Java generics at run time

Madhavan Mukund

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

Programming Concepts using Java

Week 5

Erasure of generic information

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

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

Erasure of generic information

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

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

- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`

Erasure of generic information

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

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

- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Or, the upper bound, if one is available
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`

Erasure of generic information

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

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```
- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Or, the upper bound, if one is available
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`
- Since no information about `T` is preserved, cannot use `T` in expressions like

  ```
  if (o instanceof T) {...}
  ```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns True

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

- As a consequence the following overloading is illegal

```
public class Example {
    public void printlist(LinkedList<String> strList) { }
    public void printlist(LinkedList<Date> dateList) { }
}
```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

- As a consequence the following overloading is illegal

```
public class Example {
    public void printlist(LinkedList<String> strList) { }
    public void printlist(LinkedList<Date> dateList) { }
}
```

- Both functions have the same signature after type erasure

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`

Arrays and generics

- Recall the covariance problem for arrays
    - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;           // OK
newarray = new T[100]; // Cannot create!
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;           // OK
newarray = new T[100]; // Cannot create!
```

- An ugly workaround . . .

```
T[] newarray;
newarray = (T[]) new Object[100];
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;          // OK
newarray = new T[100]; // Cannot create!
```

- An ugly workaround ... generates a compiler warning but works!

```
T[] newarray;
newarray = (T[]) new Object[100];
```

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
    - `LinkedList<T>` becomes `LinkedList<Object>`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .
- Wrapper class for each basic type:

Basic type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`

Basic type	Wrapper Class
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, ... are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, ...
- Wrapper class for each basic type:

Basic type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`

Basic type	Wrapper Class
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

- All wrapper classes other than `Boolean`, `Character` extend the class `Number`

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, . . .

- Autoboxing — implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...
- Autoboxing — implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

- Use wrapper types in generic data structures

Summary

- Java generics come with some restrictions
- Information about type variables is erased at runtime
  - `LinkedList<T>` becomes `LinkedList<Object>`
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`
- Limits the use reflection on generic types — cannot write
  - `if (o instanceof LinkedList<String>) {...}`
  - `if (o instanceof T) {...}`
- Cannot overload function signatures using instantiation of generic types
- Cannot instantiate arrays of generic type
- Need to box built-in types using wrapper types