8. Java: Generics and Annotations
Generics and Annotations

Sources

> Gilad Bracha, *Generics in the Java Programming Language*, 2004
Roadmap

- Generics
- Annotations
- Model-Driven Engineering
Roadmap

> Generics
> Annotations
> Model-Driven Engineering
Why do we need generics?

Generics allow you to *abstract* over *types*. The most common examples are container types, the collection hierarchy.
Motivating Example – Old Style

```java
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone) stones.get(0);

public int countStones(Color color) {
    int tally = 0;
    Iterator it = stones.iterator();
    while (it.hasNext()) {
        Stone stone = (Stone) it.next();
        if (stone.getColor() == color) {
            tally++;
        }
    }
    return tally;
}
```

The cast is annoying but essential!
Motivating example – new style using generics

List is a *generic interface* that takes a type as a *parameter*.

```java
List<Stone> stones = new LinkedList<Stone>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/ stones.get(0);

public int countStones(Color color) {
    int tally = 0;
    /*no temporary*/
    for (Stone stone : stones) {
        /*no temporary, no cast*/
        if (stone.getColor() == color) {
            tally++;
        }
    }
    return tally;
}
```
Compile Time vs. Runtime Safety

Old way

```java
List stones = new LinkedList();
stones.add("ceci n’est pas un stone");
...
Stone stone = (Stone) stones.get(0);
```

No check, unsafe

Runtime error

New way

```java
List<Stone> stones = new LinkedList<Stone>();
stones.add("ceci n’est pas un stone");
...
Stone stone = stones.get(0);
```

Compile time check

Runtime is safe
Stack Example

Old way

```java
public interface StackInterface {
    public boolean isEmpty();
    public int size();
    public void push(Object item);
    public Object top();
    public void pop();
}
```

New way: we define a generic interface that takes a type parameter

```java
public interface StackInterface<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```
Linked Stack Example

```
public class LinkStack<E> implements StackInterface<E> {

    public class Cell {
        public E item;
        public Cell next;
        public Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }

    public E top() {
        assert !this.isEmpty();
        return top.item;
    }

    ...

    public class Cell {
        public E item;
        public Cell next;
        public Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }

    public E top() {
        assert !this.isEmpty();
        return top.item;
    }

    ...
```

Creating a Stack of Integers

```java
Stack<Integer> myStack = new LinkedStack<Integer>();
myStack.push(42); // autoboxing
```

When a generic is instantiated, the *actual type parameters* are substituted for the *formal type parameters*.
In Java, Foo is a subtype of Bar only if Foo’s interface strictly includes Bar’s interface. Instantiated generics normally have different interfaces. (I.e., if the type parameters are used in the public interface.)
Generics and Subtyping (II)

List<String> ls = new ArrayList<String>();

List<Object> lo = ls;

lo.add(0, new Object()); // legal?!
ls.get(0); // Not a string?!

Compile error as it is not type safe!
In other words…

- Object
- String

- List<Object>
- List<String>

The last diagram is not valid.
Wildcards

void printCollection(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}

We want a method that prints our all the elements of a collection

void printCollection(Collection<Object> c) {
    for (Object e: c){
        System.out.println(e);
    }
}

Here is a naïve attempt at writing it using generics

printCollection(stones);

Won’t compile!
What type matches all kinds of collections?

`Collection<?>`  

“collection of unknown” is a collection whose element type matches anything — a wildcard type

```java
void printCollection(Collection<?> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

`printCollection(stones);`

```
stone(java.awt.Color[r=255,g=0,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
```
Pitfalls of wildcards

String myString;
Object myObject;
List<?> c = new ArrayList<String>();

// c.add("hello world"); // compile error
// c.add(new Object()); // compile error
((List<String>) c).add("hello world");
((List<Object>) c).add(new Object()); // no compile error!

// String myString = c.get(0); // compile error
myString = (String) c.get(0);
myObject = c.get(0);
myString = (String) c.get(1); // run-time error!
Bounded Wildcards

Consider a simple drawing application to draw shapes (circles, rectangles, …)

```
Shape
  ↓
draw(Canvas)

Circle
Rectangle
```

```
Canvas
  ↓
draw(Shape)
drawAll(List<Shape>)
```

Limited to List<Shape>
A Method that accepts a List of any kind of Shape…

public void drawAll(List<? extends Shape>) {...}

Shape is the upper bound of the wildcard
More fun with generics

```java
import java.util.*;
...

public void pushAll(Collection<? extends E> collection) {
    for (E element : collection) {
        this.push(element);
    }
}

public List<E> sort(Comparator<? super E> comp) {
    List<E> list = this.asList();
    Collections.sort(list, comp);
    return list;
}
```

All elements must be `at least` an `E`.

The comparison method must require `at most` an `E`.
Roadmap

> Generics
> Annotations
> Model-Driven Engineering
Annotations

> Annotations are a *special kind of comment*
  — As with comments, annotations do not change or affect the semantics of the program, i.e. the runtime behavior.

> Annotations are *meta-descriptions*
  — Unlike comments, annotations can be accessed and used by third-party tools (e.g. JUnit) or even your program itself.
JUnit uses annotations

```java
@Before
public void setup() { ... }

@Test
public void someTest() { ... }

@Test(expected=IOException.class)
public void anotherTest() { ... }
```

JUnit uses annotations to find out which methods are test methods, and which are part of the setup. You may even pass parameters to the annotations.
Roadmap

- Generics
- Annotations
- Model-Driven Engineering
The Vision of MDE
Example: a model-driven UI

> We want a UI to edit any kind of object with any kind of properties (i.e. Model-driven Engineering)

> The example requires these steps
  — Define custom annotations for getters and setters.
  — Annotate our classes with these annotations
  — Write a UI class that access these annotations at runtime to create a custom UI
Model-driven Engineering

Model can be any kind of object with any kind of properties

Model-driven UI labels and field are automatically created based on the model
Defining our custom annotations

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface GetProperty {

    public String value();
}
```

This defines a `@GetProperty` annotation for methods. The annotation is accessible at runtime.
Annotating our domain classes

```java
@GetProperty("Titel")
public void getTitle() {
    return title;
}

@GetProperty("Autor")
public void getAuthor() {
    return author;
}
...
```
Use reflection to access the annotations of any object

```java
import java.reflect.Method;

public void printAnnotatedMethods(Object obj) {
    for (Method m : obj.getClass().getMethods()) {
        if (m.isAnnotationPresent(GetProperty.class)) {
            this.processProperty(obj, m);
        }
    }
}
```

The for loop iterates over all methods of obj’s Class. The if block is only entered for annotated methods.
Use reflection to call any method of any object

```java
import java.reflect.Method;

public void processProperty(Object obj, Method m) throws Exception {
    GetProperty g = m.getAnnotation(GetProperty.class);
    this.add(new JLabel(g.value()));
    String value = (String) m.invoke(obj);
    this.add(new JTextField(value));
}
```

We use reflection to invoke the method `m` on the object `obj`. 
What you should know!

✎ Why do I need generics?
✎ Why is casting dangerous?
✎ How do I use generics?
✎ Can I subtype a generic type?
✎ When is the Abstract Factory pattern useful?
✎ Some uses of Annotations?
✎ A Model-Driven Engineering Example
Can you answer these questions?

- *Why is `List<Object>` not the supertype of `List<String>`?*
- *Which pattern could we use to implement a Windowing Toolkit that supports multiple “look-and-feel” user interfaces?*
- *What are the advantages and disadvantages of using the Abstract Factory Pattern?*
License

http://creativecommons.org/licenses/by-sa/2.5/

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

BY: Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.