Reflection

Reflection

Reflection allows the program to examine classes at *run time* to discover:

- inheritance, declarations, and modifiers
- fields (and change their values)
- methods (and invoke them)
- constructors (and instantiate objects)

The Class object

java.lang.Class is the starting point for reflection.

Various ways to obtain a Class:

- getClass()

```
String s = new String("foo");
Class c = "foo".getClass();           // object
byte[] bytes = new byte[1024];
Class c = bytes.getClass();           // array
Set<String> s = new HashSet<String>();
Class c = s.getClass();               // interface
```

- .class

```
Class c = java.io.PrintStream.class;
Class c = boolean.class;
Class c = int[][][].class;
```

The Class object

- forName()

```
Class c = Class.forName("java.io.PrintStream");
Class cDoubleArray = Class.forName("[D");
Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

- Class traversal

```
Class.getSuperclass()
Class.getInterfaces()
Class.getClasses()
Class.getDeclaredClasses()
Class.getDeclaringClass()
Class.getEnclosingClass()
```

Finding members

```
public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  System.out.println("Class: " + c.getCanonicalName());

  Field[] fields = c.getDeclaredFields();
  Method[] methods = c.getDeclaredMethods();
  Constructor[] constructors = c.getDeclaredConstructors();

  for (Field fld: fields) {
    System.out.println(fld.toGenericString());
  }
  for (Method meth: methods) {
    System.out.println(meth.toGenericString());
  }
  for (Constructor ctor: constructors) {
    System.out.println(ctor.toGenericString());
  }
}
```

What exceptions might be thrown?

Finding members

- ClassNotFoundException
- java ClassSpy java.awt.Point

Finding members

- `getDeclared{Fields,Methods,Constructors}`
  - includes private members
  - only those declared in this class
- `get{Fields,Methods,Constructors}`
  - only public members
  - includes inherited members
- `get{,Declared}{Field,Method,Constructor}(String name, Class<?>... paramTypes)`
  - retrieve by name and parameter types

Fields

```
public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  Field f = c.getField(args[1]);
  Class<?> t = f.getType();
  System.out.println("Type: " + t);
  int mods = f.getModifiers();

  System.out.print("Modifiers: ");
  if (Modifier.isPublic(mods))    System.out.print("public ");
  if (Modifier.isProtected(mods)) System.out.print("protected");
  if (Modifier.isPrivate(mods))   System.out.print("private ");
  if (Modifier.isStatic(mods))    System.out.print("static ");
  if (Modifier.isFinal(mods))     System.out.print("final ");
}
```

Getting and setting fields

```
public class Book {
  public String[] characters = { "Alice", "White Rabbit" };
  private long chapters = 0;
}

Book book = new Book();
Class<?> c = book.getClass();

Field chars = c.getDeclaredField("characters");
String[] oldChars = (String[]) chars.get(book);
String[] newChars = { "Queen", "King" };
chars.set(book, newChars);

Field chap = c.getDeclaredField("chapters");
// chap.setAccessible(true);
long before = chap.getLong(book);
chap.setLong(book, 12);
```

What exceptions might be thrown?

Getting and setting fields

FieldModifier – 3 variations

Can use to access private fields (IllegalAccessException, NoSuchFieldException)

Even final fields – but may cause problems

Watch out for incompatible object types (IllegalArgumentException)

Primitive vs. reference types – no auto-boxing

Methods

```
public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  Method[] allMethods = c.getDeclaredMethods();
  for (Method m : allMethods) {
    System.out.println(m.toGenericString());
    System.out.println("ReturnType" + m.getReturnType());

    Class<?>[] pType  = m.getParameterTypes();
    for (int i = 0; i < pType.length; i++) {
      System.out.println("ParameterType" + pType[i]);
    }

    Class<?>[] xType  = m.getExceptionTypes();
    for (int i = 0; i < xType.length; i++) {
      System.out.println("ExceptionType" + xType[i]);
    }
  }
}
```

Invoking methods

```
public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  Object t = c.newInstance();

  Method[] allMethods = c.getDeclaredMethods();
  for (Method m : allMethods) {
    Class<?> rType = m.getReturnType();
    Class<?>[] pType = m.getParameterTypes();

    if ((pType.length == 1)
         && (Date.class.isAssignableFrom(pType[0].getClass()))
         && (m.getGenericReturnType() == boolean.class)) {

      System.out.println("invoking  " + m.getName());
      m.setAccessible(true);
      Object o = m.invoke(t, new Date(args[1]));
      System.out.println("returned " + (Boolean) o);
}
```

What exceptions might be thrown?

Invoking methods

- NoSuchMethodException
- IllegalAccessException
- IllegalArgumentException
- InvocationTargetException

Invoking methods (2)

```
public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  Class[] argTypes = new Class[] { String[].class };
  Method main = c.getDeclaredMethod("main", argTypes);
  String[] mainArgs = Arrays.copyOfRange(args, 1, args.length);
  System.out.println ("Invoking main() on " + c.getName());

  // variable number of arguments
  main.invoke(null, (Object)mainArgs);
}
```

- java InvokeMain ClassSpy java.awt.Point

Invoking constructors

```
class EmailAliases {
  private Set<String> aliases;
  private EmailAliases(HashMap<String, String> h) {
    aliases = h.keySet();
  }
}

public static void main(String... args) {
  Class<?> c = Class.forName(args[0]);
  Object t = c.newInstance();   // calls 0-arg constructor
                                // throws InstantiationException

  private static Map<String, String> defaultAliases =
      new HashMap<String, String>();
  Constructor ctor =
      EmailAliases.class.getDeclaredConstructor(HashMap.class);
  ctor.setAccessible(true);
  EmailAliases email = (EmailAliases)ctor.newInstance(defaultAliases);
}
```

What exceptions might be thrown? Why not just use new()?

Invoking constructors

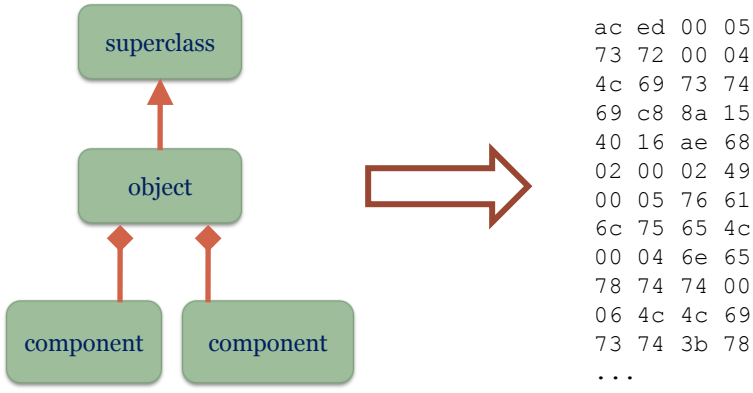
Missing default 0-argument constructor

Wrong argument type passed

No method resolution performed (if Object constructor found and String passed, String constructor will not be invoked)

IllegalAccessException

Serialization



Serialization (2)

The writeObject() and readObject() methods in ObjectOutputStream can be used to save and restore objects from a stream:

```
// sender
FileOutputStream s = new FileOutputStream("file");
ObjectOutputStream o = new ObjectOutputStream(s);
o.writeObject(car);
o.close();

// receiver
FileInputStream s = new FileInputStream("file");
ObjectInputStream o = new ObjectInputStream(s);
Car car = (Car) o.readObject();
o.close();
```

The Serializable interface

- The default implementation traverses the object graph to serialize the complete structure reachable from the original object.
  - what about cyclic references?
- A class is marked as being serializable by implementing the `java.io.Serializable` interface (which does not define any methods or fields).
- If the superclass is not serializable, the superclass must have an accessible 0-argument constructor (to allow its fields to be initialized).
- `transient` can be used to mark fields that should not be serialized.
  - what value should be set on deserialization?

Custom serialization

An object can customise its serialization by implementing the methods:

```
// instead of ObjectOutputStream.defaultWriteObject()
private void writeObject(java.io.ObjectOutputStream out)
private void readObject(java.io.ObjectInputStream in)

// write a replacement object instead of this one
<access-modifier> Object writeReplace()
<access-modifier> Object readResolve()
```

Implementing serialization

Serialization can be performed using reflection, as follows.

Basic implementation:

- If object is null, write null
- If object previously written, write handle
- If writeReplace defined, call it and serialize the returned replacement object
- If object is an array, write the class descriptor of the array type, length of the array, then serialize each element
- If object is a primitive type, encode it
- Otherwise, object is a regular object
  - Write its class descriptor
  - If superclass is serializable, serialize that
  - If class has a writeObject method, call that; otherwise, recursively serialize each serializable field

The Externalizable interface

- Full control over serialization can be obtained by implementing the `java.io.Externalizable` interface
- The object must provide implementations of the methods:

```
public void writeExternal(ObjectOutput out)
public void readExternal(ObjectInput in)
```

- It must have a public 0-argument constructor
- Superclass state must be explicitly saved

Security considerations

- Writing a object?
- Reading a object?
- Object consistency? (invariants etc)

Security considerations

- Writing a object? may expose private internal information
- Reading a object? may permit construction of invalid objects
- Object consistency?
- may not be maintained – constructors and field initialisers not called

Versioning

- The class that reads a serialized object may be a different version than the one that wrote it.
- `private static final long serialVersionUID` defines compatible classes (defaults to a hash of the class definition)
- Receiver is responsible for interpreting stream correctly
- Compatible changes:
  - Adding fields
  - Adding/removing classes from inheritance hierarchy
  - Changing field access
  - Making field non-static or non-transient
- Incompatible changes:
  - Removing fields
  - Reordering inheritance hierarchy
  - Changing field types

Generics

Motivation:

```
List list = new LinkedList();
list.add(new Integer(15));
Integer x = (Integer) list.get(0);
```

Although we put an Integer in, we got an Object out.
Can we arrange things so that we get an Integer out?

```
list.add(new String("oops"));
```

More importantly, can we make sure that only Integers can be put in?
Analogous to `Object[]` vs. `Integer[]`

Generics

First try:

```
public class IntegerList {
  public void add(Integer i);
  public Integer get(int index);
}

IntegerList list = new IntegerList();
list.add(new Integer(15));
Integer x = list.get(0);
```

Generics

New idea – *type parameters*:

```
List<Integer> list = new LinkedList<Integer>();
list.add(new Integer(15));
Integer x = list.get(0);    // aha!
```

Now the compiler can enforce the type checking for us – no more casts.

[TypeChecking.java]

Generics

Type parameters work just like formal parameters in method calls:

```
public interface List<T> {
  void add(T item);
  T get(int index);
  Iterator<T> iterator();
}

public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

Only reference types can be used as parameters, not primitive types.

Generics

This is **not** the same as a macro expansion (vs. C++ templates).

There is only one copy of the class – the actual type arguments are substituted when the declaration is used.

```
List<Float> lf = new LinkedList<Float>();
List<Integer> li = new LinkedList<Integer>();
System.out.println(lf.getClass() == li.getClass());
// prints true!
```

Static members are shared among all instances of the generic class.

Cannot refer to the formal type parameters in a static method or field.

Generics

This works for superclasses and interfaces, too:

```
class MySubClass<T> extends MySuperClass<T>
  implements InterfaceA<T>, InterfaceB<T> {
  ...
}
```

Generics

We can also use more than one type parameter:

```
public interface Map<K,V> {
  public V put(K key, V value);
}
```

or mix formal and actual types…

```
public interface StringMap<String,V>
```

or nest type parameters…

```
List<Map<String,Person>> allMaps
```

Subtyping

Are we allowed to do this?

```
List<String> ls = new LinkedList<String>();
List<Object> lo = ls;   // ?

lo.add(new Object());
String s = ls.get(0);   // Object becomes a String!
```

In general, a List<String> is not a subtype of List<Object>.

Wildcard types

So what is a supertype of all Lists?

```
void printList(List<?> list) {
    for (Object e : list) {
        System.out.println(e);
    }
}
```

Unfortunately, we can't put things in it:

```
List<?> list = new LinkedList<String>();
list.add(new Object());  // ?
```

Bounded wildcards

```
public abstract class Shape {
    public abstract void draw();
}
public class Circle extends Shape...
public class Rectangle extends Shape...

public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw();
   }
}
```

Can we do this?

```
List<Circle> circles = new LinkedList<Circle>();
drawAll(circles);
```

Bounded wildcards

What we really want is this:

```
public void drawAll(List<? extends Shape> shapes) {
    for (Shape s: shapes) {
        s.draw();
   }
}
```

Bounded wildcards

However, we still can't add things to the list:

```
public void addCircle(List<? extends Shape> shapes) {
    shapes.add(new Circle());
}
```

Not a List of (things that are shapes), it's a (List of T), where T is a particular subclass of shape.

Generic methods

Let's try this:

```
static void makeListFromArray(List<Object> c, Object[] arr)
  {
    for (Object elem : arr) {
        c.add(elem);   // ?
    }
}
```

Is this any better?

```
static void makeListFromArray(List<?> c, Object[] arr) {
    for (Object elem : arr) {
        c.add(elem);   // ?
    }
}
```

- Works, but only on Object[]'s
- Can't put things into a List<?>

Generic methods

Solution:

```
static <T> void makeListFromArray(List<T> c, T[] arr) {
    for (T elem : arr) {
        c.add(elem);    // whew!
    }
}
```

T is assigned by the compiler using *type inference*:

```
String[] strarr = new String[100];
Integer[] intarr = new Integer[100];
List<String> strlist = new ArrayList<String>();
List<Integer>intlist = new ArrayList<Integer>();

makeListFromArray(strlist, strarr);    // T = String
makeListFromArray(intlist, intarr);    // T = Integer
makeListFromArray(intlist, strarr);    // T = ?
```

- Type conflict

Lower-bounded wildcards

We can also write `? super Shape`, e.g.

```
interface Sink<T> { flush(T t); }
public static <T> T writeAll(List<T> list,
                             Sink<? super T> sink) {
  T last;
  for (T t : list) {
    last = t;
    sink.flush(last);
  }
  return last;
}

Sink<Object> objsink;
List<String> strlist;
String str = writeAll(strlist, objsink);
```

Lower-bounded wildcards

Why not this?

```
public static <T> T writeAll(List<? extends T>, Sink<T>)
T last;
  for (T t : list) {
    last = t;
    sink.flush(last);
  }
  return last;
}

Sink<Object> objsink;
List<String> strlist;
String str = writeAll(strlist, objsink);
```

- Wrong return type

Type inference

Type inference expresses dependencies between types:

```
class Collections {
  public static <T> void copy(List<T> dest,
                              List<? extends T> src) {
}
```

In fact, Collections actually uses:

```
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src)
```

Type inference

In general, if something is passed into an API, you should use `? extends T`

Conversely, if an API returns something, you should use `? super T`

Type erasure

To interoperate with legacy code, generics are implemented using *type erasure* – type information is removed from the run-time binary:

```
public Integer canYouBelieveIt() {
  List<Integer> li = new LinkedList<Integer>();
  List nongenericli = li;
  nongenericli.add(new String("foo"));
  return li.iterator().next();
}
```

Compiles, but with a warning...

```
warning: [unchecked] unchecked call to add(E) as a
  member of the raw type java.util.List
    nongenericli.add(new String("foo"));
```

Type erasure

Effectively, this is compiled to:

```
public Integer canYouBelieveIt() {
  List<Integer> li = new LinkedList<Integer>();
  List nongenericli = li;
  nongenericli.add(new String("foo"));
  return (Integer) li.iterator().next();
}
```

Type erasure

Because of type erasure, actual type information is not available at run time:

```
Collection cs = new LinkedList<Float>();
// can't do this
if (cs instanceof Collection<String>) {...}

// or this
public class MyClass<E> {
  public static void myMethod(Object item) {
    if (item instanceof E) {   // Compiler error
        ...
    }
    E item2 = new E();         // Compiler error
    E obj = (E)new Object();   // Unchecked cast warning
}
```

Class literals

`Class` is actually a generic type, `Class<T>`

What is the `T`?

```
public T newInstance();
```

Class literals

For example, creating a list through reflection:

```
List<EmpInfo> emps = sql.select(EmpInfo.class,
                                "select * from emps");

public static <T> List<T> select(Class<T> c, String query) {
  List<T> result = new ArrayList<T>();
  // run SQL query
  for (...iterate over results...) {
    T item = c.newInstance();
    // build item from SQL results
    result.add(item);
  }
  return result;
}
```

Generics and arrays

Arrays may not be created whose element type is a parameterized type.

Because of type erasure, elements having different type parameters cannot be distinguished:

```
List<String>[] lsa = new List<String>[10]; // trouble
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li;                    // unsound
String s = lsa[1].get(0); // ClassCastException
```

Generics and arrays (2)

Exception – arrays of unbounded wildcard types are permitted:

```
List<?>[] lsa = new List<?>[10];   // allowed
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Correct.
String s = (String) lsa[1].get(0); // Run time error, but
  cast is explicit.
```

Generics and arrays (3)

Arrays whose element types are type variables aren't permitted either:

```
<T> T[] makeArray(T t) {
    return new T[100]; // Error.
}
```

At runtime, there's no way to discover what type T is.
(Need to use reflection and newInstance() to work around).

Generic key-value store

Suppose I want to store key-value pairs and I want a generic definition that will work for keys of any type and values of any type.

```
interface KeyValueStore<T,S> {
  public void add(T key, S value);
  public S get(T key);
  public S remove(T key);
}
```

Keys can be T or any subclass of T.
Values can be S or any subclass of S, but will be cast to S when read.

Generic key-value store

First try:

```
class KeyValueStoreImpl<T,S> implements KeyValueStore<T,S> {
  private static final int SIZE = 10;
  private T[] keys;
  private S[] values;
  private int nextslot=0;

  KeyValueStoreImpl() {
    keys = new T[SIZE];
    values = new S[SIZE];
  }
}
```

```
private T[] keys;
  private S[] values;
```

- - valid

```
keys = new T[SIZE];
    values = new S[SIZE];
```

- - invalid

Generic key-value store

Better:

```
class KeyValueStoreImpl<T,S> implements KeyValueStore<T,S> {
  private ArrayList<T> keys;
  private ArrayList<S> values;

  KeyValueStoreImpl() {
    keys = new ArrayList<T>();
    values = new ArrayList<S>();
  }
  public void add(T key,S value) {
    keys.add(key);
    values.add(value);
  }
  public S get(T key) {
    int i = keys.indexOf(key);
    if (i>-1) return values.get(i);
    return null; }
}
```

Java Native Interface

The Java Native Interface (JNI) allows code in other languages (e.g. C, assembly) to be called from Java.

For example:

- To access facilities like special hardware not provided by standard APIs
- To reuse an existing library in another language
- To optimize part of an application

although

- Now we have just-in-time compilation
- JNI itself imposes an overhead

Native methods

```
class HelloWorld {
  // native method declaration
  public native void displayHelloWorld(String s);

  // static initialiser
  static {
    System.loadLibrary("hello");
  }

  public static void main(String args[]) {
    // native method call
    new HelloWorld().displayHelloWorld(args[0]);
  }
}
```

Native methods (2)

To compile:

```
javac HelloWorld.java
javah -jni HelloWorld
```

Creates:

```
HelloWorld.class – compiled Java class
HelloWorld.h – header to write C native method against
```

Native header

```
#include <jni.h>

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

Native header (2)

jni.h – declares conversion bits between C and Java (types etc)

JNIEnv *: A pointer to the JNI environment. This pointer is a handle to the current thread in the Java virtual machine, and contains housekeeping information and various useful function pointers:

- FindClass to get the jclass for a specified name
- GetSuperclass to map one jclass to its parent
- NewObject to allocate an object and execute a constructor on it
- CallObjectMethod, CallBooleanMethod, CallVoidMethod, etc. for method calls
- GetObjectField, GetCharField, etc. and corresponding Set...Field operations

jobject: A local reference to a Java object (this). If the calling method is static, this parameter would be type jclass instead of jobject.

jstring: Here, the parameter supplied to the native method.

Native implementation

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

JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld(JNIEnv
  *env, jobject obj, jstring js) {
  jboolean iscopy;
  const char *str = (*env)->GetStringUTFChars(env, js, &iscopy);
  printf("Hello World: %s\n", str);
  return;
}
```

Compiling and running

Build using:

```
$ JINCLUDE=/home/jkf/java/jdk1.5.0/include
$ gcc HelloWorldImpl.c -I$JINCLUDE -I$JINCLUDE/linux -shared \
-fpic -o libhello.so
```

This makes the shared library libhello.so.

Run using:

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
$ java HelloWorld
```

Converting Java types

In Java, Strings are 16-bit unicode
In C, strings are 8-bit character arrays

Convert using:

```
const char *cstr = (*env)->GetStringUTFChars(env, jstr, iscopy)
jstring jstr = (*env)->NewStringUTF(env, cstr)
(*env)->ReleaseStringUTFChars(env, jstr, cstr);
```

Arrays can be passed using:

```
NewIntArray
GetIntArrayElements
GetIntArrayRegion/SetIntArrayRegion
ReleaseIntArrayElements
etc.
```

Calling Java methods

First, need to obtain a class reference:

```
jclass cls = (*env)->FindClass(env, "ClassName");   // by name
jclass cls = (*env)->GetObjectClass(env, jobj);     // from a jobject
JNIEXPORT void JNICALL Java_meth(JNIEnv *env, jclass jcls)
                                                    // from a jclass
```

Then, get the method identifier:

```
jmethodID mid = env->GetMethodID(cls, "sendArrayResults",
  "([Ljava/lang/String;)V");
  // arguments: class, method name, method signature
  // method signatures can be found using javap -s
```

Finally, call the method:

```
env->Call{Void,Object,Boolean,etc}Method(jobj, mid, ret, args...);
```

Accessing Java fields

Similarly, to get a field identifier:

```
jfieldID fid = env->GetFieldID(cls, "arraySize",  "I");
```

To get the field value:

```
int arraysize = env->GetIntField(jobj, fid);
```

To set the field value:

```
env->SetIntField(jobj, fid, arraysize);
```

Custom class loaders

- *Class loaders* supply the JVM with class definitions
- Built-in class loaders:
  - Bootstrap – basic trusted classes
  - Extension – standard Java extensions
  - System – reads definitions in from your .class files
- Custom class loaders obtain classes from other sources – e.g. downloaded from the network, or dynamically generated

Defining a custom class loader

- Custom class loaders extend java.lang.ClassLoader
- c.loadClass(name) requests that c loads the named class, returning a java.lang.Class object.
- The default implementation of loadClass:
  - tests whether the class is already loaded,
  - delegates to a parent class loader to load it,
  - otherwise calls c.findClass(name).
- To implement a custom class loader, override findClass
- findClass can use defineClass to build a new Class object from bytecode
- Multiple class loaders can be chained in trees

Example: network class loader

```
class NetworkClassLoader extends ClassLoader {
  String host;
  int port;

  public Class findClass(String name) {
    byte[] b = loadClassData(name);
    return defineClass(name, b, 0, b.length);
  }

  private byte[] loadClassData(String name) {
    // load the class data from the connection
  }
}
```

To use:

```
ClassLoader loader = new NetworkClassLoader(host, port);
Object main = loader.loadClass("Main", true).newInstance();
```

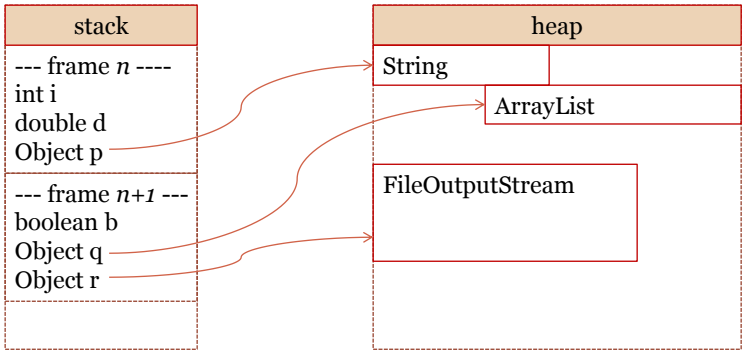
Uses of custom class loaders

- Permits dynamic loading/unloading of classes
- Permits classes to be replaced on the fly

- Classes are distinguished by name and class loader
  - Permits multiple classes of the same name
  - e.g. for hosting multiple applications in servlet containers

Memory management

Program memory is divided into two parts: *stack* and *heap*.



Memory management (2)

Java uses *garbage collection* to reclaim memory from unneeded objects in the heap.

Garbage collection

Advantages

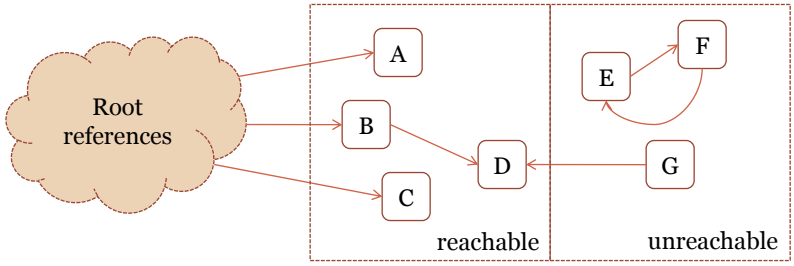
- Easier/faster code development
- Fewer memory leak bugs
- Better stability for long-running programs
- Easier to coordinate between multiple threads

Disadvantages

- Increased run-time overhead
- Less control over memory footprint
- No real-time guarantees

Garbage collection (2)

- The garbage collector identifies when an object is no longer needed.
- Memory won't be reclaimed while an object is *reachable* from the root:
  - Referenced by a static field
  - Referenced by a local (stack) variable in a live thread
  - Referenced by another reachable object
  - And not yet finalized



Garbage collection (3)

Garbage collection used to be associated with long, unpredictable pauses.

Modern garbage collection techniques:

- Generational collection: “most objects die young” → keep a small young generation which can be collected quickly and frequently.
- Parallel collection: multiple processors work on GC at the same time → application pauses for less time.
- Concurrent collection: GC occurs at the same time as application execution.
- Incremental collection: GC occurs in small bursts, e.g. each time an object is allocated: -Xincgc.

Classloaders and GC

- Only classes loaded by the system classloader contain root references.
- If the program creates custom classloaders, those classloaders themselves are subject to collection.
- Any classes created by those classloaders will also be collected.
- This allows an application server to undeploy an entire application.

Finalizers

When the garbage collector detects that an object is unreachable, it runs its finalizer (defined in java.lang.Object).

Finalizers are meant to clean up external resources used by the object, e.g.

- Close down network connections
- Flush buffers to file

Can also log debugging information – e.g. to monitor GC performance

Finalizers

Guarantees:

- Will not run before an object becomes unreachable.
- Will be invoked at most once on a given object.
  - may cause trouble with 'resurrected' objects

Un-guarantees:

- Can be executed at any time after object becomes unreachable
- May not run at all before program exit
- Can be executed on objects in any order
- Can be executed by an arbitrary thread (e.g. single dedicated thread, one thread per class, etc.)
- System.runFinalization() will cause the JVM to 'make a best effort' to complete any outstanding finalizations.

Must assume finalizers might run concurrently with anything else – make sure to avoid deadlocks and infinite loops.

Resurrection example

```
class Restore {
  int value;
  Restore next;
  static Restore found;

  Restore(int value) {
    this.value = value;
    this.next = null;
  }
  public void finalize() {
    synchronized (Restore.class) {
      this.next = found;
      found = this;
    }
  }
}
```

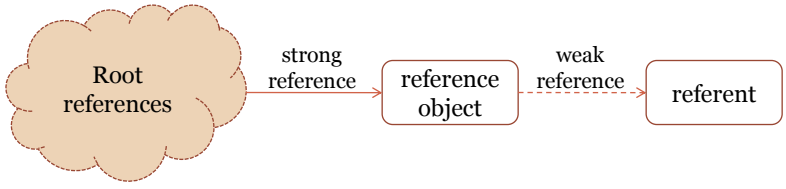
Tricky!

Reference objects

Reference objects give the programmer greater control over memory management.

A reference object holds a reference to another object (the *referent*) which the garbage collector understands as a special type of *weak reference*.

Normal pointers are *strong references*.



Reference objects (2)

```
package java.lang.ref;

public abstract class Reference<T> {
  public T get();        // return the referent
  public void clear();   // clear the reference
}
```

When an object becomes only weakly reachable, it becomes a candidate for collection.

Before collecting the referent, the GC will clear the weak reference – subsequent calls to get() will return null.

The program can also manually clear() the reference.

The referent cannot be set to a different one.

Reference objects (3)

There are three successively weaker types of reference objects:

- SoftReference
- WeakReference
- PhantomReference

Soft references

A SoftReference allows its reference to be collected when the garbage collector decides that it needs the memory.

Useful for implementing a cache of large objects – for example, images read from disk.

Example: Image cache

```
public class DisplayImage extends Applet {
  SoftReference<Image> sr = new SoftReference<Image>();

  public void paint(Graphics g) {
    // check if referent still exists
    // need to obtain strong ref first (why?)
    Image im = sr.get();

    if (im == null) {
      im = getImage(...)
      sr = new SoftReference<Image>(im);   // cache it
    }
    if (sr.get() != null)
      g.drawImage(sr.get(), 25, 25, this);
    im = null;   // clear strong reference
  }
}
```

- Avoid race condition where get() returns the referent, but it gets cleared before we have a chance to use it.

Example: Object loitering

Another use of SoftReferences is preventing object loitering:

```
public class LeakyChecksum {
    private byte[] arr;    // hangs around

    public int getFileChecksum(String fileName) {
        int len = getFileSize(fileName);
        if (arr == null || arr.length < len)
            arr = new byte[len];
        readFileContents(fileName, arr);
        // calculate checksum and return it
    }
}
```

- Arr hangs around and never gets removed or shrunk

Weak references

A WeakReference is similar to a SoftReference, but is more aggressively cleaned up by the garbage collector.

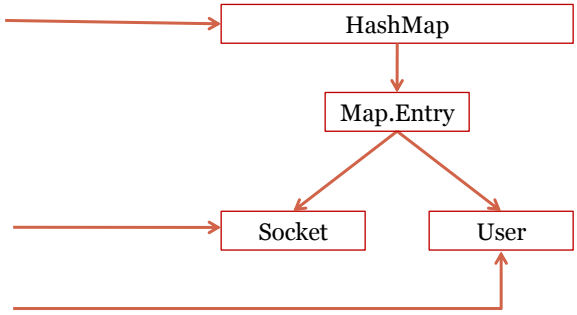
- Usually collected even if there's plenty of memory left.
- Useful for maintaining metadata associated with long-lived objects, to prevent memory leaks.

Metadata example

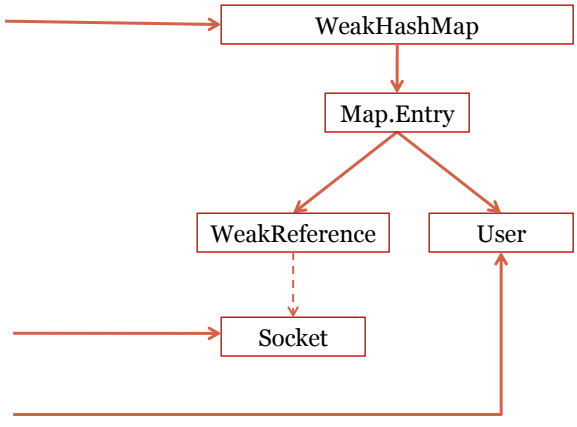
```
private static Map<Socket,User> m = new HashMap<Socket,User>();

Socket s = openSocket(...);
User u = readUserFromSocket(s);
m.put(s, u);

... somewhere else in program ...
User u = m.get(s);
// do work
s.close();
// s and u go out of scope
```

Can we collect the Socket and User?

Metadata example



Solution: WeakHashMap



Reference queues

It would be nice if we could find out when the GC had collected a referent…

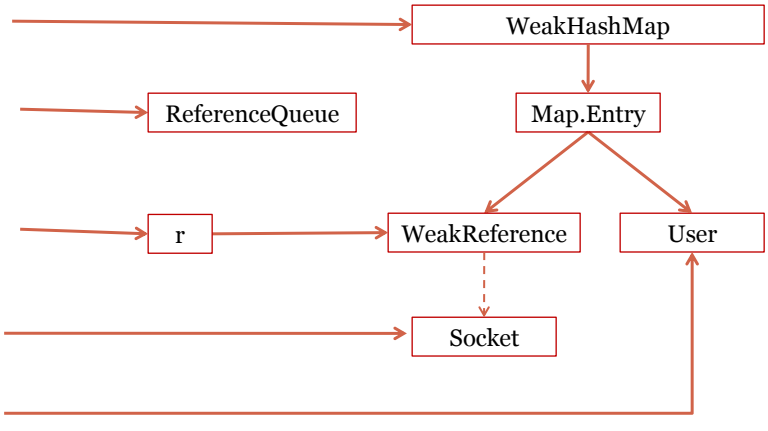
A reference queue is associated with a reference when it is constructed.

(Some time) after the GC collects a referent, it places the corresponding reference object on the queue

- not the referent – why not?

By periodically checking the reference queue, you can see which objects have been collected and need cleaning up after.

Reference queues



Phantom references

A PhantomReference is used to keep track of objects *after* they have been finalized, but before they have been reclaimed.

- The get() method always returns null.
- Gives the programmer more control over object cleanup than finalizers.
- Tells the programmer exactly when an object is about to be reclaimed.
- Must call clear() afterwards to allow referent to be collected.

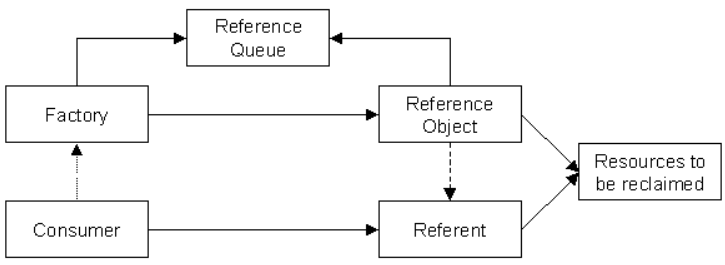
Problems with finalizers

- If an object provides a custom finalizer, it must take at least two garbage collection cycles before collection (in case it was resurrected by the finalizer).
  - possibly more if the finalizer is not run straight away
- The finalizer itself might be slow.
- Hence the program might run out of memory even if there are objects that could be collected.

Slow finalizer example

```
public class SlowFinalizer {
    public static void main(String[] argv) throws Exception {
        while (true) {
            System.out.print(".");
            Object foo = new SlowFinalizer();
        }
    }

    // some member variables to take up space
    byte[] foo = new byte[1000000];

    // and the finalizer, which does nothing but take time
    protected void finalize() throws Throwable {
        try { Thread.sleep(500L); }
        catch (InterruptedException ignored) {}
        super.finalize();
    }
}
```

Using phantom references



- The program poll()s the reference queue until a phantom is enqueued.
- Phantom reference keeps separate pointer to the referent's resources (why?) and cleans them up.
- Finally, it clear()s the reference object so the GC can collect the referent.
- A strong reference must also be held to the reference object itself (why?)

- The referent cannot be accessed through the phantom, so the phantom must have another way to access the resources (usually through a subclass).
- Don't want the reference object itself to get collected before it is placed on the queue.

Exercises

- Implement WeakHashMap, including reference queue
- Read up on the format of .jar files and write a jar ClassLoader that implements what `java -jar` does.

Reference object recap

There are three types of Java reference objects:

- Soft reference – may be reclaimed by collector
- Weak reference – must be reclaimed
- Phantom reference – about to be reclaimed

[java –Xincgc MemoryTest3,4]

```
for (int id = 0; true; id++) {
    MemoryBlock mb = new MemoryBlock(id, size);
    phantoms.add(new MyPhantomReference(mb, refqueue));
    blocks.add(new WeakReference<MemoryBlock>(mb));

    System.out.print("Current blocks: [");
    for (Reference r : blocks)
      System.out.print(r.get() + ", ");

    while ((mpr = (MyPhantomReference) refqueue.poll()) != null) {
        System.out.println("Reclaiming: block " + mpr.id);
        mpr.clear();
    }
}

static class MyPhantomReference extends PhantomReference<MemoryBlock> {
  public int id;
  public MyPhantomReference(MemoryBlock mb, ReferenceQueue<…> rq {
            super(mb, rq);
            this.id = mb.id;
        }
    }
```

Memory profiling

The `jconsole` tool can be used to monitor memory usage, among other things.

[jconsole MemoryTest3,4]

HotSpot memory management

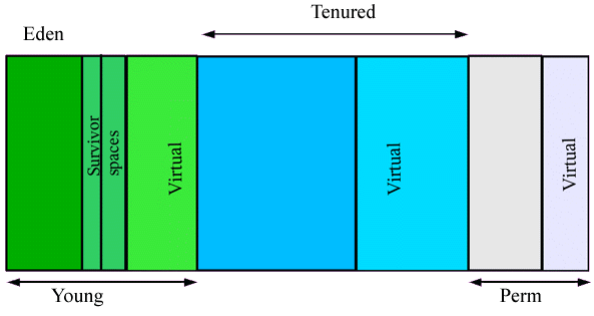
Memory pools in the HotSpot JVM:

- Eden Space (heap): The pool from which memory is initially allocated for most objects.
- Survivor Space (heap): The pool containing objects that have survived the garbage collection of the Eden space.
- Tenured Generation (heap): The pool containing objects that have existed for some time in the survivor space.
- Permanent Generation (non-heap): The pool containing all the reflective data of the virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas.
- Code Cache (non-heap): The HotSpot Java VM also includes a code cache, containing memory that is used for compilation and storage of native code.

HotSpot generations

HotSpot uses a generational garbage collector.

- When a generation gets full, the VM performs a minor GC, and promotes surviving objects.
- Once the tenured generation fills, the VM performs a full GC.




```
public static void main(String[] args) throws Exception {
  MyWeakHashMap<…> map = new MyWeakHashMap<Integer,MemoryBlock>();
  ReferenceQueue<…> refqueue = new ReferenceQueue<MemoryBlock>();
  List<…> phantoms = new ArrayList<PhantomReference<MemoryBlock>>();
  int size = 100000000;

  MyPhantomReference mpr;
  for (int id = 0; true; id++) {
    Integer i = new Integer(id);
    MemoryBlock mb = new MemoryBlock(id, size);
    phantoms.add(new MyPhantomReference(mb, refqueue));
    System.out.println("Putting " + i + ": " + mb);
    map.put(i, mb);
    System.out.println("Got " + i + ": " + map.get(i));
    System.out.println("Map contents: " + map.values());
    Thread.sleep(1000L);

    while ((mpr = (MyPhantomReference) refqueue.poll()) != null) {
        System.out.println("Reclaiming: block " + mpr.id);
        mpr.clear();
    }
```

WeakHashMap implementation

```
public class MyWeakHashMap<K,V> {
    private Map<…> delegate = new HashMap<Reference<K>,V>();
    private ReferenceQueue<K> refqueue = new ReferenceQueue<K>();

    public MyWeakHashMap() {}
    public V put(K key, V value) {
        expungeStaleEntries();
        return delegate.put(key, value);
    }
    public V get(Object key) {
        expungeStaleEntries();
        return delegate.get(key);
    }
    public Collection<V> values() {
        return delegate.values();
    }
```

Class unloading

For applications that load a large number of classes, class unloading can save memory.

When can a class be safely reclaimed?

If a custom classloader is itself reclaimed, no new instances of its classes can be created. However what if another classloader loads the same code for those classes?

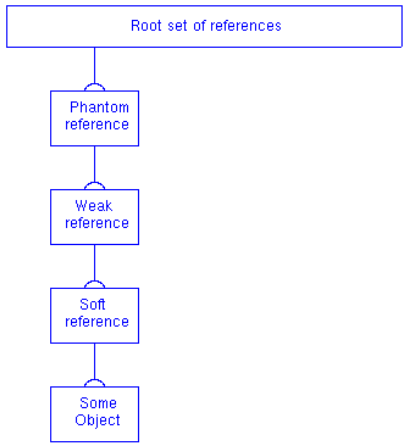
Class unloading

- Only if no further instances can be created – otherwise static state might be lost
- A class loaded from a different classloader is considered a different class even if the code is the same.

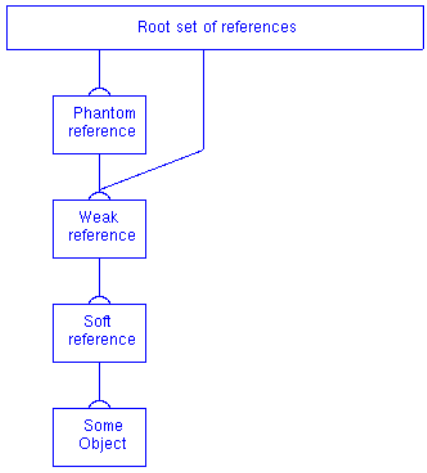
Mixed reference chains

- A chain of references may contain more than one type of reference in it.
- The reachability of an object is the *weakest link* on the *strongest chain* between it and the root.

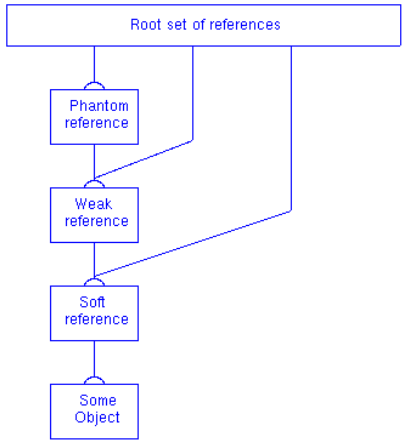
Reference chains



Reference chains (2)



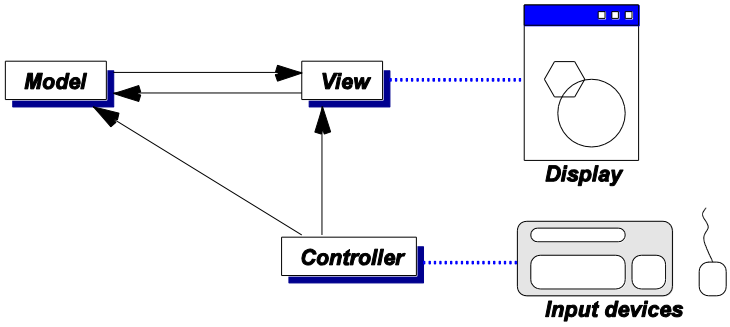
Reference chains (3)



Graphical user interfaces

- The original set of GUI classes in Java was the Abstract Windowing Toolkit (AWT).
- In AWT, each component has a peer in the native windowing system that is responsible for its rendering.
- These have been largely superseded for direct use by the lightweight Swing components and widgets, although Swing depends on AWT in several ways.
- Swing components follow a *model-view-controller* pattern.

Model-view-controller pattern



- Multiple views may be based on the same model (e.g. a table of numbers and a graphical chart). This separation allows views to be changed independently of application logic.

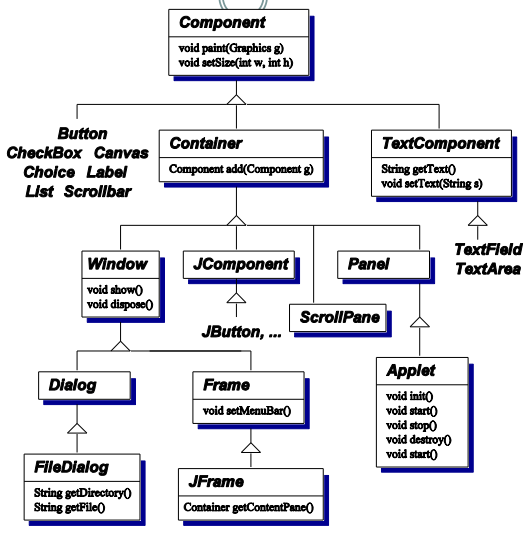
Drawing graphics

- Basic rendering primitives are performed through instances of the Graphics class, e.g.:

```
public class E1 extends java.applet.Applet {
  public void paint(java.awt.Graphics g) {
    g.drawLine(0,0,100,100);
  }
}
```

- Simple primitives are available—setColor, copyArea, drawLine, drawArc...
- More abstractly, an instance of Graphics represents the component on which to draw, a translation origin, the clipping mask, the font, etc.
- Translation allows components to assume that they're placed at (0,0).

Component hierarchy



Components

- Java graphical interfaces are built up from *components* and *containers*.
- Components represent the building blocks of the interface: for example, buttons, check-boxes, text boxes, etc.
- Each kind of component is modelled by a separate Java class (JButton, JList, JTable, etc).
- Each instance of a class handles an occurrence of it on the screen – e.g. to create a button bar the programmer would instantiate several JButtons.
- New component types can be created by sub-classing existing ones—e.g. by sub-classing JPanel (a blank, rectangular area of the screen) to define how that component should be rendered by overriding its paintComponent method:

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  ...
}
```

Containers

- Containers are a special kind of component that can contain other components — java.awt.Container extends java.awt.Component.
- Containers implement an add method to place components within them.
- Containers also provide top-level windows—for example javax.swing.JWindow (a plain window, without a title bar or borders) and javax.swing.JFrame (a 'decorared' window with a title bar, etc.)
- Other containers allow the programmer to control how components are organized — in the simplest case javax.swing.JPanel.

Input

- Input is delivered using an event mechanism (Observer pattern).
- Different kinds of event are represented by subclasses of java.awt.AWTEvent. For example, MouseEvent, KeyEvent, etc.
- Components provide methods for registering listeners, e.g. addMouseListener

```
public class JButton extends AbstractButton {
  public void addMouseListener(MouseListener l) {
    ...
  }
}
```

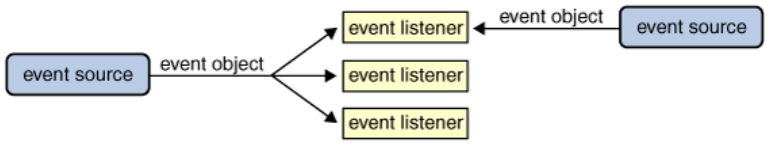
Input (2)

- The system delivers events by invoking methods on a listener:

```
public interface MouseListener extends EventListener {
  public void mouseClicked(MouseEvent e);
  ...
}
```

- AWTEvent has a getSource() method to distinguish events from different sources.
- Subclasses add methods to obtain other details—e.g. getX() and getY() on a MouseEvent.

Listeners



- A source can have multiple listeners associated with it.
- A listener can also listen to events from multiple sources.

Events

All components can generate:

- ComponentEvent when resized, moved, shown, or hidden;
- FocusEvent when it gains or loses the focus;
- KeyEvent when a key is pressed or released;
- MouseEvent when mouse buttons are pressed or released;
- MouseMotionEvent when the mouse is dragged or moved.

- Containers can generate ContainerEvent when components are added or removed.
- Windows can generate WindowEvent when opened, closed, iconified, etc.