Parametric Polymorphism in Java

Java Generics

KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk

Overview

Java Generics

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

Cast Exceptions at Runtime I

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

Cast Exceptions at Runtime II

```
public class OldBoxDriver {
    public static void main(String[] args) {
        OldBox intBox = new OldBox(42);
        int x = (Integer) intBox.getData();

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

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

`ClassCastException!`

Compiles but fails at runtime

Naive Solution I

Different types of boxes

```
public class IntBox {
    Integer data;

    public IntBox(Integer data) {
        this.data = data;
    }

    public Integer getData() {
        return data;
    }
}
```

Naive Solution II

```
public class StrBox {
    String data;

    public StrBox(String data) {
        this.data = data;
    }

    public String getData() {
        return data;
    }
}
```

Naive Solution III

```
public class FooBox {
    Foo data;

    public FooBox(Foo data) {
        this.data = data;
    }

    public Foo getData() {
        return data;
    }
}
```

Naive Solution IV

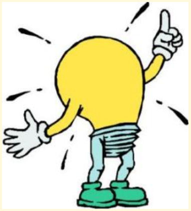
```
public class BoxDriver {
    public static void main(String[] args) {
        IntBox intBox = new IntBox(42);
        int x = intBox.getData();

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

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

Infinite many classes possible.
The errors caught by compiler.

Passing Parameters to Methods I



If we consider the way in which we deal with differing parameters being passed to methods perhaps we can get a clue as to how to deal with this problem?

Passing Parameters to Methods II

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

    public static int sum_15_22() {
        return (15 + 22);
    }
}
```

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

Bad — infinite number of methods

Passing Parameters to Methods III

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

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

Pass parameters to methods

Java Generics — key ideas

Parameterise type definitions

- classes and methods

Provide type safety

- compiler performs type checking
- prevent runtime cast errors

Java Generics provoke(d) controversy

“. . . Yet, the Generics syntax is invasive, and the implementation is worse. In an age when more and more experts assert that dynamic typing leads to simpler applications and productive programmers, Java developers are learning how to build stronger enforcement for static types.”

from *Beyond Java* by Bruce Tate

Parameterised Classes I

```
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
- Possible solution — parameterise the class definition

Parameterised Classes II

```
public class Box<T> {
    T data;

    public Box(T data) {
        this.data = data;
    }

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

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

Parameterised Classes III

Usage:

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

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

which also results in the following lines not compiling anymore:

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

So now the runtime errors have been “moved” to compile time errors

Parameterised Classes IV

Parameterised classes are used for:

- *container* classes (which hold, but do not process data)
- all the collections framework classes in Java (since Java 5.0)

Parameterised Classes V

A class can have multiple parameters, e.g.:

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

Sub-classing of parameterised classes is available:

- extending a particular type

  ```
  class IntBox extends Box<Integer> { ... }
  ```

- extending a parameterised type

  ```
  class SpecialBox<T> extends Box<T> { ... }
  ```

Parameterised Classes VI

The following assignment is legal:

```
Box<String> sb = new SpecialBox<String>("Hello");
```

as `SpecialBox<String>` is a subclass of `Box<String>`

Parameterised classes in methods I

- A parameterised 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) { ... }
```

Parameterised classes in methods II

- If a class is parameterised, 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();
  }
}
```

which results in the availability of an infinite number of types of `Box`es just by writing a single class definition

Summary. . .

- Type safety violations (using casts)
- Parameterised classes solve this problem
- Provide type safety which is enforced by the compiler
- Particularly useful for container classes
- A parameterised class is just another *type*

and now onto *bounded* parameterised classes and methods

Bounded parameterised types I

- Sometimes we want restricted parameterisation of classes
- For example, we want a box, called `MathBox` that holds only `Number` objects
- We cannot use `Box<E>` because `E` could be anything
- We want to *restrict* `E` to be a subclass of `Number`

Bounded parameterised types II

```
public class MathBox<E extends Number> extends Box<Number> {

    public MathBox(E data) {
        super(data);
    }

    public double sqrt() {
        return Math.sqrt(getData().doubleValue());
    }
}
```

Bounded parameterised types III

- 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 parameterised types IV

- Inside a parameterised class, the type parameter serves as a valid type, e.g.:

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

Note: The `<A extends B>` syntax is valid even if `B` is an interface

Bounded parameterised types V

- As Java allows multiple inheritance of interfaces we can use this in our bounded parameterised type, e.g.:

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

For instance:

```
interface A { ... }

interface B { ... }

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

Summary...

- Parameterised classes
- Bounded parameterised types (to restrict parameter types)

and now onto parameterised methods

Parameterised Methods I

Adding type safety to methods that operate on different types
Consider the following class:

```
public class Foo {
    //Foo is not parameterised

    public <T> T aMethod(T x) {
        // will not compile without <T>
        // to indicate that this is a
        // parameterised method.
        return x;
    }

    public static void main(String[] args) {
        Foo foo = new Foo();
        int k = foo.aMethod(5);
        String s = foo.aMethod("abc");
    }
}
```

How do we fix `foo` and vary the parameter to `aMethod()`?

Parameterised Methods II

```
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 a `Bar<T>` is fixed we are locked to a specific `T`

Summary. . .

- Parameterised classes
- Bounded parameterised types
- Parameterised methods

and now onto *wildcards*

Wildcards I

Come in different varieties. . .

- Bounded
  - Upper
  - Lower
- Unbounded

Wildcards II

We start to run into some new issues when we do some things that seem *normal*

- Consider the following case:

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

  The compiler issues the error message `Incompatible Type`

- This is because `numBox` can hold only a `Number` object and nothing else (which includes `Integer` which is a subclass of `Number`)

Wildcards III

So what is the solution?

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

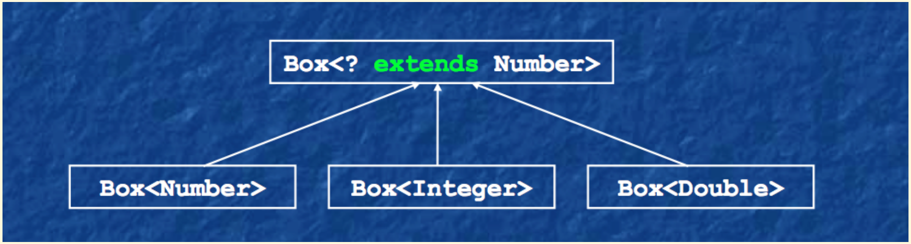
- This formalism is known as an upper bounded wildcard because it defines a type that is bounded by the specified superclass

```
//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 BoxUpper<E> {
    E data;

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

Wildcards IV



Upper bounded wildcard example

Wildcards V

The next formalism we consider is the lower bounded wildcard

- Suppose we want to write a `copyTo()` that copies data in the opposite direction to `copyFrom`
- `copyTo()` copies information from the host object to the given object, i.e.,

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

- This code fragment is fine as long as `b` and the target are boxes of exactly the same type.

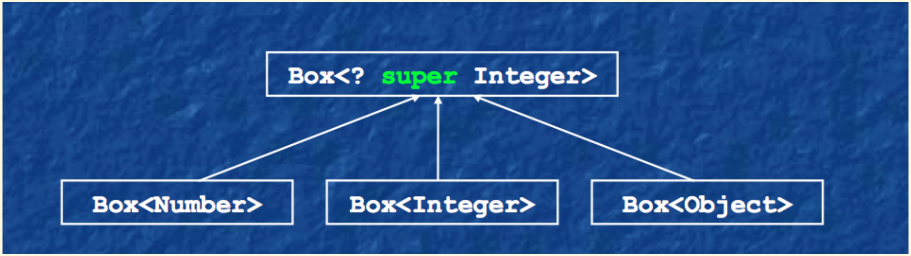
Wildcards VI

If we consider that `b` could be a box of an object that is a superclass of `E` then we need something like...

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

where `b.data()` is a superclass of `this.data()`

Wildcards VII



Lower bounded wildcard example

Wildcards VIII

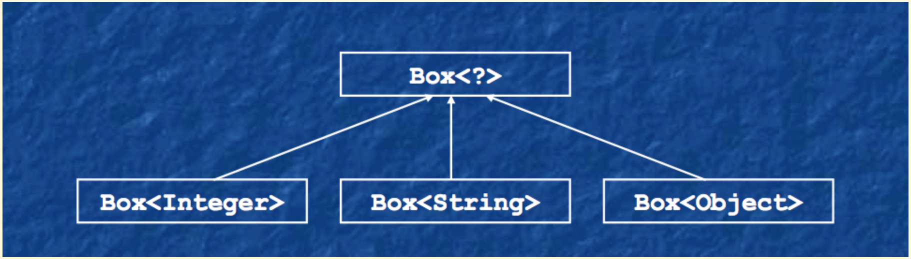
and finally... Unbounded Wildcards

- We use these when *any* type parameter will work
- This is represented as `<?>`

  Examples:

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

Wildcards IX



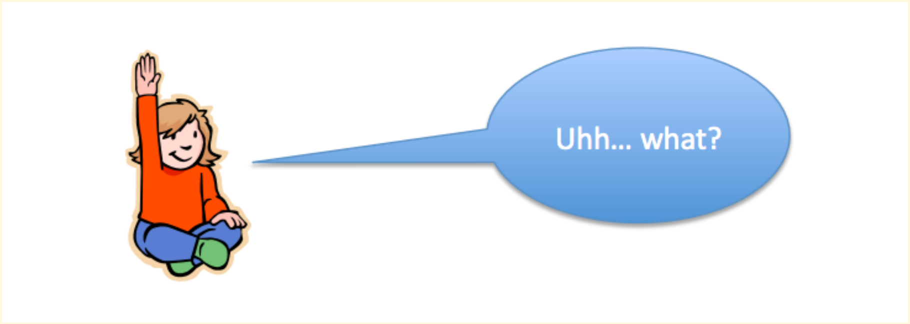
Unbounded wildcard example

Wildcards X

Note: *Wildcard capture*

- In the above example the compiler can figure out exactly what type `b1` is by considering the right hand side of the assignment.
- This *capturing* of type information means
  1. The type on the left hand side of the assignment does not need to be specified
  2. The compiler can do additional type checks because it knows the type of `b1` (in this example)

Wildcards XI



Wildcards XII

Josh Bloch's Bounded Wildcards Rules

- Use `<? extends T>` when parameterised instance is a producer of `T` (for reading/input)
- Use `<? super T>` when parameterised instance is a consumer of `T` (for writing output)

How it all works — *Type Erasure...* I

Reminder:

- Java Generics implement *parametric polymorphism*
  - Parametric: the type parameter (e.g., `<T>`)
  - Polymorphism: can take many forms
- However if we are going to program with these types we need to understand how the language rules apply to them otherwise we might get a little **shock**!
- Java Generics are implemented using *type erasure*, which leads to some *interesting* issues...

How it all works — *Type Erasure. . .* II

- Rather than change every JVM between Java 1.0 and the Java 1.5 (when generics were introduced) they chose to use *erasure*
- After the compiler does its type checking, it *discards* the generics; the JVM never sees this information
- Which works something like this:
  - Type information between angle brackets is thrown out, e.g., `List<String>` $\rightarrow$ `List`
  - Uses of type variables are replaced by their upper bound (usually `Object`)
  - *Casts* are inserted to preserve type correctness

How it all works — *Type Erasure...* III

The Pros and Cons of type erasure

Good Backward compatibility is maintained, so you can still use legacy (non-generic) code (e.g., libraries)

Bad You cannot find out what type a generic class is using at runtime

```
public class Example<T> {
    void method(Object item) {
        if (item instanceof T) { ... } // Compiler error!
        T anotherItem = new T(); // Compiler error!
        T[] itemArray = new T[10]; // Compiler error!
    }
}
```

Using Legacy Code in Generic Code

- What if we have some generic code dealing with *Fruit* but I want to call the following legacy library method:

```
Smoothie makeSmoothie(String name, List fruits);
```

- We can pass in a generic `List<Fruit>` for the `fruits` parameter, which has the raw type `List`
- But why?
- That seems unsafe…
- `makeSmoothie()` could put a `Vegetable` in the list, and that might not taste too good!

Raw Types and Generic Types

- `List` does not mean `List<Object>` because then we could not pass in a `List<Fruit>`
- `List` does not mean `List<?>` either, because then we could not assign a `List` to a `List<Fruit>` (which is legal)
- We need both of these to work for generic code so that we can interoperate with legacy code
- *Raw types* basically work like wildcard types, just that they are not checked as stringently (they will probably generate an *unchecked* warning)

The problem with legacy code

As the lead developer for Java Generics put it…

> *"Calling legacy code from generic code is inherently dangerous; once you mix generic code with nongeneric legacy code, all the safety guarantees that the generic type system usually provides are void. However, you are still better o than you were without using generics at all. At least you know the code on your end is consistent."*
>
> Gilad Bracha

Summary

- Parameterised classes and methods
- Type safety
- Horrendous syntax and (perhaps) dodgy semantics

That's all Folks!