Programming Language Concepts: Lecture 5

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 5, 02 February 2009

Abstract classes and interfaces

- Use abstract functions to specify common properties
  - Abstract definition of `perimeter()` for all `Shape`s

    ```
    public abstract double perimeter();
    ```

- Classes with abstract functions must themselves be abstract
- Cannot create objects of abstract type ...
- ... but we can define and use variables of abstract type

```
Shape sarr[] = new Shape[3];

Circle c = new Circle(...);   sarr[0] = c;
Square s = new Square(...);   sarr[1] = s;
Rectangle r = new Rectangle(...);  sarr[2] = r;

for (i = 0; i < 2; i++){
  size = sarr[i].perimeter();
}
```

Generic functions

- Use abstract classes to specify capabilities

```
abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s, 0 if this == 0,
    //        +1 if this > s
}
```

- Now we can sort any array of objects that extend `Comparable`

```
class Sortfunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use a[i].cmp(a[j])
  }
}
```

Mutiple inheritance and interfaces

- How do can we sort `Circle` objects?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to "implement" it:

```
class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
   ...
}
```

- Can implement multiple interfaces

Generic programming

- Java's tree-like hierarchy with `Object` at root allows polymorphic functions

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length; i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Generic programming

- Java's tree-like hierarchy with `Object` at root allows polymorphic functions

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length; i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- What if we wanted to swap two objects?

```
public static void swap (Object x, Object y){
  Object temp = x;
  x = y;
  y = temp;
}
```

Generic programming . . .

- What happens if we write the following?

```
Date d = new Date(...);
Circle c = new Circle (...);
..
swap(c,d);
```

Generic programming . . .

- What happens if we write the following?

```
Date d = new Date(...);
Circle c = new Circle (...);
..
swap(c,d);
```

- Type error at run time!

Generic programming . . .

- A generic function to copy arrays

```
public static void arraycopy (Object[] src, Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Generic programming . . .

- A generic function to copy arrays

```
public static void arraycopy (Object[] src, Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

- Given the following definitions

```
Ticket[] ticketarr = new Ticket[10];
ETicket[] elecarr = new ETicket[10];
```

`arraycopy(elecarr,ticketarr);` $\surd$

`arraycopy(ticketarr,elecarr);` $\times$

Polymorphic data structures

Polymorphic lists

```
public class Node {
  public Object data;
  public Node next;
  ...
}
```

Polymorphic data structures

Polymorphic lists

```
public class Node {
  public Object data;
  public Node next;
  ...
}
```

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return returnval;
  }
}

 public void insert(Object newdata){
   ...
 }
```

Two problems

- Type information is lost, need casts

```
LinkedList list = new LinkedList();
Ticket t1,t2;

t1 = new Ticket();
list.insert(t1);
t2 = (Ticket)(list.head());  // head() returns an Object
```

Two problems

- Type information is lost, need casts

```
LinkedList list = new LinkedList();
Ticket t1,t2;

t1 = new Ticket();
list.insert(t1);
t2 = (Ticket)(list.head());  // head() returns an Object
```

- List need not be homogenous!

```
LinkedList list = new LinkedList();
Ticket t = new Ticket();
Date d = new Date();
list.insert(t);
list.insert(d);
...
```

Java "generics"

Use type variables

```
public class Node<T> {
  public T data;
  public Node next;
  ...
}
```

Java "generics"

Use type variables

```
public class Node<T> {
  public T data;
  public Node next;
  ...
}
```

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return returnval;
  }
}

 public void insert(T newdata){
   ...
 }
```

Java "generics"

Not quite!

```
public class Node<T> {
  public T data;
  public Node next;
  ...
}
```

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return (T) returnval; // Cast!!
  }
}

 public void insert(T newdata){
   ...
 }
```

Java "generics"

- Instantiate generic classes using concrete type

```
LinkedList<Ticket> ticketlist = new LinkedList<Ticket>();
LinkedList<Date> datelist = new LinkedList<Date>();
Ticket t = new Ticket();
Date d = new Date();
ticketlist.insert(t);
datelist.insert(d);
...
```

Polymorphic functions

- A better `arraycopy`

```
public <T> void arraycopy (T[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Polymorphic functions

- A better `arraycopy`

```
public <T> void arraycopy (T[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

- Beware — a type variable may get hidden

```
public <T> T head(){
  T returnval;
  ...
  return returnval;
}
```

Dependent type variables

- Can we copy arrays of one type to another?

```
public <S,T> void arraycopy (S[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Dependent type variables

- Can we copy arrays of one type to another?

```
public <S,T> void arraycopy (S[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

- Instead

```
public <S extends T,T> void arraycopy (S[] src, T[] tgt){
  ...
}
```

Dependent type variables

- Can we copy arrays of one type to another?

```
public <S,T> void arraycopy (S[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

- Instead

```
public <S extends T,T> void arraycopy (S[] src, T[] tgt){
 ...
}
```

- A more generous polymorphic list

```
public <S extends T> void insert(S newdata){...}
```

The covariance problem

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

The covariance problem

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
  // Not OK.  ticketarr[5] refers to an ETicket!
```

The covariance problem

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
  // Not OK.  ticketarr[5] refers to an ETicket!
```

- Again a type error at run time!

The covariance problem

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But . . .

```
...
ticketarr[5] = new Ticket();
  // Not OK.  ticketarr[5] refers to an ETicket!
```

- Again a type error at run time!

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`

Problems with generics

- The following does not work

```
if (s instanceof T){ ... } // T a type variable
```

  Cannot use a type variable wherever a type is expected!

Problems with generics

- The following does not work

  ```
  if (s instanceof T){ ... } // T a type variable
  ```

  Cannot use a type variable wherever a type is expected!

- Cannot define generic arrays

  ```
  T[] newarray;    // Not allowed!
  ```

Problems with generics

- The following does not work

  ```
  if (s instanceof T){ ... } // T a type variable
  ```

  Cannot use a type variable wherever a type is expected!

- Cannot define generic arrays

  ```
  T[] newarray;   // Not allowed!
  ```

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

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```