La covarianza del tipo restituito.

Il linguaggio Java 5 è dotato di una caratteristica forse passata in sordina che ha, tuttavia, un certo pregio. La dichiarazione del tipo restituito da un metodo è diventata covariante.

```
class  SuperType {
        Object getValue() {
                return new Object();
        }
}

class SubType extends SuperType {
        Integer getValue() {
                return 10;
        }
}
```

La caratteristica è interessante in tutti quei casi in cui una parte del sistema si trovi a lavorare con il sotto-tipo ed altre parti si limitino al super-tipo.

```
public class SystemPart {
        public void doSomenthing(SuperType t) {
                Object value = t.getValue();
        }
}

public class SubSystemPart {
        private SystemPart sysPart = new SystemPart();

        public SubSystemParth(SubType t) {
                int value = t.getValue() + 100;
                sysPart.doSomenthing(t);
        }
}
```

La covarianza influenza il codice byte: ciò significa che ogni volta che sia mutato il tipo restituito da un metodo "covariante" sarà necessario ricompilare i suoi sottotipi ad evitare eccezioni di conversione in esecuzione. Si prendano ad esempio i tre tipi seguenti:

```
class SuperType {
        Object getValue() {
                return new Object();
        }
}

public class SubType extends SuperType {
        Number getValue() {
                return 10.2;
        }
}

public class SubSubType extends SubType {
        Integer getValue() {
                return 4;
        }
}
```

e il programma:

```
public class Main {
        public static void main(String...args) {
                System.out.println(new SuperType().getValue());
                System.out.println(new SubType().getValue());
        }
}
```

Il cambiamento della dichiarazione di tipo restituito in SubType da Number a String:

```
public class SubType extends SuperType {
        String getValue() {
                return "Hello world";
        }
}
```

rende invalido il codice byte di SubSubType, come intuibile, per via dell'incompatibilità tra il tipo String e il tipo Integer. Il fenomeno meno evidente ma pure intuibile è che il cambiamento renda invalido anche il codice byte di Main.class. Usando javap è facile capire il perchè:

```
Compiled from "Main.java"
public class Main extends java.lang.Object{
public Main();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   new     #3; //class SuperType
   6:   dup
   7:   invokespecial   #4; //Method SuperType."<init>":()V
   10:  invokevirtual   #5; //Method SuperType.getValue:()Ljava/lang/Object;
   13:  invokevirtual   #6; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
   16:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   19:  new     #7; //class SubType
   22:  dup
   23:  invokespecial   #8; //Method SubType."<init>":()V
   26:  invokevirtual   #9; //Method SubType.getValue:()Ljava/lang/Number;
   29:  invokevirtual   #6; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
   32:  return

}
```

Il compilatore specifica (come sempre ha fatto) la diversità del tipo restituito nel codice byte di chi invochi il metodo covariante. Nulla di strano ma l'effetto è, nell'ambito della sovrascrittura dei metodi, relativamente nuovo. Nelle versioni precedenti del linguaggio Java qualsiasi mutamento ad un metodo sovrascritto era ininfluente ai fini di chi usasse quel metodo. In Java 5, poichè un metodo è considerato sovrascrivente anche quando dichiari un tipo restituito diverso dal sovrascritto e poichè il tipo restituito entra sempre a far parte del codice compilato ecco che abbiamo per le mani un dettaglio in più di cui curarci nell'ambito della compatibilità binaria.

Un caso di contravarianza?

Va premesso un rassicurante "no". Il tipo restituito è passato da invariante a covariante, fine della storia. Esiste tuttavia un caso di contravarianza che coinvolge indirettamente il tipo restituito da un metodo. Prendiamo ad esempio il tipo generico:

```
public class GenType<T> {}
```

ed un ipotetico oggetto che ne faccia uso:

```
public class SuperType {

        public GenType<? extends java.awt.Component> metodo() {
                return new GenType<java.awt.Component>();
        }
}
```

Dato un "lower bounded wildcard" (<? extends Number>) sappiamo che i suoi sottotipi sono tutti i T

compatibili in assegnamento con il limite inferiore (Number). Dunque è covariante, rispetto a metodo in SuperType, la dichiarazione:

```
public class SubType extends SuperType {

	public GenType<javax.swing.JComponent> metodo() {
		return new GenType<javax.swing.JComponent>();
	}
}
```

Integer è compatibile in assegnamento (T è T' o sottotipo di T') con Number dunque GenType<Integer> è sottotipo di GenType<? extends Number>. Le cose cambiano quando sia coinvolto un "upper bounded wildcard". Per questo tipo di dichiarazione sappiamo che sono sottotipi di GenType<? super T> tutti i tipi GenType<E> in cui E sia supertipo di T. La relazione tra i tipi dichiarati in un "upper bounded wildcard" è contravariante nel senso che il rapporto tra i tipi generici ed i tipi dei rispettivi parametri è invertito, rispetto alla compatibiità in assegnamento dei tipi generici. Rispetto alla dichiarazione di metodo:

```
public class SuperType {

	public GenType<? super javax.swing.AbstractButton> metodo() {
		return new GenType<java.awt.Component>();
	}
}
```

Un ipotetico sottotipo potrebbe restituire un GenType<AbstractButton> oppure un qualsiasi GenType<T> dove T sia un supertipo di AbstractButton. Ad esempio:

```
public class SubType {

	public GenType<java.awt.Component> metodo() {
		return new GenType<java.awt.Component>();
	}
}
```

Alla fine della fiera.

Credo sia bene far notare come il fenomeno non renda contravariante il tipo restituibile da un metodo sovrascritto. Il rapporto tra il tipo dei parametri ed i tipi generici è una questione interna a questi ultimi. Dal punto di vista dei metodi sovrascritto/sovrascrvente, il rapporto è semplicemente covariante: se il metodo M nel tipo T restituisce R allora il metodo M' che sovrascrive M in T', sottotipo di T può restituire R o un qualsiasi sottotipo di R. Se R sia un tipo generico allora emergerà una questione di contravarianza, se il tipo del parametro sia definito attraverso un "upper bounded wildcard" ma questo particolare appartiene ad una caratteristica diversa e autonoma del linguaggio Java 5. Circa l'utilità della covarianza del tipo restituito verrebbe da dire che risolva una lacuna logica. Dal punto di vista del comportamento è un bel passo avanti poter specializzare i comportamenti di un sottotipo rispetto al suo genitore non solo per quanto riguarda i dettagli di esecuzione ma anche per ciò che concerne il tipo di "risposta" offerta alle richieste. I casi applicabili sono un'infinità. In linea generale, tutti gli oggetti che trattengano un certo valore possono essere ora specializzati in sottotipi che restituiscano valori più specifici:

```
interface SomeValueHolder {
	Object getValue();
}

interface IntegerValueHolder extends SomeValueHolder {
	Integer getValue();
}
```

Il che assume sia un valore logico (IntegerValueHolder sa di conservare un intero, perchè dovrebbe restituire un Object?) sia un valore pratico, eliminando la necessità di conversioni esplicite.

.