CS2030 Programming Methodology II

Lecture VI

Shengdong Zhao
Spring 2019

Acknowledge: slides are adapted from Henry Chia

Lecture Outline

- **Abstraction principle**
- **Java Collection example: ArrayList**
- **Generics**
  - Generic classes
  - Sub-typing
    - Wildcards
    - PECS
  - Generic methods
- **Java Collections Framework**
  - Collection / List interfaces
  - Comparator functional interface

Let's start with an example: Super Sort

```
public class SuperSort {


   public int[] superSort(int[] array){
      //... sort implemenation ...
      return array;
   }
}
```

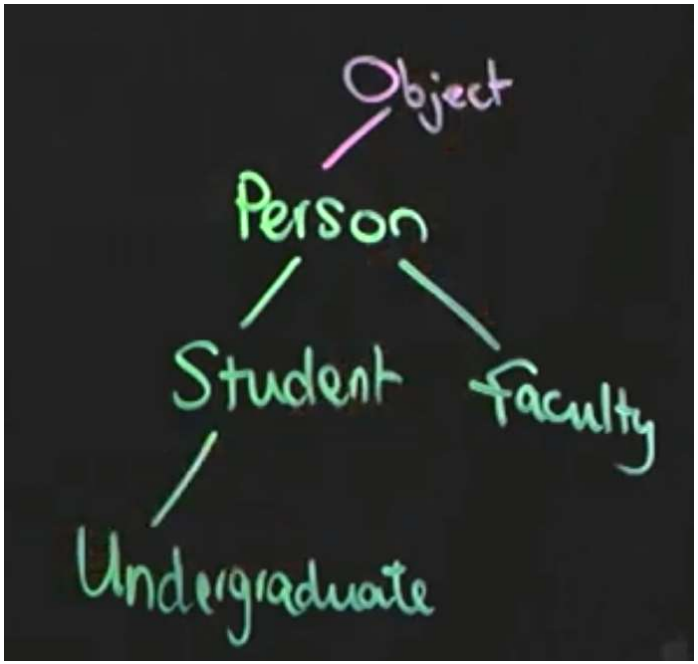
- **Limitations:** this super sorting algorithm only works for int, but what about other types (e.g., double, float, String, person, etc.) that can also use this algorithm?

Solution 1: Use Inheritance

```
//We can redefine super sort based on objects
public class SuperSort {
  public Object[] superSort(Object[] array){
        //... sort implemenation ...
        return array;
  }
}

public static void main(String[] args) {
  SuperSort ss = new SuperSort();
  Object[] objectArray = new Object[10];
  ss.superSort(objectArray);
  String[] stringArray = new String[10];
  ss.superSort(stringArray);

  //however, this solution is not type safe
  Object[] objectArray = new Object[10];
  objectArray[0] = "Hello world"
  objectArray[1] = new Point(0, 0);
  ss.superSort(objectArray); //runtime error!
}
```



diff types of object?

Introducing Java Generics

- Generics
  - a.k.a. Parameterized Types (passing the type as a parameter to the class and method definition)

specify types of object

```
public class SuperSort <T> {
  public T[] superSort(T[] array){
        //... sort implemenation ...
        return array;
  }
}

public static void main(String[] args) {
  SuperSort<String> ss = new SuperSort<String>();
  Object[] objectArray = new Object[10];
  ss.superSort(objectArray); //not allowed
  String[] stringArray = new String[10];
  ss.superSort(stringArray); //OK
}
```

Using Generics

```
//Parameterized type can be used in class or methods. Below
are 4 ways a parameterized type can be used in a class.
Note use it in static method is not allowed
public class ExampleOne<T> {
  //1) Use parametered types to declare a class variable
  private T value; // 1)
  //2) Use parametered type to define a return type
  public T getValue(){ // 2)
    return value;
  }
  //3) Use parametered type to define a method variable
  //4) Use it to define a local variable
  public void setValue(T value){ // 3)
    T temp = value; // 4)
    this.value = temp;
  }
  //Use parameterized type for a method (static ok)
  public static <Z> Z noOp(Z val){
    return val;
  }
}
```

non static because if create new list with diff parameter T gts confused

Generics in Java

- Generic classes: classes that allow some type parameter
- Convention: T for type; E for element; K for key; V for value
- Diamond notation <> lets the compiler infer the element type from the declaration; the following is equivalent ArrayList<Circle> numbers = **new** ArrayList<>();
- Some commonly used methods of ArrayList include:

Collections and the ArrayList

void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	add(E e)	Appends the specified element to the end of this list.
void	clear()	Removes all of the elements from this list.
boolean	contains(Object o)	Returns true if this list contains the specified element.
E	get(int index)	Returns the element at the specified position in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty()	Returns true if this list contains no elements.
E	remove(int index)	Removes the element at the specified position in this list.
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
void	trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

- Methods specified in interface Collection<E>
  - size, isEmpty, contains, add(E), remove(Object), clear
- Methods specified in interface List<E>
  - indexOf, get, set, add(int, E), remove(int),

Auto-boxing and Unboxing

- Only reference types allowed as type arguments; primitives need to be auto-boxed/unboxed, e.g. ArrayList<Integer>

```
jshell> ArrayList<Integer> numbers = new ArrayList<>()
numbers ==> []
jshell> numbers.add(1)
$4 ==> true
jshell> numbers.add(0, 2)
$5 ==> true
jshell> for (int i : numbers) System.out.println(i * 10)
20
10
```

- Placing an int value into ArrayList<Integer> causes it to be **auto-boxed**
- Getting an Integer object out of ArrayList<Integer> causes the int value inside to be **(auto-)unboxed**

Generics Behind the Scene

Can we implement the following three methods in the same class?

```
//Draw method 1
void draw(ArrayList list){
  System.out.println("draw a raw arraylist");
}

//Draw method 2
void draw(ArrayList<Object> list){
  System.out.println("draw an arraylist of objects");
}

//Draw method 3
void draw(ArrayList<String> list){
  System.out.println("draw an arraylist of String");
}
```

The answer is no, but why?

Generics Behind the Scene

- What happens if execute the following statements

```
ArrayList rawList = new ArrayList();
ArrayList<Object> objectList = new ArrayList<>();
ArrayList<String> stringList = new ArrayList<>();

System.out.println("RawList's class type is " +
rawList.getClass());
System.out.println("ObjectList's class type is " +
objectList.getClass());
System.out.println("StringList's class type is " +
stringList.getClass());
```

- Here is the answer:

```
> RawList's class type is class java.util.ArrayList
> ObjectList's class type is class java.util.ArrayList
> StringList's class type is class java.util.ArrayList
```

- Why the classnames are the same?

Generics Behind the Scene

Original code:

```
ArrayList<String> stringList = new ArrayList<>();
stringList.add("Hello world");
String aString = stringList.get(0);
```

Replaced code:

```
ArrayList stringList = new ArrayList();
stringList.add("Hello world");
String aString = (String) stringList.get(0);
```

Note that after compilation, the code above is converted into the code below, so the runtime environment does not know the "ArrayList<String>" type, it only knows the "ArrayList" type. This is called Type Erasure.

Aha Moments

- Now you should know why the classname print out for the different lists are the same
  - Because java runtime does not know anything about type of the objects a container holds, it is only used by the compiler to ensure code consistency
- For the same reason, you should also know why the three different types of draw methods listed earlier can not co-exist in the class at the same time
  - Because to the Java runtime, the signature of these three methods are exactly the same
- Now, what's the implication on inheritance and polymorphism?
  - Can we declare "List<Object> list = new ArrayList<String>();"?
  - The answer is No, not in a straight-forward way, since java runtime does not know about the parameterized types.

What about Inheritance?

- We know in Java, a variable can represent anything that belongs to its type or its subtypes.
  - Shape s = new Circle(); //valid as Circle is a child of shape
- What about generics?
  - List<Shape> list = new ArrayList<Shape>(); *//valid, as Arraylist is a child type of List*
  - List<Shape> list = new ArrayList<Circle>(); *//no longer valid, but why?*
    - Hint: remember the runtime knows nothing about whether the list is shape or circle, it only knows that it is an arraylist. If we allow the above statement, what kind of problem will it cause?
    - Imagine passing an ArrayList of Circle into the function below:

```
void draw(List<Shape> list){
  list.add(new Rectangle()); //but I can add rectangles, which is wrong
  for(Shape s: list){ s.draw(); }}
```

But since Java don't know this list is circle at runtime, this error will not be caught easily, therefore, Java disallow this behavior.

How can I Declare a Generic Variable with Flexibility

- What if I want to allow a variable with type List<Something> to represent more possible types?
- Java allows you to do that using wildcard "?"
- You can specify
  - List<?> list = new ArrayList<Shape>(); //valid
  - list = new ArrayList<Circle>(); //valid
  - list = new ArrayList<Object>(); //valid
- Question: what happens if I want to add something to the list?
  - list.add(new Circle()); //is it allowed? list.add(new Object()); //is it allowed? list.add(null); //is it allowed?

not allowed

not allowed

allowed

- Question: what happens if I want to get something from the list?
  - list.get(0); //What is the type of the return value? Can I assign it to Shape?

retrieve okay but gives exception

for loop and System.out.println not allowed

Restrict Generic Types

- Wildcard provides too much freedom. How to restrict it?
- Allow only base-type and subtypes (covariance)
  - List<? extends Shape> list = new ArrayList<Circle>();
  - But there is a cost:
    - Not able to add anything to it
- Allow only base-type and supertypes (contra-variance)
  - List<? super Circle> list = new ArrayList<Shape>();
  - But there is a cost:
    - Not able to determine the type of element
- This is summarized as PECS
  - Producer extends (can't consume, but ok to produce), consumer super (can only produce object, but ok to consume)

only add null because object x specified if it is circle list or rectangle list,
if add circle to rectangle list?

for loop ok

adding ok

cannot for loop because (Shape o: list) not allowed
but Object o: list can

Usage examples

Generic Methods

- Consider the following:

```
Integer[] nums = {19, 28, 37};
System.out.println(max3(nums));
```

- Other than using Integer class, can define generic methods

```
public static <T extends Comparable<T>> T max3(T[] nums)
     {
     T max = nums[0];
     if (nums[1].compareTo(max) > 0) {
          max = nums[1];
     }
     if (nums[2].compareTo(max) > 0) {
          max = nums[2];
     }
     return max;
}
```

Java Collections Framework

- Collections contain references to objects (elements) of type <E>, or objects of sub-type of <E>
- Collection-framework interfaces declare operations to be performed generically on various type of collections

Interface	Description
Collection	The root interface in the collections hierarchy from which interfaces Set, Queue and List are derived.
Set	A collection that does not contain duplicates.
List	An ordered collection that can contain duplicate elements.
Map	A collection that associates keys to values and cannot contain duplicate keys.
Queue	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Collection<E> Interface

- Generic interface parameterized with a type parameter E
- toArray(T[]) is a generic method; the caller is responsible for passing the right type
- containsAll, removeAll, and retainAll has parameter type Collection<?>, we can pass in a Collection of any reference type to check for equality
- addAll has parameter declared as Collection <? **extends** E>; we can only add elements that are upper-bounded by E

1 Otherwise, an ArrayStoreException will be thrown

Collection<E> Interface

```
public interface Collection<E>
                extends Iterable<E> {
        boolean add(E e);
        boolean contains(Object o);
        boolean remove(Object o);
        void clear();
        boolean isEmpty();
        int size();
        Object[] toArray();
        <T> T[] toArray(T[] a); //<-
        boolean addAll(Collection<? extends E> c); //<-
        boolean containsAll(Collection<?> c); //<-
        boolean removeAll(Collection<?> c); //<-
        boolean retainAll(Collection<?> c); //<-
        :
}
```

List<E> Interface

- List<E> interface extends Collection<E>
  - For implementing a collection of possibly duplicate objects where element order matters
  - Classes that implement List<E> include ArrayList and LinkedList: List<Circle> circles = new ArrayList<>();
  - circles declared with List<Circle> to support possible future modifications to LinkedList
- List<E> interface also specifies a sort method default void sort(Comparator<? super E> c)
- Interface with default method indicates that List<E> comes with a default sort implementation
  - A class that implements the interface need not implement it again, unless the class wants to override the method

Lecture Summary

- Appreciate higher-level abstraction thinking and design
- Appreciate the use of Java generics in classes and methods
- Understand autoboxing and unboxing involving primitives and its wrapper classes

Lecture Summary

- Understand parametric polymorphism and sub-typing
  - mechanism, e.g. given Burger <: FastFood
  - covariant: Burger[] <: FastFood[]
  - invariant: C<Burger> and C<FastFood>
  - covariant: C<Burger> <: C <? extends FastFood>
  - contravariant: C<FastFood> <: C<? super Burger>
- Appreciate PECS and accompanying notions of upper and lower bound wildcards
- Familiarity with the Java Collections Framework