Week 7

Brendon Faleiro

Recap:
Interface vs. Abstract Class

Interface

- Methods can be declared but No method bodies
- Constant can be declared
- Has no constructors
- Multiple inheritance possible
- Has no top interface
- Multiple “parent” interfaces

Abstract class

- Methods can be declared and method bodies can be defined
- All types of variables can be declared
- Can have constructors
- Multiple inheritance not possible
- Always inherits from Object
- Only one parent class

Recap: Inheritance vs SubTyping

Inheritance

- Used to reuse code from one class in another.
- Works with method and variable definitions.
- Relation between classes.
- Achieved by using the extends keyword (in java).

Subtyping

- Used to pass one type in place of another.
- Works with API intefaces.
- Relation between types.
- Achieved by using the extends or implements keyword.

Static Overloading vs Dynamic dispatch

- Every method call is dynamically dispatched in Java.
- Dynamic dispatch looks at calling objects.
- Since a subtype object can be passed instead of a parent type, the object in hand could belong to either the parent type or any of its subtype.
- The method called on the object is always the one that is defined closest to its actual type.
- Overriding is a result of dynamic dispatch.

- Within a class, the parameters are disambiguated using Static Overloading.

```java
class A {
 void callme() {
   System.out.println("Inside A's callme method");
 }
}

class B extends A {
 void callme() {
   System.out.println("Inside B's callme method");
 }
}

class C extends A {
 void callme() {
   System.out.println("Inside C's callme method");
 }
}
```

```java
class Dispatch {
 public static void main(String args[]) {
  A a = new A(); // object of type A
  B b = new B(); // object of type B
  C c = new C(); // object of type C
  A r; // obtain a reference of type A

  r = a; // r refers to an A object
  r.callme(); // calls A's version of callme

  r = b; // r refers to a B object
  r.callme(); // calls B's version of callme

  r = c; // r refers to a C object
  r.callme(); // calls C's version of callme
 }
}
```

- Example 2 (Java test code)

Understanding Java’s Memory model

- Objects always are on the heap.
- Variables are on the stack.
- Variables can only store references (pointers) to objects.
- The dot (.) operator is actually a pointer dereference to access object fields.

- C c1 = new C();
  - Here c1 is a pointer to an object on the heap.
- C c2 = c1;
  - Now c2 points to the exact same object that c1 points to.
- Now if I make any change to c1, c2 has the same change (and vice versa)!
- The only way to create a copy of the old object is to create a new object
  - using the “new” keyword with a constructor that is passed an older object of the same type.
  - the clone function.

```java
class Int {
  public int val;

  public Int(int val) {
    this.val = val;
  }

  void swap_with_1(Int j) {
    int value = this.val;
    this.val = j.val;
    j = new Int(value);
  }

  void swap_with_2(Int j) {
    Int j_copy = j;
    j.val = this.val;
    this.val = j_copy.val;
  }

  void swap_with_3(Int j) {
    this.val += j.val;
    j.val = this.val - j.val;
    this.val -= j.val;
  }}
```

i=new Int(123);   j=new Int(3456);

a) i.swap_with_1(j);

Ans: i.val =3456, j.val =3456

b) i.swap_with_2(j);

Ans: i.val = 123, j.val = 123

c) i.swap_with_3(j);

Ans: i.val = 3456, j.val = 123

Understanding Java Primitives

- All Java primitives have wrapper classes that can convert the primitive to the equivalent class.
  - int -> Integer
  - double -> Double
- All generic classes use the reference classes for type parameters rather than the primitives.
- The primitives can be converted to the wrapped objects and vice-versa by auto-boxing and auto-unboxing.

- == : referential equivalence
- .equals() : value equivalence/logical equivalence

Let l be a list with elements 3 -> 4 -> 5. get(i) returns the ith element of the list.

- Q: System.out.println(3 == l.get(0).intValue())

-> true

- Q: System.out.println(3 == l.get(0))

-> true

- Q: System.out.println(new Integer(3) == l.get(0))

-> false

- Q: System.out.println(new Integer(3).equals(l.get(0)))

-> true

- Q: System.out.println(l.get(0).equals(l.get(0)))

-> true

Java’s Parametric Polymorphism (Generics)

- List without Generics:

```
interface myList{
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}


class myListImpl implements myList{…}
```

```
myList l = new myListImpl();
l.add(“hi”);
String s = l.get(0);
```

Compiler complains about Object and String not being the same type!

```
myList l = new myListImpl();
l.add(“hi”);
String s = (String)l.get(0);
```

Compiler happy!

- List without Generics:

```
interface myList{
   boolean contains(Object o);
   void add(Object o);
   Object get(int i);
}

class myListImpl implements myList{…}
```

```
myList l = new myListImpl();
l.add(“hi”);
Integer s = (Integer) l.get(0);
```

Compiler is happy!

But at runtime the cast fails!

Hence Generics!

- List without Generics:

```
interface myList<T>{
  boolean contains(T o);
  void add(T o);
  T get(int i);
}

class myListImpl implements myList{…}
```

```
myList l = new
myListImpl<String>();
l.add(“hi”);
l.add(1);
String s = l.get(0);
```

Does not compile as Integers and Strings are not the same type

So Generics gives us:

- Stronger type checks at compile time.
- If the generic version type checks, then all instantiations will work perfectly!
- No need to cast clients.
- Programmers can write generic code!

Type Erasure

- Explicit type annotation are removed from a program

```
ArrayList<Integer> li = new ArrayList<Integer>();
ArrayList<Float> lf = new ArrayList<Float>();
System.out.println(li.getClass()); // class java.util.ArrayList
System.out.println(lf.getClass()); // class java.util.ArrayList
if (li.getClass() == lf.getClass()) { // evaluates to true
  System.out.println("Equal");
}
```

Java’s first class functions

Java doesn't technically have first-class functions.

Java can simulate first-class functions to a certain extent, with anonymous classes and generic function interface.

Understanding the example from class:

```
class Sort{
	public static void main(String[] args){
		List<String> l = new LinkedList<String>(args);
		Collections.sort(l);
		for(String s : l)
			System.out.println(s+ “ ” );
	}
}
```

This is the default sort.

Writing my own compare

Method 1:

```
• class Reverse implements Comparator<String>{
        public int compare(String s1, String s2){
                return s2.compareTo(s1)
        }
}

Collections.sort(l, new Reverse());
```

Method 2:

```
Collections.sort(l, new Comparator<String>() {
        public int compare(String s1, String s2)
                {return s2.compareTo(s1);}
        });
```

Method 3:

```
Collections.sort(l, (String s1, String s2)-> s2.compareTo(s1));
```

Method 4:

```
Collections.sort(l, (s1, s2)-> s2.compareTo(s1));
```

Peek into next week: Parallelization

- **Automatic parallelization**, also **auto parallelization**, **autoparallelization**, or **parallelization**, the last one of which implies automation when used in context, refers to converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine.

- The goal of automatic parallelization is to relieve programmers from the hectic and error-prone manual parallelization process.

- Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.

- Your operations must be state independent and associative in order to achieve parallelism.

```
public class SumStream{
	public static void main(String[] args){
		int size = Integer.parseInt(args[0]);
		int[] a = new int[size];

		for (int i=0; i<size; i++)
			a[i]=i;

		int sum = Arrays.stream(a).reduce(0, (i1,i2) -> i1+i2);
		// to parallelize :
		// int sum = Arrays.stream(a).parallel().reduce(0, (i1,i2) -> i1+i2);

		System.out.println(sum);
	}
}
```