Polymorphism revisited

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - S is a subclass of T
  - S overrides a method f() defined in T
  - Variable v of type T is assigned to an object of type S
  - v.f() uses the definition of f() from S rather than T
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list (need to compare values)

Structural polymorphism

- Use the Java class hierarchy to simulate this

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`

```
public void reverse (Object[] objarr){
  Object tempobj;
  int len = objarr.length;
  for (i = 0; i < n/2; i++){
    tempobj = objarr[i];
    objarr[i] = objarr[(n-1)-i];
    objarr[(n-1)-i] = tempobj;
  }
}
```

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
  - `==` translates to `Object.equals()`

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

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
  - `==` translates to `Object.equals()`
- Polymorphic `sort`
  - Use interfaces to capture capabilities

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

public 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])
  }
}
```

Type consistency

- Polymorphic function to copy an array

```
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];
  }
}
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time

```
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];
  }
}


Date[] datearr = new Date[10];
Employee[] emparr = new Employee[10];

arraycopy(datearr,emparr); // Run-time error
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
    - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array

```
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];
  }
}


public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(etktarr,tktarr); // Allowed
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array
- But the converse is illegal

```
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];
  }
}


public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(tktarr,etktarr); // Illegal
```

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`

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

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

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

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

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems

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

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

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

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

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts

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

  public Object head(){ ... }

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

  private class Node {...}
}


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

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

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts
  - List need not be homogenous!

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

  public Object head(){ ... }

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

  private class Node {...}
}
```

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

Generic programming in Java

- Java added generic programming to address these issues
- Classes and functions can have type parameters
  - `class LinearList<T>` holds values of type `T`
  - `public T head(){...}` returns a value of same type `T` as enclosing class
- Can describe subclass relationships between type variables
  - `public static <S extends T,T> void arraycopy (S[] src, T[] tgt){...}`