Generic programming in Java

Madhavan Mukund

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

Programming Concepts using Java

Week 5

Structural polymorphism

- Functions that depends only a specific capabilities
  - 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
- May need to impose constraints on types of arguments
  - Copying an array needs source type to extend target type
- Polymorphic data structures
  - Hold values of an arbitrary type
  - Homogenous
  - Should not have to cast return values

Java Generics

- Use type variables

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."

```
public <T> void reverse (T[] objarr){
  T 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;
  }
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` . . . "
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error

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

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical

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

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical
- A more generous `arraycopy`
  - Source and target types may be different
  - Source type must extend target type

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

Polymorphic data structures

- A polymorphic list

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

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

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

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

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole

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

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

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

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

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`

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

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

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

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

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`

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

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

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

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

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`
- Instantiate generic classes using concrete type

```
public class LinkedList<T>{
    ...
}

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 data structures

- Be careful not to accidentally hide a type variable

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

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

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

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

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

Polymorphic data structures

- Be careful not to accidentally hide a type variable

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

- `T` in the argument of `insert()` is a new `T`

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

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

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

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

Polymorphic data structures

- Be careful not to accidentally hide a type variable

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

- `T` in the argument of `insert()` is a new `T`
- Quantifier `<T>` masks the type parameter `T` of `LinkedList`

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

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

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

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

Polymorphic data structures

- Be careful not to accidentally hide a type variable

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

- `T` in the argument of `insert()` is a new `T`
- Quantifier `<T>` masks the type parameter `T` of `LinkedList`
- Contrast with

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

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

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

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

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

Summary

- Generics introduce structural polymorphism into Java through type variables
- Classes and functions can have type parameters
  - `class LinearList<T>` holds values of an arbitrary type `T`
  - `public T head(){...}` returns a value of same type `T` used when creating the list
- Can describe subclass relationships between type variables
  - `public static <S extends T,T> void arraycopy (S[] src, T[] tgt){...}`
- Be careful not to accidentally hide type variables

  `public <T> void insert(T newdata){...}` inside `class LinearList<T>`

  vs

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