Java: Reflection, Cloning

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 12, 16 February 2023

Reflection

Wikipedia

Reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour.

- Two components involved in reflection
  - Introspection

    A program can observe, and therefore reason about its own state.
  - Intercession

    A program can modify its execution state or alter its own interpretation or meaning.

Reflection in Java

- Simple example of introspection

```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

- What if we don't know the type that we want to check in advance?
- Suppose we want to write a function to check if two different objects are both instances of the same class?

```
public static boolean classequal(Object o1, Object o2){
  ...
  // return true iff o1 and o2 point to objects of same type
  ...
}
```

Reflection in Java . . .

```
public static boolean classequal(Object o1, Object o2){...}
```

- Can't use `instanceof`
  - Will have to check across all defined classes
  - This is not even a fixed set!
- Can't use generic type variables
  - The following code is syntactically disallowed

    ```
    if (o1 instance of T) { ...}
    ```

Introspection in Java

- Can extract the class of an object using `getClass()`

- Import package `java.lang.reflect`

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

- What does `getClass()` return?

- An object of type `Class` that encodes class information

The class `Class`

- A version of `classequal` the explicitly uses this fact

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

- For each currently loaded class `C`, Java creates an object of type `Class` with information about `C`
- Encoding execution state as data — reification
  - Representing an abstract idea in a concrete form

Using the Class object

- Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
  // Create a new object of same type as obj
...
```

- Can also get hold of the class object using the name of the class

```
...
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
...
```

- ..., or, more compactly

```
...
Object o = Class.forName("Manager").newInstance();
...
```

The class `Class` . . .

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- Constructors, methods and fields themselves have structure
  - Constructors: arguments
  - Methods : arguments and return type
  - All three: modifiers `static`, `private` etc
- Additional classes `Constructor`, `Method`, `Field`
- Use `getConstructors()`, `getMethods()` and `getFields()` to obtain constructors, methods and fields of `C` in an array.

The class `Class` ...

- Extracting information about constructors, methods and fields

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
...
```

- `Constructor`, `Method`, `Field` in turn have functions to get further details

The class Class ...

- Example: Get the list of parameters for each constructor

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
  Class params[] = constructors[i].getParameterTypes();
  ..
}
```

- Each parameter list is a list of types
  - Return value is an array of type Class[]

The class Class ...

- We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }
  // construct an array  of arguments
methods[3].invoke(obj,args);
  // invoke methods[3] on obj with arguments args
...
Field[] fields = c.getFields();
Object o =  fields[2].get(obj);
   // get the value of fields[2] from obj
...
fields[3].set(obj,value);
  // set the value of fields[3] in obj to value
...
```

Reflection and security

- Can we extract information about private methods, fields, . . . ?
- `getConstructors()`, . . . only return publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`
- Should this be allowed to all programs?
- Security issue!
- Access to private components may be restricted through external security policies

Using reflection

- BlueJ, a programming environment to learn Java
- Can define and compile Java classes
- For compiled code, create object, invoke methods, examine state
- Uses reflective capabilities of Java — BlueJ need not internally maintain “debugging” information about each class
- See http://www.bluej.org

Limitations of Java reflection

- Cannot create or modify classes at run time
  - The following is not possible

    ```
    Class c = new Class(....);
    ```

  - An environment like `BlueJ` must invoke Java compiler before you can use a new class
- Contrast with Python
  - `class XYZ:` can be executed at runtime in Python
- Other OO languages like Smalltalk allow redefining methods at run time

Erasure of generic information

- Type erasure — Java does not keep record all versions of `LinkedList<T>` as separate types
  - Cannot write

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```
- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Or, the upper bound, if one is available
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`
- Since no information about `T` is preserved, cannot use `T` in expressions like

  ```
  if (o instanceof T) {...}
  ```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

- As a consequence the following overloading is illegal

```
public class Example {
    public void printlist(LinkedList<String> strList) { }
    public void printlist(LinkedList<Date> dateList) { }
}
```

- Both functions have the same signature after type erasure

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;           // OK
newarray = new T[100]; // Cannot create!
```

- An ugly workaround . . . generates a compiler warning but works!

```
T[] newarray;
newarray = (T[]) new Object[100];
```

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .
- Wrapper class for each basic type:

Basic type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`

Basic type	Wrapper Class
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

- All wrapper classes other than `Boolean`, `Character` extend the class `Number`

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...

- Autoboxing — implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

- Use wrapper types in generic data structures

Copying an object

- Normal assignment creates two references to the same object
  - Updates via either name update the object
- What if we want two separate but identical objects?
  - e2 should be initialized to a disjoint copy of e1
- How does one make a faithful copy?

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1;
e2.setname("Eknath");  // e1 also updated
```

The clone() method

- Object defines a method clone()
- e1.clone() returns a bitwise copy of e1
- Why a bitwise copy?
  - Object does not have access to private instance variables
  - Cannot build up a fresh copy of e1 from scratch
- What could go wrong with a bitwise copy?

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

Shallow copy

- What if we add an instance variable Date to Employee?
  - Assume update() updates the components of a Date object
- Bitwise copy made by e1.clone() copies the reference to the embedded Date
  - e2.birthday and e1.birthday refer to the same object
  - e2.setbday() affects e1.birthday
- Bitwise copy is a shallow copy
  - Nested mutable references are copied verbatim

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){
    name = n;
  }

  public void setbday(int dd, int mm, int yy){
    birthday.update(dd,mm,yy);
  }
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 name not updated
e2.setbday(16,4,1997); // e1 bday updated!
```

Deep copy

- Deep copy recursively clones nested objects
- Override the shallow `clone()` from `Object`
- `Object.clone()` returns an `Object`
  - Cast `super.clone()`
- `Employee.clone()` returns an `Employee`
  - Allowed to change the return type

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){
    Employee newemp =
          (Employee) super.clone()
    Date newbday = birthday.clone();
    newemp.birthday = newbday;
    return newmp;
  }
}
```

Deep copy . . .

- What if Manager extends Employee?
- New instance variable promodate
- Manager inherits deep copy clone() from Employee
- However Employee.clone() does not know that it has to deep copy promodate!
- Cloning is subtle, so Java puts in some restrictions

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}

public class Manager extends Employee {
  private Date promodate;
  ...
}
```

Restrictions on clone()

- To allow clone() to be used, a class has to implement Cloneable interface
  - Marker interface

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

Restrictions on clone()

- To allow clone() to be used, a class has to implement Cloneable interface
  - Marker interface
- clone() in Object is protected
  - Only Employee objects can clone()
- Redefine clone() as public to allow other classes to clone Employee
  - Expanding visibility from protected to public is allowed

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}
```

Restrictions on clone()

- To allow `clone()` to be used, a class has to implement `Cloneable` interface
  - Marker interface
- `clone()` in `Object` is `protected`
  - Only `Employee` objects can `clone()`
- Redefine `clone()` as `public` to allow other classes to clone `Employee`
  - Expanding visibility from `protected` to `public` is allowed
- `Object.clone()` throws `CloneNotSupportedException`
  - Catch or report this exception
  - Call `clone()` in `try` block

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone()
    throws CloneNotSupportedException {...}
}
```