

Comp 212: Intermediate
Programming
Lecture 18 – Java Generics

By: Anupam Chanda

Today's Lecture

- Java generics
 - Parameterized classes and methods
 - Compiler provides type safety
- Syntax and semantics
 - Examples
- Generics-based implementation of the list framework – next class

Outline

- Motivation
- Parameterized classes
- Parameterized methods
- Upper bounded wildcards
- Lower bounded wildcards
- Unbounded wildcards

Cast Exceptions at Runtime

```
public class OldBox {
    Object data;
    public OldBox(Object data) {
        this.data = data;
    }
    public Object getData() {
        return data;
    }
}

OldBox intBox = new OldBox(42);
int x = (Integer) intBox.getData();

OldBox strBox = new OldBox("Hi");
String s = (String) strBox.getData();

int y = (Integer) strBox.getData();
intBox = strBox;
```

ClassCastException!
Compiles but fails at runtime

Naïve Solution

```
public class IntBox {
    Integer data;
    public IntBox(Integer data) {
        this.data = data;
    }
    public Integer getData() {
        return data;
    }
}
```

```
public class StrBox {
    String data;
    public StrBox(String data) {
        this.data = data;
    }
    public String getData() {
        return data;
    }
}
```

```
IntBox intBox = new IntBox(42);
int x = intBox.getData();

StrBox strBox = new StrBox("Hi");
String s = strBox.getData();

int y = (Integer) strBox.getData();
intBox = strBox;
```

```
public class FooBox {
    Foo data;
    public FooBox(Foo data) {
        this.data = data;
    }
    public Foo getData() {
        return data;
    }
}
```

Errors caught by compiler

Infinite many classes possible

Passing Parameters to Methods: An Analogy

```
public abstract class Sum {  
    public static int sum_0_1() {  
        return (0+1);  
    }  
    ...  
    public static int sum_15_22() {  
        return (15+22);  
    }  
    ...  
}
```

```
public abstract class NewSum {  
    public static int  
        sum(int m, int n) {  
        return (m+n);  
    }  
}
```

Methods accept parameters

```
public class Main {  
    public static void  
    main(String[] nu) {  
        int j = Sum.sum_0_1();  
        ...  
        int k = Sum.sum_15_22();  
    }  
}
```

Bad – infinite many methods

```
public class NewMain {  
    public static void  
    main(String[] nu) {  
        int j = NewSum.sum(0,1);  
        ...  
        int k = NewSum.sum(15,22);  
    }  
}
```

Pass parameters to methods

Java Generics: Key Idea

- Parameterize type definitions
 - Parameterized classes and methods
- Provide type safety
 - Compiler performs type checking
 - Prevent runtime cast errors

Parameterized Classes

```
public class OldBox {  
    Object data;  
    public OldBox(Object data) {  
        this.data = data;  
    }  
    public Object getData() {  
        return data;  
    }  
}
```

- We want the box to hold a “specific” class – abstractly represented
- `Object` does not work as we have seen earlier
- Solution – parameterize the class definition

```
public class Box<E> {  
    E data;  
    public Box(E data) {  
        this.data = data;  
    }  
    public E getData() {  
        return data;  
    }  
}
```

- `E` refers to a particular type
- The constructor takes an object of type `E`, not any object
- To use this class, `E` must be replaced with a specific class

How to Use Parameterized Classes

```
public class Box<E> {
    E data;
    public Box(E data) {
        this.data = data;
    }
    public E getData() {
        return data;
    }
}
```

```
Box<Integer> intBox =
    new Box<Integer>(42);
int x = intBox.getData(); //no cast needed

Box<String> strBox =
    new Box<String>("Hi");
String s = strBox.getData(); //no cast needed
```

Following lines will not compile anymore:

```
String s = (String) intBox.getData();
int y = (Integer) strBox.getData();
intBox = strBox;
```

**Runtime errors now converted to
compile time errors**

When to Use Parameterized Classes

- Particularly useful for “container” classes
 - Containers hold but do not process data
- All collections framework classes in Java 5.0 defined using generics
 - See the Java 5.0 API documentation

Parameterized Classes: Syntax Note

A class can have multiple parameters, e.g:

```
public class Stuff<A,B,C> { ... }
```

Subclassing parameterized classes allowed, e.g:

```
/* Extending a particular type */  
class IntBox extends Box<Integer> { ... }
```

Or

```
/* Extending a parameterized type */  
class SpecialBox<E> extends Box<E> { ... }
```

SpecialBox<String> is a subclass of Box<String>.

```
/* Following assignment is legal */  
Box<String> sb = new SpecialBox<String>("Hi");
```

Parameterized Classes in Methods

A parameterized class is a type just like any other class.

It can be used in method input types and return types, e.g:

```
Box<String> aMethod(int i, Box<Integer> b) { ... }
```

If a class is parameterized, that type parameter can be used for any type declaration in that class, e.g:

```
public class Box<E> {  
    E data;  
    public Box(E data) {  
        this.data = data;  
    }  
    public E getData() {  
        return data;  
    }  
    public void copyFrom(Box<E> b) {  
        this.data = b.getData();  
    }  
} //We have added an infinite number of types of Boxes  
//by writing a single class definition
```

So Far...

- Type safety violations
 - Using casts
- Parameterized classes solve this problem
- Provide type safety
 - Enforced by the compiler
- Particularly useful for container classes
- A parameterized class is another *type*
- Next – bounded parameterized classes

Bounded Parameterized Types

Sometimes we want restricted parameterization of classes.
We want a box, called `MathBox` that holds only `Number` objects.
We can't use `Box<E>` because `E` could be anything.
We want `E` to be a subclass of `Number`.

```
public class MathBox<E extends Number> extends Box<Number> {  
    public MathBox(E data) {  
        super(data);  
    }  
    public double sqrt() {  
        return Math.sqrt(getData().doubleValue());  
    }  
}
```

Bounded Parameterized Types (Contd.)

```
public class MathBox<E extends Number> extends Box<Number> {  
    public MathBox(E data) {  
        super(data);  
    }  
    public double sqrt() {  
        return Math.sqrt(getData().doubleValue());  
    }  
}
```

The `<E extends Number>` syntax means that the type parameter of `MathBox` must be a subclass of the `Number` class.

We say that the type parameter is **bounded**.

```
new MathBox<Integer>(5); //Legal  
new MathBox<Double>(32.1); //Legal  
new MathBox<String>("No good!"); //Illegal
```

Bounded Parameterized Types (Contd.)

Inside a parameterized class, the type parameter serves as a valid type. So the following is valid.

```
public class OuterClass<T> {  
    private class InnerClass<E extends T> {  
        ...  
    }  
    ...  
}
```

Syntax note: The `<A extends B>` syntax is valid even if `B` is an interface.

Bounded Parameterized Types (Contd.)

Java allows multiple inheritance in the form of implementing multiple interfaces. So multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

```
<T extends A & B & C & ...>
```

For instance:

```
interface A {  
    ...  
}  
interface B {  
    ...  
}  
class MultiBounds<T extends A & B> {  
    ...  
}
```

So Far...

- Parameterized classes
- Bounded parameterized types
 - To restrict parameter types
- Next – parameterized methods

Parameterized Methods

Consider the following class:

```
public class Foo {  
    //Foo is not parameterized  
    public <T> T aMethod(T x) {  
        //will not compile without <T>  
        //to indicate that this is a  
        //parameterized method.  
        return x;  
    }  
    public static void  
    main(String[] args) {  
        Foo foo = new Foo();  
        int k = foo.aMethod(5);  
        String s = foo.aMethod("abc");  
    }  
}
```

Fix `foo` and vary parameter to `aMethod()`

```
public class Bar<T> {  
    //Bar is parameterized  
    public T aMethod(T x) {  
        return x;  
    }  
    public static void  
    main(String[] args) {  
        Bar<Integer> bar =  
            new Bar<Integer>();  
        int k = bar.aMethod(5);  
        String s = bar.aMethod("abc");  
        //Compilation error here  
    }  
}
```

Once `Bar<T>` object is fixed, we are locked to a specific `T`.

Use of Parameterized Methods

- Adding type safety to methods that operate on different types
 - Return type dependent on input type

So Far...

- Parameterized classes
- Bounded parameterized types
- Parameterized methods
- Next – wildcards
 - Bounded
 - Upper
 - Lower
 - Unbounded

Upper Bounded Wildcards in Parameterized Types

We start to run into some new issues when we do some things that seem "normal". For instance, the following seems reasonable:

```
Box<Number> numBox = new Box<Integer>(31);
```

Compiler comes back with an **"Incompatible Type"** error message.

This is because `numBox` can hold only a `Number` object and nothing else, not even an object of type `Integer` which is a subclass of `Number`.

The type of `numBox` we desire is "a `Box` of any type which extends `Number`".

```
Box<? extends Number> numBox = new Box<Integer>(31);
```

Upper Bounded Wildcards in Parameterized Types (Contd.)

```
public class Box<E> {  
    public void copyFrom(Box<E> b) {  
        this.data = b.getData();  
    }  
}
```

//We have seen this earlier

//We can rewrite copyFrom() so that it can take a box
//that contains data that is a subclass of E and
//store it to a Box<E> object

```
public class Box<E> {  
    public void copyFrom(Box<? extends E> b) {  
        this.data = b.getData(); //b.getData() is a  
                                //subclass of this.data  
    }  
}
```

<? extends E> is called "*upper bounded wildcard*" because it defines a type that is bounded by the superclass E.

Lower Bounded Wildcards in Parameterized Types

Suppose we want to write `copyTo()` that copies data in the opposite direction of `copyFrom()`.

`copyTo()` copies data from the host object to the given object.

This can be done as:

```
public void copyTo(Box<E> b) {  
    b.data = this.getData();  
}
```

Above code is fine as long as `b` and the host are boxes of exactly same type. But `b` could be a box of an object that is a superclass of `E`.

This can be expressed as:

```
public void copyTo(Box<? super E> b) {  
    b.data = this.getData();  
    //b.data() is a superclass of this.data()  
}
```

`<? super E>` is called a "*lower bounded wildcard*" because it defines a type that is bounded by the subclass `E`.

Unbounded Wildcards

Use unbounded wildcards when *any* type parameter works.
<?> is used to specify unbounded wildcards.

The following are legal statements.

```
Box<?> b1 = new Box<Integer>(31);  
Box<?> b2 = new Box<String>("Hi");  
b1 = b2;
```

Wildcard capture:

The compiler can figure out exactly what type `b1` is above from the right hand side of the assignments.

This “capturing” of type information means:

1. The type on the left hand doesn't need to be specified.
2. The compiler can do additional type checks because it knows the type of `b1`.

Conclusions

- Java generics
 - Parameterized classes and methods
 - Type safety
 - Syntax and semantics through examples
- Links to tutorials on the lecture page
- Generics-based implementation of the list framework – next class