Selected Topics

Selected Topics

- Reflection Library
- Garbage collection
- Java Native Interface
- Remote Method Invocation

Reflection Library

Reflection

- Toolset to write program to manipulate Java code dynamically
  - Reflective program can analyse the capabilities of classes
    - Analyse the capabilities of classes at runtime
    - Inspect objects at runtime
    - Implement generic array manipulation code
    - Use **`Method`** objects like function pointers in C++
- It is of interest mainly to tool builders, not application programmers

Retrieving Class Objects

- `Object.getClass()`
- `Class.forName()`
- The `.class` Syntax

`Class` Class

- Runtime Type Identification
  - A **`Class`** object describes the properties of a particular class

```
Employee e;
. . .
Class cl = e.getClass();
```

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

prints

```
Employee Harry Hacker
```

if **`e`** is an employee, or

```
Manager Harry Hacker
```

if **`e`** is a manager.

`Class` Class

- Obtain a **`Class`** object corresponding to a class name
  - Using the static **`forName`** method

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

  - Use this method if the class name is stored in a string that varies at runtime
- Obtain an object of type **`Class`** using **`T.class`**, if **`T`** is any Java type

```
Class cl1 = Date.class; // if you import java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Analyse the Capabilities of Classes

- **`java.lang.reflect`** package
  - **`Field`**, **`Method`**, and **`Constructor`** classes describe the fields, methods, and constructors of a class
  - **Discovering Class Members**
    - **`getFields`**, **`getMethods`**, and **`getConstructors`** methods of the **`Class`** class return arrays of the public fields, methods, and constructors that the class supports

Analyse Objects at Runtime

- Use **`get`** method in the **`Field`** class to look at fields of objects that were not known at compile time

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();
   // the class object representing Employee
Field f = cl.getDeclaredField("name");
   // the name field of the Employee class
Object v = f.get(harry);
   // the value of the name field of the harry object,
   //    i.e., the String object "Harry Hacker"
```

Uses

- Extensibility Features
  - An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- Class Browsers and Visual Development Environments
  - A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- Debuggers and Test Tools
  - Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language

Drawbacks

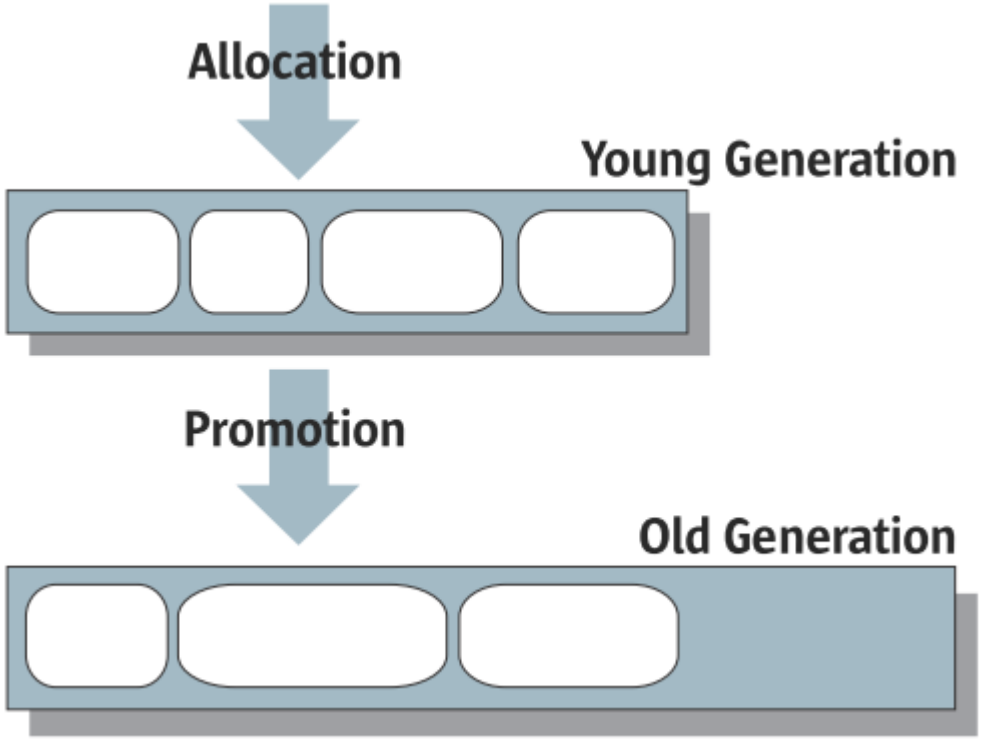
- Performance Overhead
  - Reflective operations have slower performance
- Security Restrictions
  - Reflection requires a runtime permission which may not be present when running under a security manager
- Exposure of Internals
  - Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform
    - Accessing private fields and methods

Garbage Collection

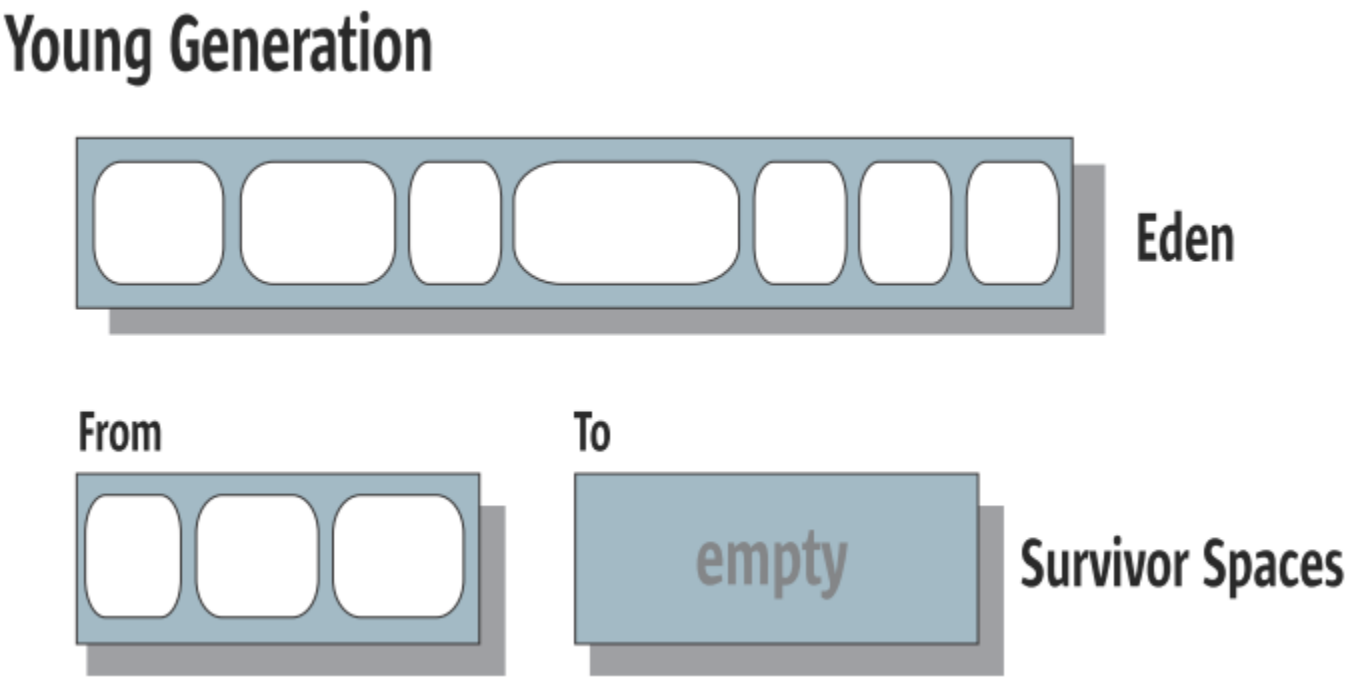
Garbage Collection

- Garbage: objects that are no longer be reached from any object reference
  - They are not useful but occupy memory space
  - Three HotSpot generations of Java heap: ***Young, Tenured (Old) and Permanent*** generations
    - Young generation: ***Eden*** space, ***Survivor 1*** and ***Survivor 2*** space
    - An object is first created on heap in the Young generation inside Eden space
- Garbage collection: JVM detects garbage and reclaims the occupied memory space
  - Automatic memory management relieves the developer from the complexity of memory management and garbage collection (more reliable code)
  - Explicit memory management (like C/C++)
    - Common problems: dangling references (reallocating still referenced space), space leaks (not releasing space no longer referenced)
- Once garbage collection is the principal bottleneck, it is worth understanding some aspects of this hidden implementation
  - For many applications, it does not matter; for some, it does

Generational Garbage Collection



Young Generation Spaces



Garbage Collector

- Garbage collector (GC) is a daemon thread
  - Types of collection
    - ***Minor garbage collection*** or ***Young generation collection*** (moves objects from Eden to Survivor 1 then Survivor 2)
    - ***Major garbage collection*** or ***Full garbage collection*** (moves objects from Young to Tenured generation)
      - It includes tracing all live objects in the heap and sweeping and compacting the tenure generation
  - Types of collectors
    - ***Concurrent GC*** (runs concurrently with the application threads)
    - ***Full GC*** (pauses the application threads)
      - The tenured generation is full before concurrent GC finishes
      - Whenever major garbage collection occurs application threads stops during that period which will reduce application's performance and throughput

Eligibility for Garbage Collection

- Eligibility for garbage collection
  - Not reachable from any live threads or any static references
- Making objects eligible for garbage collection
  - Explicitly assign `null` to the reference variable for the object
  - Object reference goes out the scope in which it is created
  - Parent object is set to `null`
- Explicit garbage collection
  - `System.gc()` (`Runtime.getRuntime.gc()`): sending request of garbage collection to JVM
    - Not guaranteed that garbage collection will happen; not forced
      - JVM makes a best effort to reclaim space from all discarded objects
    - Triggers a full garbage collection
    - There is no manual way of doing garbage collection in Java
- Object pooling – helping the garbage collector
  - Maintaining a pool of frequently used objects and get one form the pool instead of creating a new one

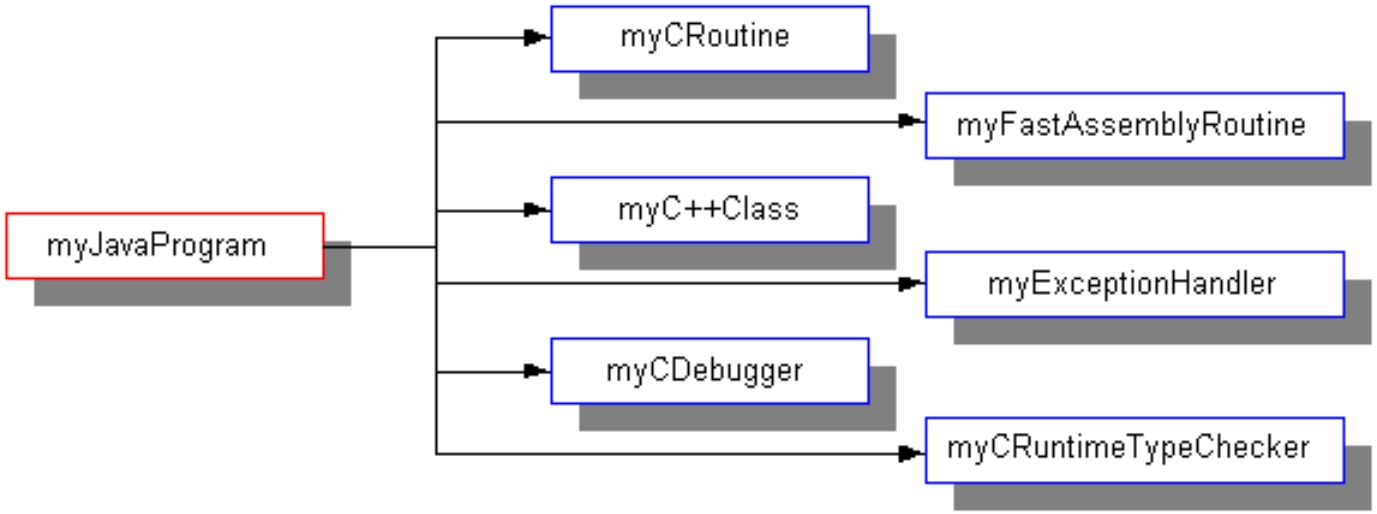
For many applications, explicit nulling, object pooling, and explicit garbage collection will harm the throughput of your application, not improve it

Java Native Interface

Java Native Interface (JNI)

- Use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language

JNI Usage



JNI Example – Creating Java Code

```
class HelloWorld {
    public native void displayHelloWorld();
    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

JNI Example – Create the .h File

```
javah -jni HelloWorld
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
  #define _Included_HelloWorld
  #ifdef __cplusplus
    extern "C" {
  #endif
     /*
     * Class: HelloWorld
     * Method: displayHelloWorld
     * Signature: ()V
     */
     JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld (JNIEnv *, jobject);
  #ifdef __cplusplus
    }
  #endif
#endif
```

JNI Example – Create a Shared Library

- In Java code:

```
System.loadLibrary("hello");
```

- Create Shared Library:

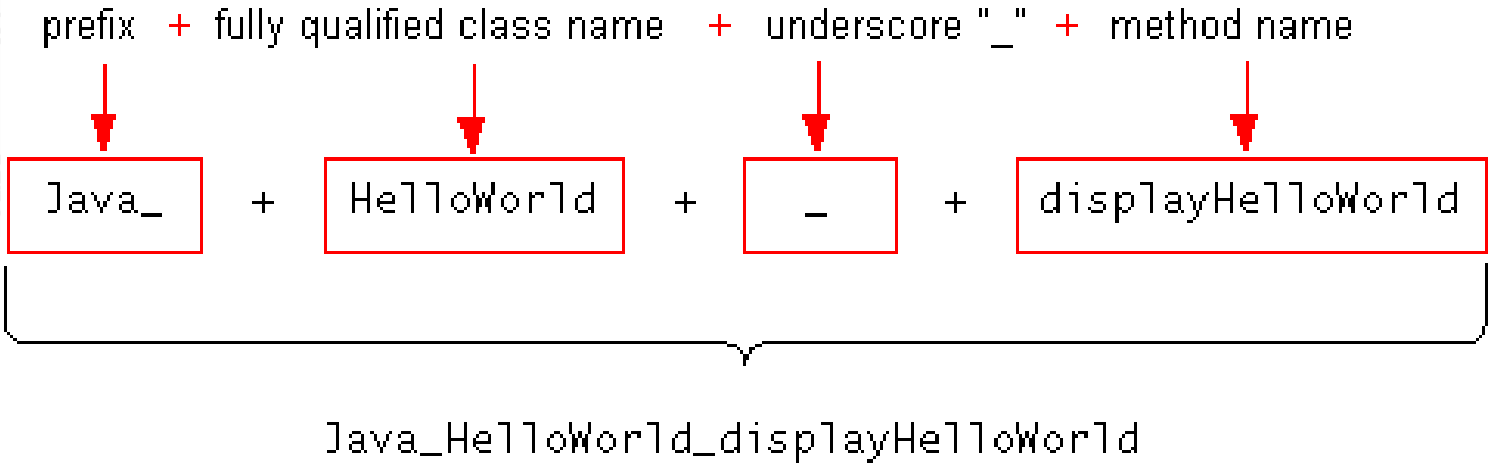
Solaris:

```
cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris \
            HelloWorldImp.c -o libhello.so
```

Windows:

```
cl -Ic:\java\include -Ic:\java\include\win32
            -LD HelloWorldImp.c -Fehello.dll
```

JNI Example – Native Language Function



JNI Example – Write the Native Method Implementation

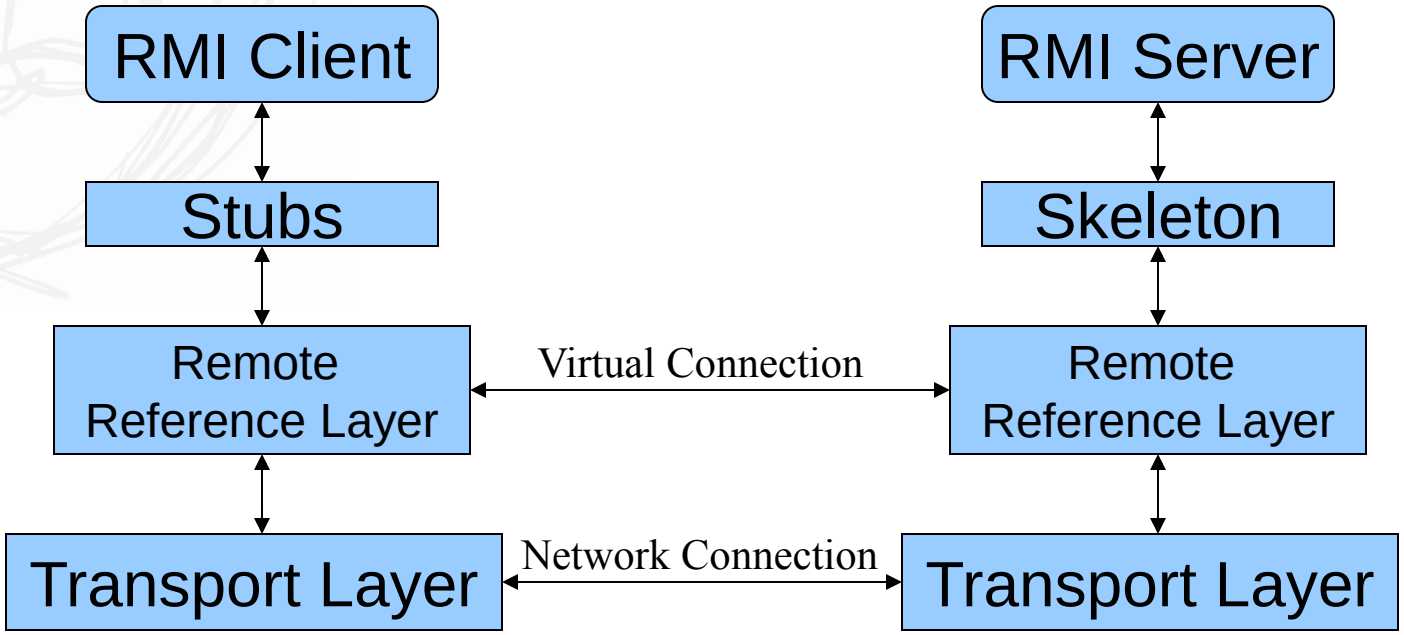
```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL

Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Remote Method Invocation

Java Networking

- Fundamental
  - Socket based networking
    - java.net
- Higher-level
  - Remote Method Invocation (RMI)
    - java.rmi: for Java objects only
  - COBRA
    - org.omg: any other objects (C++)
- Highest-level (request/response model)
  - Servlet
    - javax.servlet/javax.servlet.http
  - JavaServer Page
    - javax.servlet.jsp

RMI Architecture



RMI Solution

- Special feature of Java
- Used to simplify communications between objects in a distributed application
- Socket and thread management are hidden from the programmer
  - RMI Servers
    - Implements a Java RMI object interface and registers the object with Java RMI registry. The Java RMI server must have access to skeletons for the server class
  - RMI Registry
    - A daemon that keeps track of all the available remote objects
  - RMI Clients
    - Calls to a Java RMI registry to look up remote objects. The Java RMI client must have access t a stub for the server class.

Creating RMI Application – An Example

- Develop remote interface

```
import java.rmi.*;

public interface Hello extends Remote {

        String sayHello() throws RemoteException;

}
```

Creating RMI Application – An Example

- Create the implementation class (Servant)

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloImpl extends UnicastRemoteObject implements Hello{
        private String name;
        public HelloImpl(String s) throws RemoteException {
                super();
                name =s;
        }
        public String sayHello() throws RemoteException {
                return "Hello from RMI HelloServer!";
        }

}
```

Creating RMI Application – An Example

- Create the server

```
import java.rmi.Naming;
public class HelloServer {
    public static void main(String args[]){
        try{
            HelloImpl obj = new HelloImpl("HelloServer");
            Naming.rebind("HelloServer", obj);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Creating RMI Application – An Example

- Create the Client Application

```
import java.rmi.*;

public class HelloApp {
    public static void main(String[] args) {
        String message = "";
        try{
           Hello obj = (Hello)Naming.lookup("rmi://localhost/"+"HelloServer");
           message = obj.sayHello();
        }catch(Exception e){
           e.printStackTrace();
        }
        if(!message.equals(""))
          System.out.println("RMI sayHello: "+message);
        }
}
```

Creating RMI Application – An Example

- Create the Stubs and Skeletons
  - The stub compiler runs against the previously compiled class files

```
> rmic HelloImpl
```

  - It generates the stub and skeleton files

HelloImpl_Stub.class

HelloImpl_Skel.class (no longer necessary for Java 2)

Running RMI Application – An Example

- Start Java RMI registry

```
> rmiregistry
```

- Start the server

```
> java HelloServer
```

- Start the client

```
> java HelloApp
```

Stubs and Parameter Marshalling

- Stub method (client side)
  - An identifier of the remote object to be used
  - A description of the method to be called
  - The marshaled parameters
- Receiver object (server side)
  - Unmashals the parameters
  - Locates the object to be called
  - Calls the desired method
  - Captures and marshals the return or exception of the call
  - Sends a package consisting of the marshaled return data back to the client
- Client stub
  - Unmarshals the return value or exception from the server as the return value of the stub call or rethrows the exception

Good news: this complex process is automatic and transparent from the programmer

RMI Applications – A Summary

- Development
  - Develop remote interface
  - Implement remote interface
  - Create the server program
  - Create the client program
  - Generate stubs and skeletons
- Deployment
  - Start Java RMI registry
  - Start the server
  - Start the client