Java generics and subtyping

Madhavan Mukund

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

Programming Concepts using Java

Week 5

Extending subtyping in contexts

- 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[]
```

Extending subtyping in contexts

- 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!
```

Extending subtyping in contexts

- 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!
```

- A type error at run time!

Extending subtyping in contexts

- 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!
```

- A type error at run time!
- Java array typing is covariant
  - If `S extends T` then `S[] extends T[]`

Generics and subtypes

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

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

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

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

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

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- How can we get around this limitation?

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

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

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

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

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- <T> is a type quantifier: *For every type* T, ...

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

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

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- `<T>` is a type quantifier: *For every type* `T`, ...
- Note that `T` is not actually used inside the function
  - We use `Object o` as a generic variable to cycle through the list

Wildcards

- Instead, use ? as a wildcard type variable

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

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

Wildcards

- Instead, use ? as a wildcard type variable

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

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- ? stands for an arbitrary unknown type

Wildcards

- Instead, use ? as a wildcard type variable

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

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- ? stands for an arbitrary unknown type
- Avoids unnecessary type variable quantification when the type variable is not needed elsewhere

Wildcards

- Can define variables of a wildcard type

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

LinkedList<?> l;
```

Wildcards

- Can define variables of a wildcard type

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

LinkedList<?> l;
```

- But need to be careful about assigning values

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

LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

Wildcards

- Can define variables of a wildcard type

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

LinkedList<?> l;
```

- But need to be careful about assigning values

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

LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

- Compiler cannot guarantee the types match

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`
- All subclasses override `draw()`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`
- All subclasses override `draw()`
- Want a function to draw all elements in a list of `Shape` compatible objects

```
public static void drawAll(LinkedList<? extends Shape> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    o.draw();
  }
}
```

Bounded wildcards

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

Bounded wildcards

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

- Can reverse the constraint, using `super`

```
public static <T,? super T>
       void listcopy (LinkedList<T> src,
                      LinkedList<?> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

Summary

- Java generics are not covariant, unlike arrays
- Cannot substitute `Object` for `T` to get most general type
- Instead, use type quantification `<T>` or wild card type variable `?`
- Wild card can be used wherever the type `T` is not required within the function
  - When `T` is not needed for return type, or to declare local variables
- Wild cards can be bounded
  - `LinkedList<? extends T>`
  - `LinkedList<? super T>`