CS601: Principles of Software Development

Generics Continued.

Olga A. Karpenko

This presentation is based on the lecture notes of Anupam Chanda.

Quiz 6

Announcements

- Lab 3 Part 2 is out, due on Friday
- Midterm Exam: next week on Friday
  - Please look at Midterm Review

Java Generics

- Allow the programmer to write more "generic" code
- **Provide compiler time type checks**

Parameterized Methods

- Another example: `MapUtil.java`

Java 7 Type Inference

```
ArrayList<Integer> a = new ArrayList<>();
```

no type

- Type inferred from the variable it is assigned to

Raw Type

```
List list = new ArrayList();
```

- Using generics without specifying the type

Raw Type

```
List list = new ArrayList();
```

- Using generics without specifying the type
- Allowed for backward compatibility
- See `Max.java`

Raw Type

```
List list = new ArrayList();
list.add("red");
list.add(34);
list.add(new Student("Jason"));
```

- Compiles and runs fine

Raw Type

```
List list = new ArrayList();
list.add("red");
list.add(34);
list.add(new Student("Jason"));
for (Object obj: list) {
    String s = (String)obj;
    System.out.println(s);
}
```

- Compiles, but crashes at runtime
- Note: compiler was not able to catch it!

Raw Type

- Avoid raw types! You lose type safety
- Generics give compiler time safety

Bounded Parameterized Types

- Sometimes useful to bound (restrict) the parameter of the class
- Example:
  - A “box” that holds only Number objects
    - and subclasses
  - Can’t use MathBox<T> . Why?

Bounded Parameterized Types

- Example: Container for objects of class Number and its subclasses

```
public class MathBox<T extends Number> {
  private T data;
  public MathBox(T data) {
    this.data = data;
  }
  public T getData() {
    return data;
  }
}
  public double sqrt() { // can only work on numbers
    return Math.sqrt(getData().doubleValue());
  }
}
```

Bounded Parameterized Types

```
MathBox<Integer> intBox = new MathBox<Integer>(15);
int i = intBox.getData(); // ok

MathBox<Double> doubleBox = new MathBox<Double>(4.5);
double d = doubleBox.getData(); // ok
```

- Error if we try to create a MaxBox of Strings

```
MathBox<String> strBox = new MathBox<String>("oops");
// Compiler Error
```

Bounded Parameterized Types

- Use the keyword `extends` even when the type *implements* an interface

```
public class Test<T extends Comparable<T>>
{
    // code


}
```

Example

- Earlier we implemented findMax with raw Comparable types
- We can now implement it with Generic types
- `GenericMax.java` – better solution
- Now type errors will be caught at compile time

Type Erasure

- Compiler replaces every type parameter with
  - Object, if type is unbounded
  - Bound
- Compiler adds casting as needed
- Compiler might add some methods (for polymorphism etc.)

https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Type Erasure: Example

```
class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

```
class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node
next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData()
    { return data; }
    // ...
}
```

Inheritance and Generics

- Number is a superclass of Integer
- But MathBox<Number> is **not** a superclass of MathBox<Integer>

```
MathBox<Number> numBox;

numBox = new MathBox<Integer>(31);//Compiler Error!
```

Another example:

```
ArrayList<Object> arr = new ArrayList<Integer>();//Error!
```

Unbounded Wildcards

- MathBox<?> is a superclass of MathBox<Integer>
- Inheritance with parameterized classes is tricky

```
MathBox<?> numBox = new MathBox<Integer>(31); // ok
```

See WildCardsExample.java

Unbounded Wildcards

- When *any* type parameter works
- <?> is used to specify unbounded wildcards

```
Box<?> b1 = new Box<Integer>(31);
Box<?> b2 = new Box<String>( “Hi” );
b1 = b2; // valid
```

Upper Bounded Wildcards in Parameterized Types

- We want “a MyBox of any type which extends Number”

```
MathBox<? extends Number> numBox = new MathBox<Integer>(31);
System.out.println(numBox.getData());
```

Upper Bounded Wild Cards

<? extends T>

- Is called an “upper bounded wildcard”
- Defines a type that *is bounded* by the superclass T

- See `WildCardsExample.java`

Lower Bounded Wild Cards

<? super T>

- Is called a “lower bounded wildcard”
- Defines a type that *is bounded* by the subclass T
- Use when T is a “consumer”

- See `WildCardsExample.java`