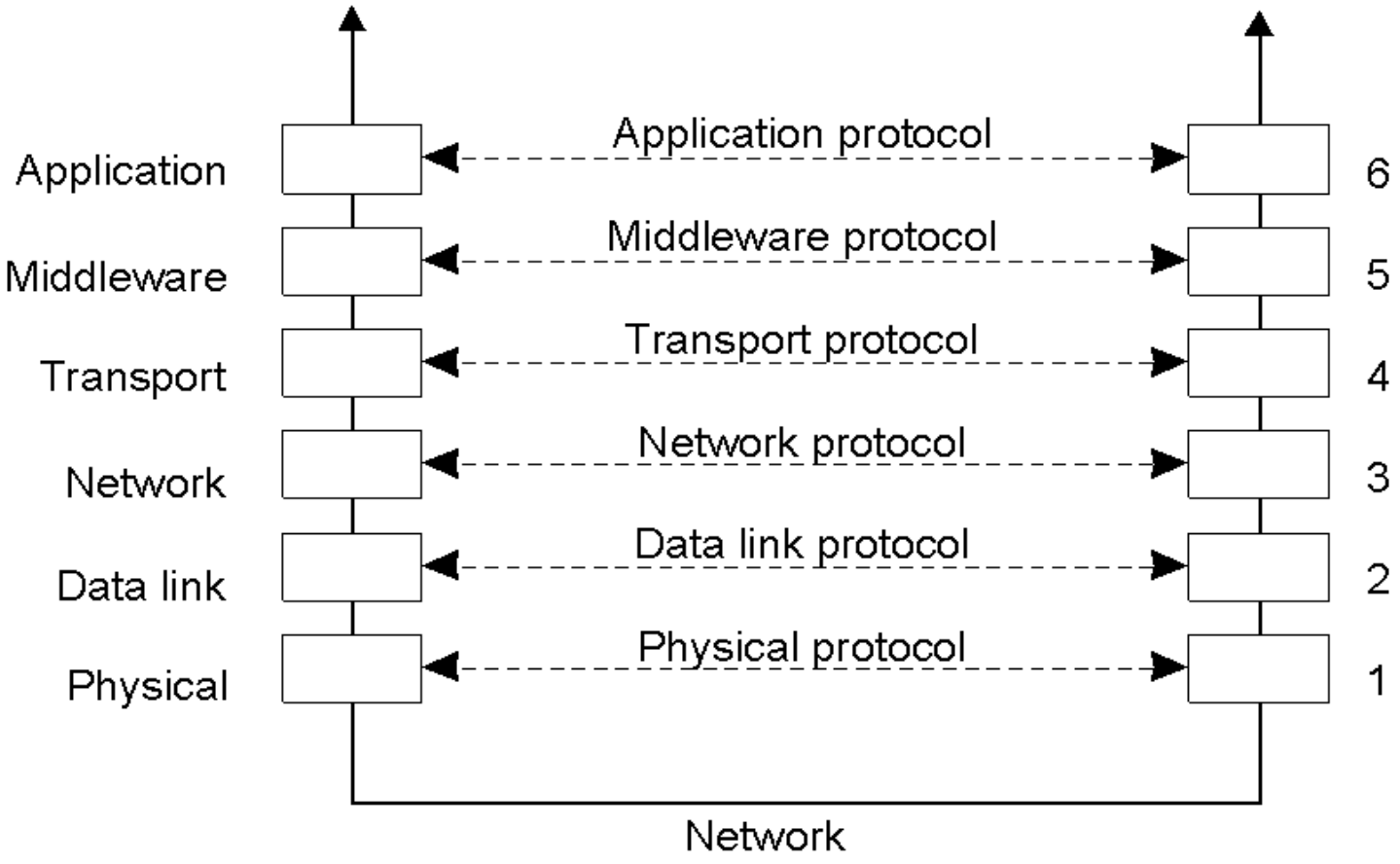
❖ ***Topics***

- Middleware
- CORBA CDR
- XML Object Binding
- JSON
- Java Serialization
- JNDI and Naming Services

Network Prog & Middleware

- A ***software layer*** that lies between the operating system and the applications on each site of the system.
- A software layer that:
  - masks the ***heterogeneity*** of systems
  - provides a ***convenient programming*** abstraction
  - provides protocols for providing general-purpose services to more specific applications, eg.
    - authentication protocols
    - authorization protocols
    - distributed commit protocols
    - distributed locking protocols
    - high-level communication protocols
      - remote procedure calls (RPC)
      - remote method invocation (RMI)
- ***Used to "glue" heterogeneous remote systems together.***

Middleware and the OSI Model



Middleware Prog Models

- **Remote Calls**
  - Remote Procedure Calls (RPC)
  - Distributed objects and Remote Method Invocation (RMI)
    - eg. Java RMI
- **Common Object Request Broker Architecture (CORBA)**
  - Cross-language RMI.
- **Web Services**
  - XML-based service discovery, binding and invocation.
- Other programming models:
  - remote event notification.
  - remote SQL access.
  - distributed transaction processing.
- Usually provided by an ***application server*** in an n-tier client-server environment.

External Data Representation

- ***Interprocess communication*** is the passing of messages from one process to another.
  - May be on the same host, may not.
- Challenge to devise an architecture that enables applications in heretogenous environments to communicate and work together.
  - ***Homogenous***: relatively straight forward – COM/DOM on Windows, Pipes in Unix, RMI in Java.
- A platform and language neutral, vendor-agnostic framework for IPC has been the philosopher's stone of distributed computing.
- *We will look at the following standards for data representation:*
  - CORBA Common Data Representation (CDR)
  - Java Serilization
  - XML and XML Object Binding
- These data representation formats form the basis of the technologies which we will discuss later.

External Data Representation

- Why can binary invocations work in different environments?
  - ***Byte ordering variants*** for the ordering of integers – Big-endian (MSD comes first) v Little-Endian (LSD comes first).
  - Range and size of ***primitive types*** (ints, double etc..) differ from vendor to vendor.
  - ***Character sets*** also differ – Unicode v ASCII v EBCDIC.

Irrespective of the form of communication, the data structures must be "flattened" (converted to a sequence of bytes) before transmission and rebuilt on arrival.

- ***Marshalling***: process of taking a collection of data items and assembling into a form suitable for the transmission of a message.
- ***Unmarshalling***: process of dissembling them on arrival to produce an equivalent collection of data items at the destination.

CORBA CDR

- A transfer syntax, mapping from data types defined in OMG IDL to a bicanonical, low-level representation for transfer between agents.
  - Can represent all of the datatypes that can be used as ***arguments and return values*** in remote invocations in CORBA.
- ***Variable Byte Ordering***
  - Machines with a common byte order may exchange messages without byte swapping.
  - For different byte orders, the message originator determines the message byte order, and the receiver swaps bytes to match its native ordering.
  - Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.
- The types of the data structures and the basic data items are described in the CORBA ***Interface Definition Language (IDL).***

CORBA CDR

- The flattened form represents a Person struct with value: { 'Smith', 'London', 1934}

index in sequence of bytes ← 4 bytes →		
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1934	unsigned long

XML Object (data) Binding

- ***XML databinding***: ***mapping*** an instance of an *XML Schema* into the appropriate ***object*** model (set of classes and types which represent the data) .
- Allows applications (usually data-centric) to manipulate data that has been serialized as XML in a way that is more natural than using DOM.
- Why not use DOM or SAX?
  - May not want to access the structure of an XML document.
  - These APIs can be tedious when working with typed content.
- ***Databinding:***
  - Applications deal with the ***correct representation*** of the data.
  - Provides for strong ***type checking***.
- Popular XML object binding APIs include Castor and JAXB.
  - ***Castor:*** XML->Java, RDBMS, LDAP ***and back!***

XML Databinding & Marshalling

- ***Marshalling Framework***
  - Handles the conversion of Java objects to and from XML.
  - Uses descriptors to obtain information about a class and it's fields.
  - It is designed to work with any "bean-like" class (get/set methods).
  - Uses reflection to create descriptors on the fly.
  - Can use ***mapping file*** to override defaults.
- ***Marshalling API***
  - ***Marshaller*** converts objects to XML instances. Can marshal to a Writer or DocumentHandler.
  - ***Unmarshaller*** converts XML instances to objects. Can marshal from a Reader, or EventProducer. Can return a DocumentHandler.

Object/Databinding Technologies

- ***Castor:*** An open source data-binding framework for Java.
  - ***Greek Mythology***: Castor and Pollux, the twin sons of Leda and brothers of Helen and Clytemnestra, who were transformed by Zeus into the constellation Gemini. See ***castor.org***
  - Can map ***XML to Java/RDBs (via JDO) and vice-versa***.
  - Can marshall/unmarshall XML to java ***without*** mapping definition.
  - Does not need to build any ***intermediate Java classes***.
- ***Zeus:*** ObjectWeb open source Java to XML databinding tool.
  - ***Greek Mythology***: principal god of the Greek pantheon, ruler of the heavens, and father of other gods and mortal heroes.
  - Uses DTD and XML Schemas for mapping. See ***enhydra.org***.
  - See enh
- ***JAXB:*** Sun Microsystem's Java Architecture for XML Binding
  - Provides a ***binding compiler*** and a runtime framework mapping.
  - No support for XML Schema.

Example Using Castor

```
package gmit;
public class Person implements java.io.Serializable {
    private String name = null;
    private java.util.Date dob = null;
    public Person(){ super();}
    public Person(String name){
        this.name  = name;
    }
    public Date getDateOfBirth(){ return dob; }
    public String getName() { return name; }
    public void setDateOfBirth(Date dob) {
        this.dob = dob;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Simple Example Using Castor

To marshal the Person object:

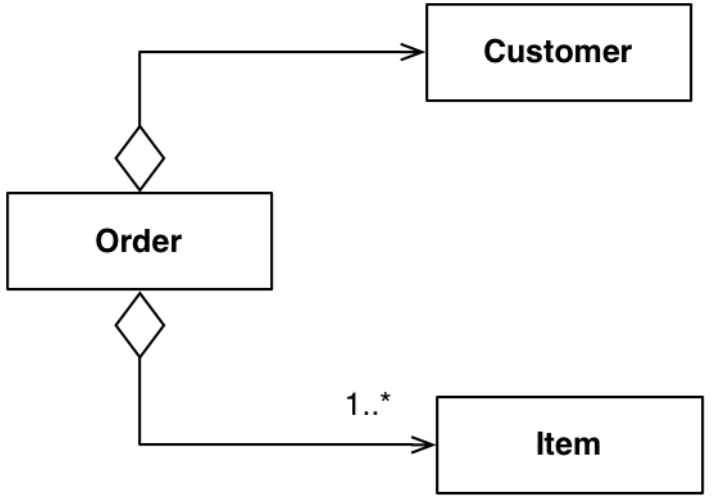
```
Person person = new Person("Michael Murphy");
person.setDateOfBirth(new Date(1970, 7, 7));
// Create a File to marshal to
FileWriter writer = new FileWriter("test.xml");
// Marshal the person object
Marshaller.marshal(person, writer);
```

To unmarshal the XML to a Java object:

```
// Create a Reader to the file to unmarshal from
FileReader reader = new FileReader("test.xml");
// Marshal the person object
Person person = (Person) Unmarshaller.unmarshal(Person.class, reader);
System.out.println(person.getName());
```

XML Representation

- *Consider the following example of representing object state in XML:*




```
<order number="12566">
    <customer code="1">
        <name>Patrick Sarsfield</name>
        <dob>7/7/1660</dob>
        <address1>Ballyneety</address1>
        <address2>Pallasgreen</address2>
        <county>Limerick</county>
        <phone>087-12345678</phone>
    </customer>
    <details>
        <item partNumber="QB-111-AA" partName="Match Lock Musket" qty="1"
              price="99.99"/>
        <item partNumber="QB-233-AB" partName="Pike" qty="2" price="4.00"/>
        <item partNumber="QB-134-CC" partName="Bayonet" qty="2" price="35.00"/>
        <item partNumber="QB-454-ZA" partName="Sword" qty="1" price="350.00"/>
    </details>
</order>
```

JSON

- *JSON (JavaScript Object Notation) is an open standard format for transmission of human-readable key/value pairs.*
  - Douglas Crockford (2001) and IETF RFC 7158.
  - Arose from requirement for stateful, real-time web browser / server communication, independent of plugins.
- ***JSON basic types:***
  - ***Number:*** signed decimal number that may contain a fractional part and may use exponential E notation.
  - ***String:*** sequence of zero or more Unicode characters
  - ***Boolean:*** true / false
  - ***Array:*** ordered list of zero or more values. Uses square bracket notation with elements being comma-separated.
  - ***Object:*** unordered associative array (key/value pairs). Delimited with curly brackets. Uses commas to separate each pair. Keys are distinct strings.

JSON Representation of State


```
{
  "order": {
    "-number": "12566",
    "customer": {
      "-code": "1",
      "name": "Patrick Sarsfield",
      "dob": "7/7/1660",
      "address1": "Ballyneety",
      "address2": "Pallasgreen",
      "county": "Limerick",
      "phone": "087-12345678"
    },
    "details": {
      "item": [
        {
          "-partNumber": "QB-111-AA",
          "-partName": "Match Lock Musket",
          "-qty": "1",
          "-price": "99.99"
        },
        {
          "-partNumber": "QB-233-AB",
          "-partName": "Pike",
          "-qty": "2",
          "-price": "4.00"
        },
        {
          "-partNumber": "QB-134-CC",
          "-partName": "Bayonet",
          "-qty": "2",
          "-price": "35.00"
        },
        {
          "-partNumber": "QB-454-ZA",
          "-partName": "Sword",
          "-qty": "1",
          "-price": "350.00"
        }
      ]
    }
  }
}
```

Java Serialization

- The method of externalisation used in the Java language.
  - Powerful and flexible.
- ***Serialization***: saving the current state of an object to a stream.
- ***Deserialization***: Restoring an equivalent object from that stream.
- Stream functions as a container for the object.
  - Includes a partial representation of the object's internal structure, including variable types, names, and values.
- May be transient (RAM-based) or persistent (disk-based).
  - ***Transient*** container may be used to prepare an object for transmission from one computer to another (e.g. over a socket).
  - ***Persistent*** container, such as a file on disk, allows storage of the object after the current session is finished.

Serializable and Externalizable

- Objects must be an instance of a class that implements either the ***Serializable*** or ***Externalizable*** interface (java.io lib).
  - Can only save data associated with an object's variables.
  - Depend on the class definition being available to the JVM at reconstruction time.
- Serializable interface relies on the Java runtime default mechanism to save an object's state.
  - ***writeObject()*** used to serialize an object (ObjectOutputStream class & ObjectOutput interface).
  - Use ***write<datatype>()*** method to write a primitive value.
  - ***readObject()*** used to deserilaize (ObjectInputStream class)
  - Use read***<datatype>()*** method to read primitives.
- ***Any object references are also written to the stream*** (otherwise lots of null pointers and exceptions…)

Serializable and Externalizable

- ***Externalizable*** interface specifies that the implementing class will handle the serialization on its own.
  - Doesn't rely on the default runtime mechanism.
- Includes which fields get written (and read), and in what order.
- Class must define a ***writeExternal()*** method to write out the stream, and a corresponding ***readExternal()*** method to read the stream.
  - Inside of these methods the class calls ***ObjectOutputStream writeObject()***, ***ObjectInputStream readObject()***, and any necessary ***write<datatype>()*** and ***read<datatype>()*** methods, for the desired fields.
- ***Note:*** *writeExternal() and readExternal()* ***must be declared public***.
  - Increases risk that a rogue object could use them to determine the format of the serialized object.

Serialization - Hiding Data

- Might want to ***prevent certain fields from being stored*** in the serialized object.
  - Serialization allows us to specify that some of fields do not get saved or restored.
  - Done by placing the keyword ***transient*** before the data type in the variable declaration.
- ***Note: static fields are not serialized (written out), and so cannot be deserialized (read back in) – why?***
- Can also override *writeObject()* of *Serializable* to hide data.
  - Will also need to override readObject(). Similar to *Externalizable*.

Serialization - Versioning

- What happens when an object format has been superseded by a new, different version of the class?
  - Need to check during deserialization for backward compatibility.
- Changes to classes specified using a version number.
  - A specific class variable, serialVersionUID (representing the Stream Unique Identifier, or SUID) used to specify the earliest version of the class that can be deserialized.
- The SUID is declared as follows:

```
static final long serialVersionUID = 2L;
```

- ***SUID is a measure of backward compatibility.***
  - Same SUID can be used for multiple representations of a class, as long as newer versions can still read the older versions (interfaces).
- JVM ***automatically assigns a default SUID*** (unless done explicitly). *Secure Hash Algorithm (SHA).*
  - Consists of a hash of class name, interfaces, methods, and fields.

Serialization - Versioning

- Can check the SUID of a class at runtime by querying the JVM:

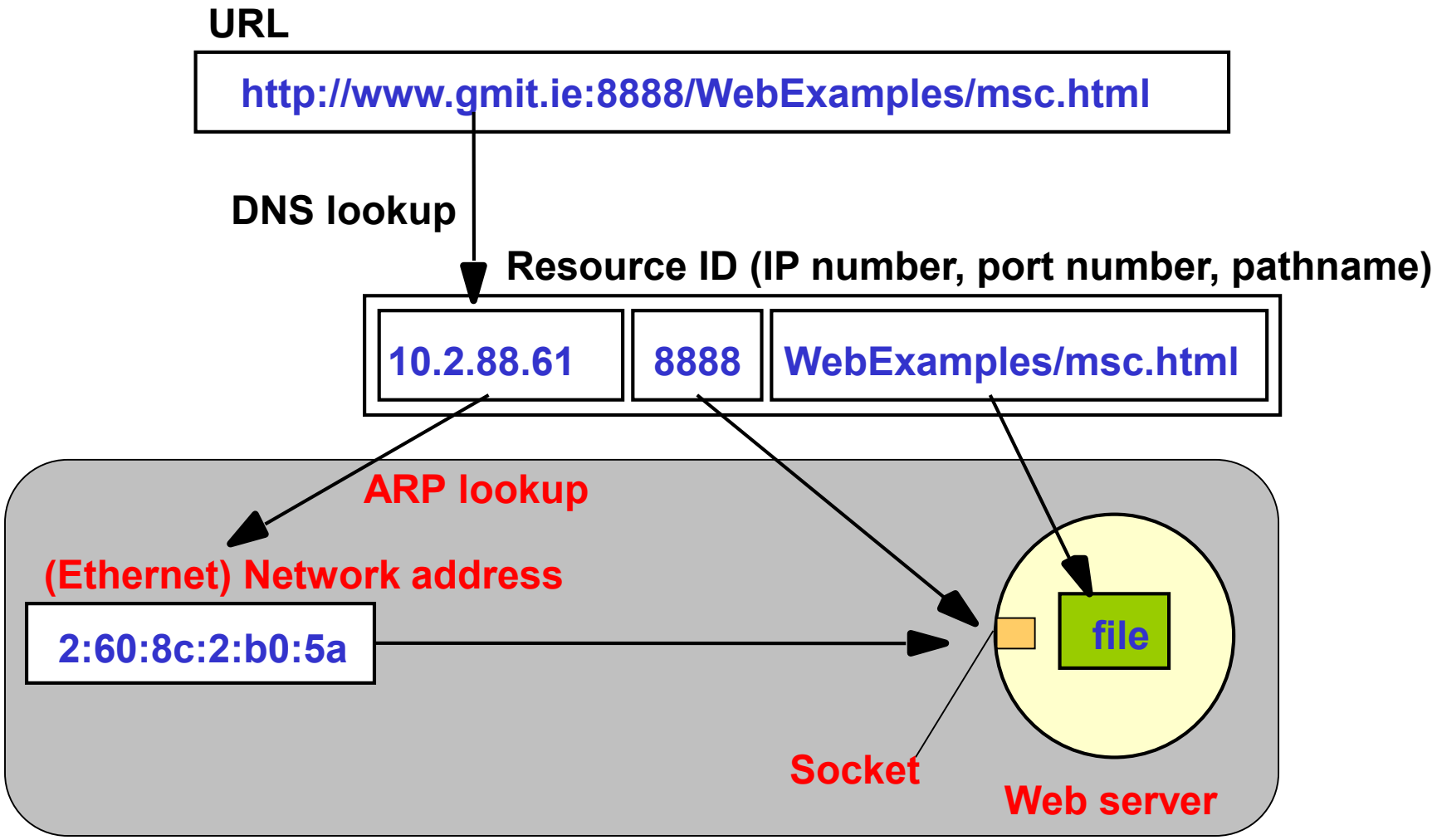
```
ObjectStreamClass myObject = ObjectStreamClass.lookup(
Class.forName( "MyClass" ) );
long theSUID = myObject.getSerialVersionUID();
```

- Can compare the SUID of a restored *Externalizable* object to the class SUID just obtained.
  - If mismatch occurs, we should take appropriate action.
- Java runtime will check the SUID for a *Serializable* object – if we override *readObject()* we'll have to implement this ourselves.
- Problems with forward compatibility – older object with fewer fields. Converse is not the case – default values can be added:
  - ***Deleting a field***, or changing it from non-static or non-transient to static or transient.
  - Changing the position of classes in a ***hierarchy***.
  - Changing the ***data type*** of a primitive field.
  - Changing the ***interface*** for a class from Serializable to Externalizable (or vice-versa).

Naming Services

- Fundamental facility in any computer system.
  - Maps names to objects and object to names.
  - File, email and DNS systems.
- We're primarily interested in mapping names to remote object references – principle remains the same.
- A naming system provides a ***naming service*** for naming-related operations.
  - Naming service is accessed through it's own interface.
  - ***DNS*** offers a name service that maps machine names to IP addresses.
  - ***LDAP*** offers a name service that maps LDAP names to entries.
- To look up an object in a naming system, you supply the ***name*** for that object.
  - Naming convention determines the syntax for valid name.
  - E.g. ***cn=John Healy, o=GMIT, c=Ireland***

Domain Name Service



Naming Services

- The association of a ***name*** with an object is called a ***binding***.
  - E.g. a file name is bound to a file, a machine name to an IP address.
- Naming service may not store object directly – must be ***stored by reference***.
  - A ***pointer or reference*** to the object is placed inside the naming service, e.g. a file handle.
- A ***context*** is a set of name to object bindings.
  - Every context has an ***associated naming convention***.
  - Context ***provides a lookup*** (resolution) operation that returns the object.
  - May ***provide operations*** for binding names, unbinding and listing bound names.
  - A name in one context object ***can be bound to another context*** object (a subcontext) that has the same naming convention.
  - e.g. a unix subdirectory named relative to another directory.

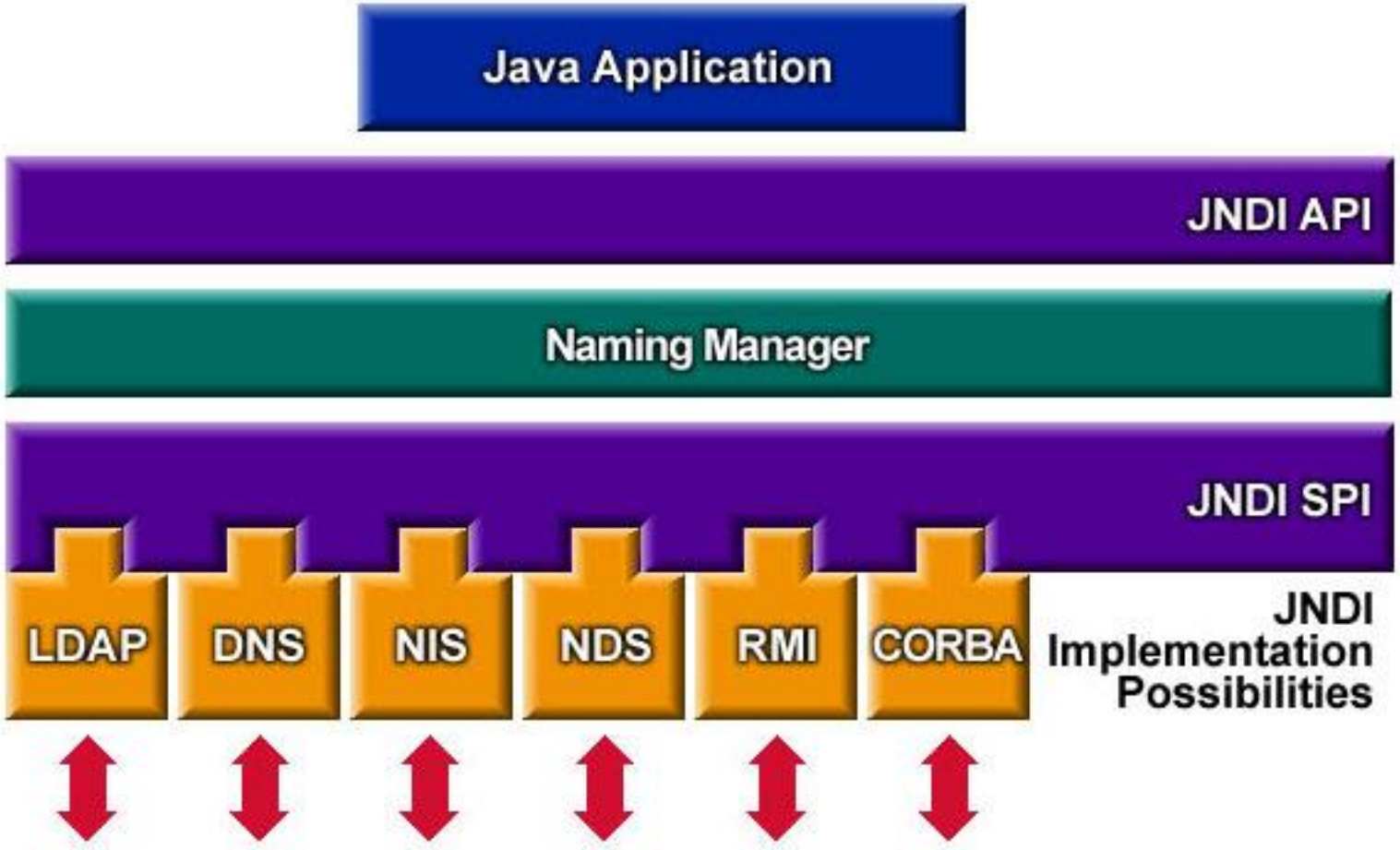
Naming Services

- A naming system therefore can be thought of as ***connected set of contexts of the same type***.
  - Same naming convention and common set of operations.
- A ***namespace*** is the set of names in a naming system.
  - May vary from DNS domains to 128-bit object GUIDs.
- Naming services can be extended with a ***directory service***.
  - Associates names with objects but also ***allows objects to have attributes***. (can search for objects based on attributes).
  - Attributes consist of an ***attribute identifier and a set of attribute*** values (name/value pairs).
- A directory is a connected set of diurectory objects.
  - A directory service provides operations for creating, adding, removing and modifying the attributes of directory objects.
- Examples of directory services include the Novell Directory Service (NDS) and Sun's Network Information Service (NIS).

Java Naming & Dir Interface

- JNDI is an API that provides ***naming and directory*** functionality to applications written in Java.
  - Designed to be ***independent*** of any specific directory service implementation.
  - Can access a variety of directory services in a common way.
- Architecture consists of an ***API and Service Provider Interface*** (SPI).
  - SPI enables different naming & directory services to be plugged in transparently.
- Available with the ***1.3 SDK*** and later releases.
- Includes service providers for:
  - LDAP, CORBA Common Object Services (COS) Name Service and Java Remote Method Invocation RMI Registry.
  - Main ***packages*** are javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap and javax.naming.spi.

JNDI Hierarchy



JNDI Naming Package

- Contains classes and interfaces for accessing naming services.
- ***Context***: core interface for looking up, binding/unbinding, renaming objects and creating and destroying *subcontexts*.
  - Most commonly used operation is ***lookup(java.lang.String)***.
  - Returns the object bound to the lookup name.

```
Printer printer = (Printer) ctx.lookup("HP-2400");
printer.print(report);
```

- ***Binding***: a tuple containing the name of the bound object, the name of the object's class, and the object itself.
  - ***listBindings()*** returns an enumeration of name-to-object bindings.
- ***InitialContext***: provides a starting point for naming and directory operations.
  - All naming and directory operations are performed *relative to a context*. No absolute roots.

JNDI Directory Package

- Allows applications to retrieve attributes associated with objects stored in the directory and to search for objects using specified attributes.
- ***DirContext***: represents a *directory context*.
  - Defines methods for examining and updating attributes associated with a directory object.
  - ***getAttributes(<name>)***: retrieves the attributes associated with a directory object (for which you supply the name).
  - ***modifyAttributes(<name>, <modifications>)***: Add, replace, or remove attributes and/or attribute values.
  - ***search(<name>, <matching attributes>)***: Searches in a single context for objects that contain a specified set of attributes.

Static / Dynamic Object Binding

- Usually we know what objects we want to use at design time (***static binding***).
  - E.g. Person p = new Person();
- ***Dynamic binding***: Code executed to perform a given operation is determined at run time.
  - An expression may denote an object which may have ***more than one possible class*** and that class can only be determined at run time.
  - Binding done "on the fly". Allows us to build powerful applications.
- ***Implemented using interfaces.***
  - Allows us to harness the power of ***polymorphism*** in an OO environment. Even distributed polymorphism!
  - ***Interfaces are the key*** to understanding how may object binding technologies work – Java Interfaces, CORBA IDL, COM ODL and WSDL (web services).