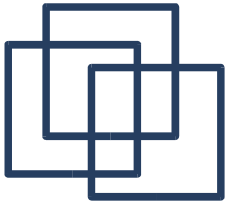




Wildcards in Java

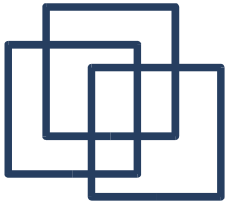
Wildcards in Java

Christian Plesner Hansen



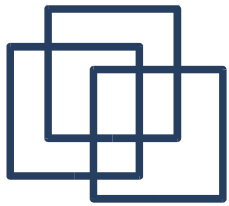
What are wildcards?

- An extension of the genericity mechanism in Java.
- A part of the forthcoming release of the Java platform, 1.5. Forthcoming means sometime this decade.
- Developed by people at DAIMI and Sun Microsystems.



Overview

- Introduction
- **Why wildcards?**
- Basic wildcards
- Advanced wildcards
- Summary



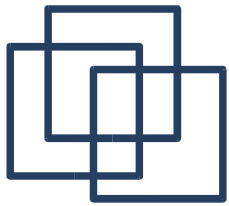
Reminder: Why generics?

- Before generics:

```
/** All elements of the list must be mutually
 * comparable ... */
Object max(List list) {
    ...
}
```

- After:

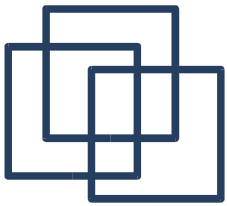
```
<T extends Comparable<T>> T max(List<T> list) {
    ...
}
```



Reminder: Why generics?

- Without generics, we cannot express “extra” type information, such as the element type of a list.
- With generics, we have to give the extra information:

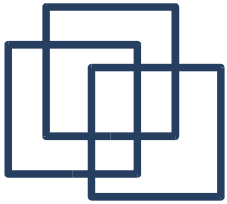
```
List<Object> objs = ...;
```



Why wildcards?

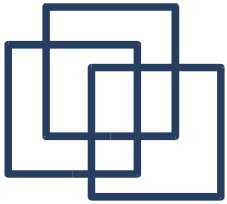
- Unfortunately, no killer one-slide example.
- Instead, a killer ten-slide example: Batch Printing.





Batch Printer

```
class BatchPrinter {  
  
    /* ... thread and printing stuff ... */  
    public void start() { ... }  
  
    private final List jobs;  
  
    /** @param jobs The jobs to print. All  
     * elements must implement Printable */  
    public BatchPrinter(List jobs) {  
        this.jobs = jobs;  
    }  
  
}
```

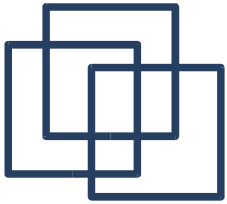


Batch Printer

- The BatchPrinter is a utility class:

```
public class GhostView {  
    private List psDocs = ...;  
    ...  
    new BatchPrinter(psDocs).start();  
}
```

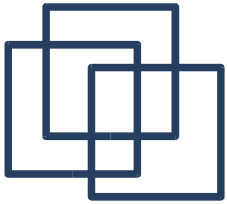
```
public class OpenOffice {  
    private List writerDocs = ...;  
    ...  
    new BatchPrinter(writerDocs).start();  
}
```



Batch Printer

- Along comes java generics, and we can now generify the BatchPrinter:

```
class BatchPrinter {  
    /* ... thread and printing stuff ... */  
    private final List<Printable> jobs;  
    public BatchPrinter(List<Printable> jobs) {  
        this.jobs = jobs;  
    }  
}
```

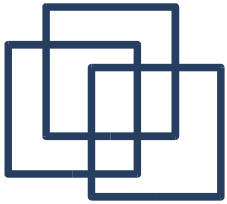


Batch Printer

- ... meanwhile, Aladdin and Sun are also generifying their code ...

```
public class GhostView {  
    private List<PSDoc> psDocs = ...;  
    ...  
    new BatchPrinter(psDocs).start();  
}
```

```
public class OpenOffice {  
    private List<WriterDoc> writerDocs = ...;  
    ...  
    new BatchPrinter(writerDocs).start();  
}
```



Batch Printer

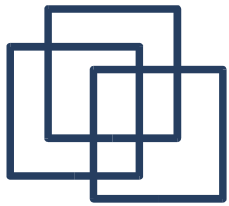
- ... and suddenly getting errors ...

```
GhostView.java:1335: cannot find symbol  
  symbol   : constructor BatchPrinter(List<PSDoc>)  
  location: class Test.BatchPrinter  
    new BatchPrinter(psDocs);  
      ^
```

- What is wrong here?

```
public BatchPrinter(List<Printable> jobs) {  
    ...  
}
```

```
private List<PSDoc> psDocs = ...;  
new BatchPrinter(psDocs).start();
```

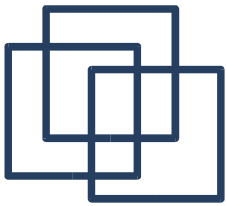


Invariant Subtyping

- The type `List<PSDoc>` is not a subtype of `List<Printable>`:

```
List<PSDoc> psDocs = new ArrayList<PSDoc>();  
List<Printable> prts = psDocs;  
prts.set(0, new WriterDoc(...));  
PSDoc doc = psDocs.get(0); // Error!
```

- The types `C<X>` and `C<Y>` are considered unrelated unless $X = Y$.
- We have too much type information!



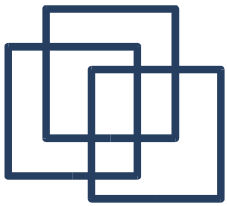
Invariant Subtyping

- Usually, in an OO language, you are free to discard type information:

```
Object obj = "Hola Mundo";
```

- With generics, you are forced to always keep around type argument information.

```
ArrayList<String> strs = nCopies("Hola Mundo", 10);  
Collection<String> cstrs = strs; // Ok  
Collection<Object> objs = cstrs; // Illegal!
```



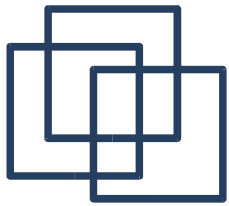
Batch Printing

- How do we avoid breaking GhostView and OpenOffice?
- An idea: use polymorphic methods.

```
<T extends Printable> BatchPrinter(List<T> jobs) {  
    this.jobs = jobs;  
}
```

```
new BatchPrinter(psDocs).start();  
new BatchPrinter(writerDocs).start();
```

- Problem solved!



Polymorphic Methods

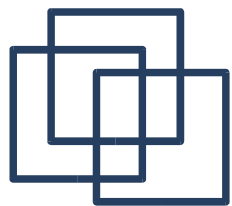
- No, problem not solved:

```
BatchPrinter.java:1356: incompatible types  
found   : List<T>  
required: List<Printable>  
        this.jobs = jobs;  
                ^
```

- Why? Because we store the jobs in a field:

```
private final List<Printable> jobs;  
<T extends Printable> BatchPrinter(List<T> jobs) {  
    this.jobs = jobs;  
}
```

- Still too much type information.



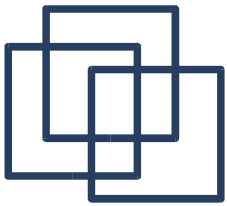
Working around the problem

- A possible workaround:

```
private final List<Printable> jobs;
<T extends Printable> BatchPrinter(List<T> jobs) {
    this.jobs = new ArrayList<Printable>();
    for (Printable job : jobs)
        this.jobs.add(job);
}
```

- Another workaround:

```
class BatchPrinter<T extends Printable> {
    private final List<T> jobs;
    public BatchPrinter(List<T> jobs) {
        this.jobs = jobs;
    }
}
```

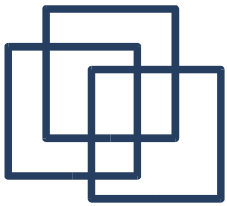


Batch Printing

- What we want is a common supertype of `List<PSDoc>` and a `List<WriterDoc>`.

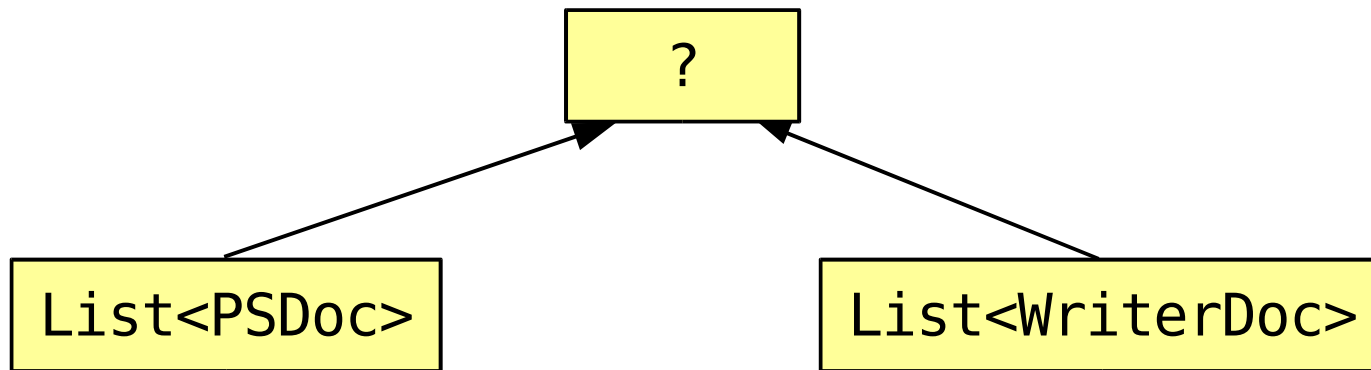
```
private final ??? jobs;  
public BatchPrinter(??? jobs) {  
    this.jobs = jobs;  
}
```

```
new BatchPrinter(psDocs);  
new BatchPrinter(writerDocs);
```

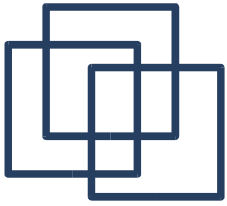


The problem

- We need a type that abstracts over different parameterizations of the same class.
- In other words: discard type argument information.

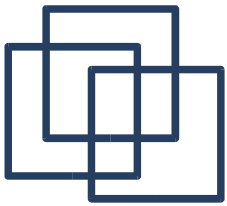


- Enter wildcards.



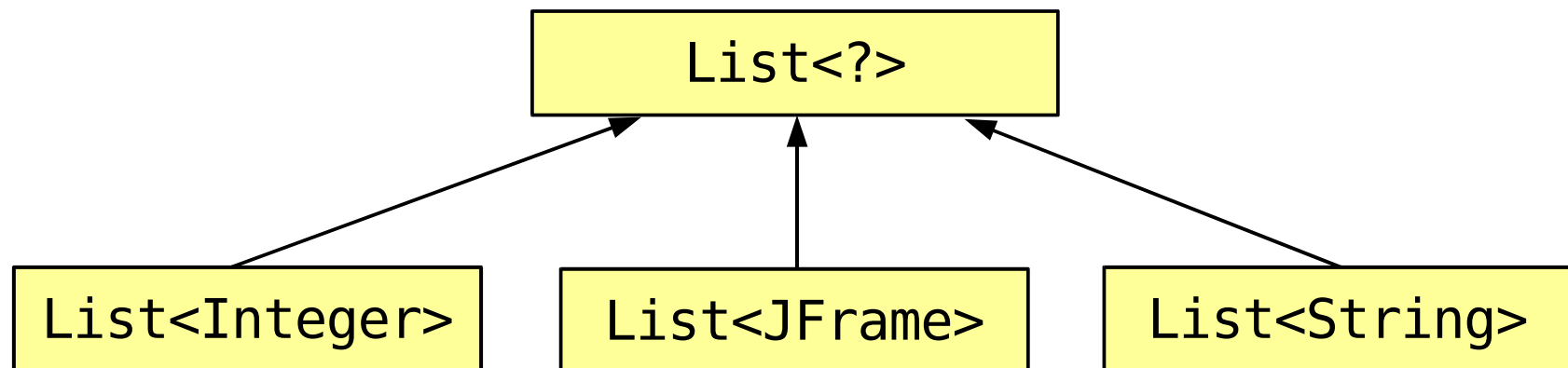
Overview

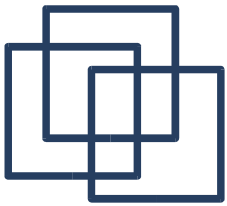
- Introduction
- Why wildcards?
- **Basic wildcards**
 - Unbounded
 - Extends-bounded
 - Super-bounded
- Advanced wildcards
- Summary



Unbounded wildcards

- A `List<?>` represents any kind of list.
- A supertype of `List<T>` for any `T`.





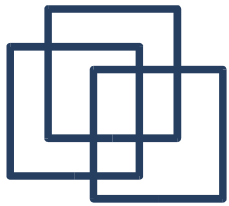
Unbounded wildcards

- You do not know the element type of a `List<?>`, so writing is illegal (except `null`):

```
List<String> strs = new ArrayList<String>();  
List<?> objs = strs;  
objs.put(0, new Integer(0)); // Error!  
String str = strs.get(0);
```

- You can read Objects:

```
List<?> objs = ...;  
Object obj = objs.get(); // Ok
```



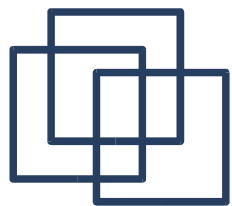
Unbounded wildcards

- Example: `printAll` prints all the element in a list:

```
void printAll(List<?> list) {  
    for (Object elm : list) {  
        System.out.println(elm);  
    }  
}
```

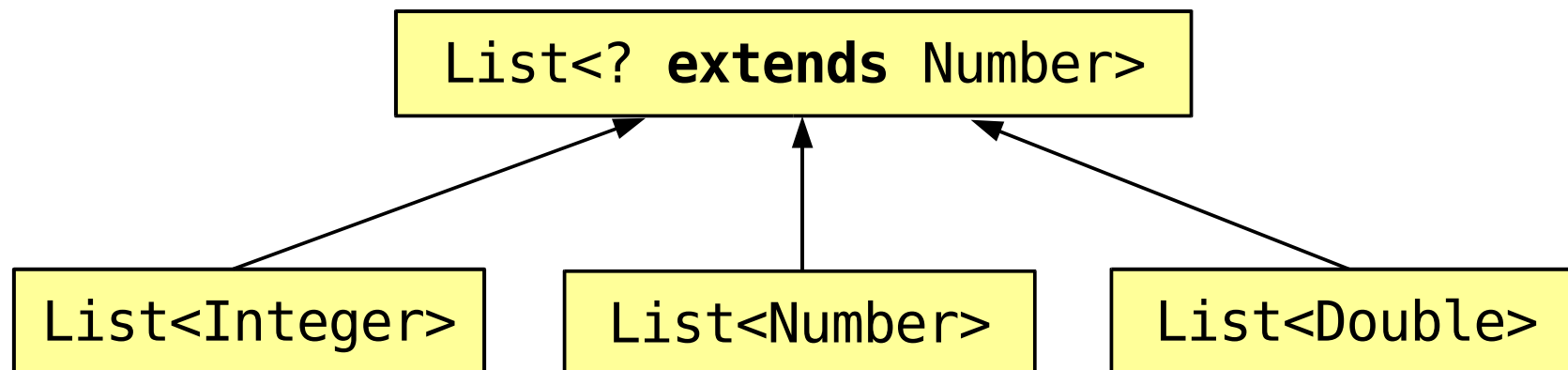
```
List<String> strs = ...;  
printAll(strs);
```

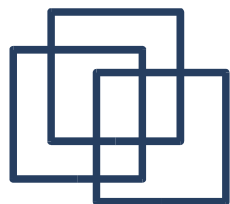
```
List<Point> points = ...;  
printAll(points);
```



Extends-bounded wildcards

- A `List<? extends Number>` represents any `List` whose element type is a subtype of `Number`.



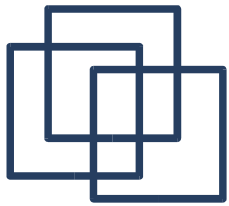


Extends-bounded wildcards

- As with unbounded wildcards, you do not know the exact element type, and so can only write `null`s.

```
List<? extends Number> nums = ...;  
nums.put(0, new Integer(0)); // Error!
```

- You do know that a `List<? extends Number>` contains some form of `Number`, so you can read numbers.

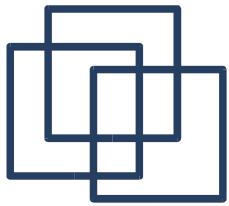


Extends-bounded wildcards

- Flashback: the BatchPrinter problem

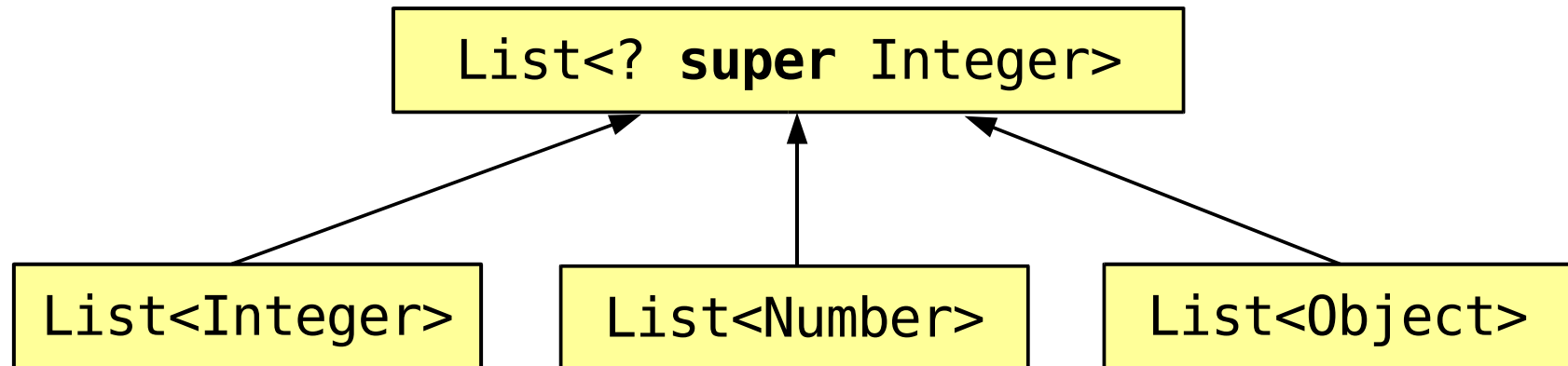
```
class BatchPrinter {  
    /* ... thread stuff ... */  
    private final List<? extends Printable> jobs;  
    BatchPrinter(List<? extends Printable> jobs) {  
        this.jobs = jobs;  
    }  
}
```

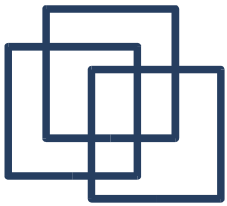
```
new BatchPrinter(psDocs).start();  
new BatchPrinter(writerDocs).start();
```



Super-bounded wildcards

- A `List<? super Integer>` represents any `List` whose element type is a supertype of `Integer`:





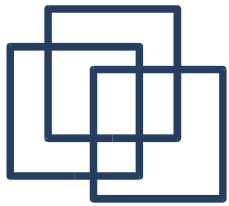
Super-bounded wildcards

- Example: the fill method replaces all elements with a given object:

```
<T> void fill(List<? super T> list, T elm) {  
    /* ... nasty ... */  
}
```

```
List<String> strs = ...;  
fill(strs, "fisk");
```

```
List<Object> objs = ...;  
fill(objs, "hest");
```



Super-bounded wildcards

- The element type of a `List<? super Integer>` may be `Object`, so you can only read `Objects`.
- On the other hand, you are allowed to write `Integers`.



Super-bounded wildcards

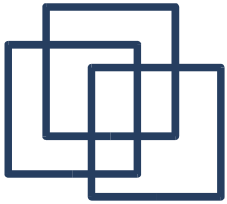
- Another example:

```
interface Comparator<T> {  
    int compare(T first, T second);  
}
```

```
<T> void sort(List<T> ts, Comparator<? super T> c) {  
    ...  
}
```

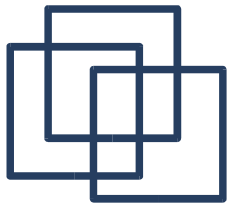
- A `Comparator<Number>` can be used to sort Integers:

```
Comparator<Number> numComp = ...  
ArrayList<Integer> ints = ...  
sort(ints, numComp)
```



Overview

- Introduction
- Why wildcards?
- Basic wildcards
- **Advanced wildcards**
 - Wildcard capture
 - Improved inference
- Summary



Wildcard Capture

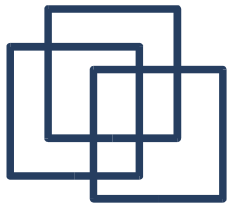
- Consider the signature of reverse:

```
public void reverse(List<?> list) {  
    /* huh? */  
}
```

- How is this method implemented?

```
Object elm = list.get(0);  
...  
list.add(elm); // Illegal!
```

- Once you have taken out an element, you can't put it back in!



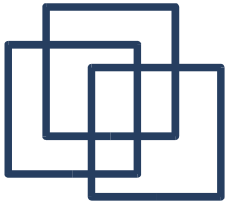
Wildcard Capture

- The trick is wildcard capture:

```
public void reverse(List<?> list) {  
    localReverse(list); // Capture  
}
```

```
private <T> void localReverse(List<T> list) {  
    T elm = list.get(0);  
    ...  
    list.add(elm); // Now this is legal!  
}
```

- Wildcard capture gives you a name for the wildcard.

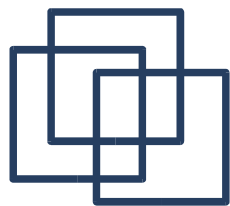


Wildcard Capture

- Capture only works in some cases

```
Stack<?> a = ..., b = ...;
<T> void swapTop(Stack<T> stack) {
    T first = stack.pop(), second = stack.pop();
    stack.push(first);
    stack.push(second);
}
swapTop(a); // Ok
```

```
<T> void swapTops(Stack<T> fst, Stack<T> snd) {
    T first = fst.pop(), second = snd.pop();
    fst.push(second);
    snd.push(first);
}
swapTops(a, b); // Error!
```



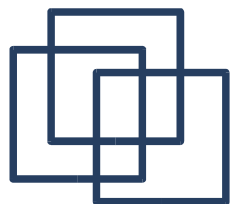
Improved Type Inference

- Type inference allows you to call polymorphic methods without giving the type parameters:

```
public <T> T choose(T a, T b) { ... }
```

```
Number n = choose(new Integer(0),  
                 new Double(0.0));
```

- Number is inferred, because it is a common supertype of Integer and Double.



Improved Type Inference

- With pure generics:

```
List<Integer> ints = ...;
```

```
List<Double> dbs = ...;
```

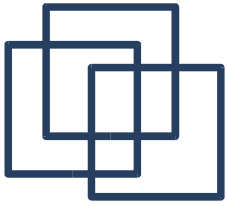
```
Object nums = choose(ints, dbs);
```

- With wildcards:

```
List<Integer> ints = ...;
```

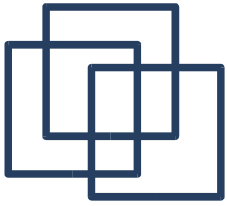
```
List<Double> dbs = ...;
```

```
List<? extends Number> nums = choose(ints, dbs);
```



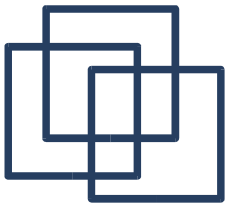
Overview

- Introduction
- Why wildcards?
- Basic wildcards
- Advanced wildcards
- **Summary**



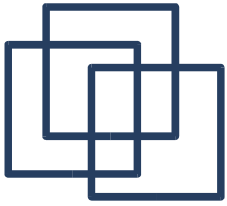
Summary

- “Plain” generic java forces you to maintain exact type argument information. This is contrary to ordinary OO.
- Wildcards make a generics mechanism “more OO” by allowing you to abstract over different parameterizations of the same class.



References

- *Torgersen, Hansen, Ernst, Ahé, Bracha and Gafter:*
Adding to the Java Programming Language
- *Bracha:*
Generics in the Java Programming
Language
<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- Java 1.5, Standard Edition Beta 1
<http://java.sun.com/j2se/1.5.0/>



Wildcards

Questions?