Reflection

Review

What does an ObjectStream do?

a) Reads/writes objects as text files

b) Reads/writes objects as binary files

c) Reads/writes objects as random access files

d) Reads/writes objects in CSV format

Review

What advantage do binary files of homogeneous data have?

a) They are easy to change in a text editor

b) They are the most commonly used data file types

c) They can be processed by accessing records as you would an array

d) They are easier to create, write to, and read from than text files

Objectives

- Topics
  - Compile-time class information
  - Run Time Type Information (RTTI)
- Goals: after this lecture, you will be able to
  - query the type of a class and get other information about it
  - query the type of an "unknown" class, gather information about it, and call its methods

Class

- When you have the .java file for a class, there's not much mystery about what's in the class
- But you can still ask questions about the contents of the class in a generic way
- This is called ***reflection***
- And a class can be cast to other compatible types if needed

Class, cont.

- `Class.forname(String)` returns a class containing information about the class named in the parameter
  - Throws `ClassNotFoundException`
  - Other methods in these notes throw other exceptions – let IntelliJ handle it
- Some of its methods are:
  - `getName( )`, `getCanonicalName( )` – includes package(s)
  - `simpleName( )`
  - `isInterface( )`, `getInterfaces( )`
  - and many others: see later slides

Class, cont.

```
Class myclass = Class.forName("Employee");
// Assumes Employee.java is present

String name = myclass.getName();
boolean inter = myclass.isInterface( );
boolean b = myclass.isEnum();
System.out.println(name + " is interface? " + inter +
   " is enum? " + b);
```

Casting

- We've already seen upcasting and downcasting for inheritance hierarchies
    - upcasting by default in a Factory Method

```
public Note createNote(String type) {
   if (type.equals("Memo") { return new Memo( ); }
   …
```

Casting, cont.

- downcasting to try to use the methods specific to a child class not present/overridden in the parent class

```
Note note = Factory.createNote(type);
( (Memo)note).uniqueToChildMethod( );
```

- But recall this is unsafe: the class may not convert correctly or have that method. So use `instanceof`:

```
if (note instanceof Memo) {
   ( (Memo)note).uniqueToChildMethod( );
}
```

RTTI

- This is all fine, but again, if you have the .java file, you know everything
- What if you don't? What if you download a .class file from somewhere instead?
- Java keeps track of all the same information there, too
- The difference is you can query that information at ***runtime*** instead of compile time – that is, dynamically
- This is called ***Run Time Type Information (RTTI)***

RTTI, cont.

- To use it, the .class file must be in the out/production/<yourProject> directory
  - Or: use a class loader
- Start with forName
  - But use Class<?>
- Then you can query the other information
- Start with something like:

```
Class<?> myclass = Class.forName("Employee");
// Assumes Employee.class is present
```

RTTI Class Information

- Get its member data fields, of type Field, with

```
Field[] f = myclass.getDeclaredFields();
```

  - Then you can iterate over the array

- Similarly, get its constructors, of type Constructor, with

```
Constructor[] c =
     myclass.getDeclaredConstructors( );
```

  - and iterate over the array

RTTI Class Information, cont.

- Then get its methods, of type Method, with

```
Method[] m = myclass.getDeclaredMethods( );
```

  - Again, iterate over the array
- There are also corresponding methods without the word "Declared" in them that return *all* public members, constructors, and methods, even inherited ones

RTTI Class Information, cont.

- To get the method parameters, use:

Note: a method variable

```
Types[] t = m[0].getParameterTypes( );
// for example - any m[i] can be used
```

- Get the return types of methods, with

```
Class<?> r = m[0].getReturnType( );
```

Using RTTI Information

- Now you can declare a class using

```
Object o = myclass.getDeclaredConstructor().newInstance();
// Default constructor
```

- Then you can call methods using (choosing method #0) on object o

on the object you created

```
m[0].invoke(o);
```

- If you need to give it parameters (choosing method #1)

```
m[1].invoke(o, param1, …);
```

just an example

RTTI, cont.

- Note that the Method[ ] array does not always return the methods in the same numerical order
  - So m[0] and m[1] on the previous slide might be different methods in the next run of the program for the same .class file
  - You can instead find a method by name by searching through the array, looking for a method with a specific name, like "getSalary"

RTTI, cont.

- Or: you could present the user with a menu of the methods and let them choose
- Then present the parameter types and prompt for some values
- Then use invoke( ) to call the method

Example

```
Class<?> myclass = Class.forName("Employee");
// Assumes Employee.class has been loaded

Object o =
myclass.getDeclaredConstructor().newInstance();
Methods[] m = myclass.getDeclaredMethods();
// Print them, along with their parameter types
// Let the user choose one – say #0, with one
// int parameter, then call it:
m[0].invoke(o, 5);
```