Generics i Java

Generic: allmän, genersisk.

På menyn på en asiatisk restaurang:
”Denna rätt serveras med valfritt kött, fisk eller skalddjur”

Bakgrund

Generics i Java ger oss att skriva kod, klasser och metoder, där vi kan välja vilka typer som ska ingå först när metoden anropas eller klasssen insatansieras och samtidigt göra detta på ett typsäkert sätt. Ett av många exempel då vi har nytta av generics är när vi ska skriva en klass för att lagra samlingar av objekt.

Exempel, en lista

En klass för att lagra texter, String-objekt, skulle kunna se ut så här:

```
public class ListOfStrings {
        private String[] theList;
        private int noOfElements;
        …
        public ListOfStrings() {
                theList = new String[10];
                noOfElements = 0;
        }
        public void addString(String s) { …
        public String getString(int index) { …
        …
```

Ett objekt av denna typ kan endast lagra String-objekt (eller objekt av subtyp till String). Vill vi lagra objekt av annan typ måste vi skriva en ny version av klassen, även om det enda vi ändrar är typen på det som lagras.

Om vi istället använder oss av referenser till Javas supertyp, Object, kan objekt av vilken typ som helst lagras i listan eftersom en referens av supertyp får referera objekt av subtyp[1].

```
public class ListOfObjects {
        private Object[] theList;
        private int noOfElements;
        …
        public ListOfObjects() {
                theList = new Object[10];
                noOfElements = 0;
        }
        public void addObject(Object o) { …
        public Object getObject(int index) { …
        …
```

Vi skapar en lista och adderar objekt. Observera att vi nu kan lägga till olika typer av objekt i en och samma lista.

```
ListOfObjects list = new ListOfObjects();
list.addObject(new String("Forty-two"));
```

[1] Det var denna ”teknik” som tidigare användes i Javas Collections Framework.

```
list.addObject(new Integer(42));
```

Så långt så väl, men när vi nu hämtar objekten måste vi veta vad det är för typ av objekt, och göra en down-cast av referenstypen.

```
String s = (String) list.getObject(0);
Integer i = (Integer) list.getObject(1);
```

Programmeraren måste veta typen på objekten och göra en korrekt typomvandling. Problemet är att kontroll av om referenstyper används korrekt visserligen sker vid kompileringen men att kontrollen av huruvida det refererade *objektet* är av rätt typ sker först vid exekvering, run-time. Det finns alltså ingen möjlighet för kompilatorn att avgöra om det objekt som returneras av metoden är av en typ som matchar typomvandlingen.

En generisk lista

Det bästa vore om man vid skapandet av listan kunde ange vilken typ som ska lagras i listan, så att kompilatorn kan göra en korrekt kontroll av typerna, och så att vi slipper s.k. down-casts. Det är detta generics i Java hjälper oss med.

```
List<String> list = new List<String>();
list.add(new String("Forty-two"));
String s = list.get(0);
```

Namnet som anges inom <…> är en typ, den typ som listan ska kunna lagra. Vi behöver här inte göra några typomvandlingar, metoden get returnerar referenser av typen String i vårt fall.
Det är nu möjligt för kompilatorn göra typkontroller. Följande sats innebär, för den lista för String-objekt vi deklarerat ovan, ett syntaxfel.

```
list.add(new Integer(42)); //Fel, argumentet måste nu vara av typen String!
```

List-klassen skrivs en gång för alla, och kan sedan instansieras för att lagra olika typer av objekt vid olika tillfällen. Vi ska nu studera hur man kan skriva en sådan klass med hjälp av generics.

Syntax

När vi skriver en generisk klass eller metod anger vi en eller flera typparametrar, t.ex. <T, U>. Det är en oskriven regel att man använder stora bokstäver för typparametrar.
När klassen sedan instansieras anges vilka typer som ska användas istället för typparametern för just detta objekt, t.ex. <String, Integer>.

Exempel på en generisk klass

Klassen DataSample ska internt lagra någon form av data, av godtycklig typ. Vi deklarerar klassen med en typparameter, T, som representerar den ännu okända typen på datat.

```
public class DataSample<T> {

        private T theData;

        public DataSample(T theData) {
                this.theData = theData;
        }

        public T getData() {
                return theData;
        }

        public String toString() {
                return theData.toString();
        }
}
```

Vi kan sedan instansiera klassen och då ersätta typparametern med en verklig typ. Notera att den verkliga typ som ersätter typparametern anges både för referensen och för objektet som skapas.

```
DataSample<String> dss = new DataSample<String>("Forty-two");
String s = dss.getData();

DataSample<Integer> dsi = new DataSample<Integer>(42);
int i = dsi.getData();
```

En typparameter som deklareras som T ovan har en räckvidd som motsvarar hela den aktuella klassen.

Exempel på en generisk metod

Man kan också ange typparametrar för enskilda metoder. Räckvidden för typparametern är då naturligtvis endast aktuell metod.
Metoden nedan returnerar det mittersta elementet i en array av godtycklig typ.

```
public static <T> T getMiddleElement(T[] arr) {
        return arr[arr.length/2];
}
```

Metoden anropas så här.

```
String[] words = {"C#", "Android", "Java Micro Edition"};
…
String middle = GenericMethods.<String>getMiddleElement(words);
```

Begränsade typparametrar (bounds for type parameters)

Det finns situationer då man vill begränsa de typer en genrisk klass eller metod ska kunna instansieras för. Om man till exempel vill skriva en generisk kod som bara får instansieras för subtyper till Shape (Line, Circle, …) kan typparametern deklareras <T extends Shape>. Kompilatorn kommer att signalera ett fel om T ersätts av en typ som inte uppfyller villkoret.

Vi exemplifierar med en generisk sorteringsmetod, som kan sortera vilken typ av objekt som helst, bara typen implementerar interfacet java.lang.Comparable[2].
Så här ser sorteringsmetoden ut.

```
public static <T extends Comparable<T>> void sortObjects(T[] arr) {

        int minindex;
        T temp;

        for(int i = 0; i < arr.length - 1; i++) {
                minindex = i;
                for(int j = i + 1; j < arr.length; j++) {
                        if(arr[j].compareTo(arr[minindex]) < 0) { // compare
                                minindex = j;
                        }
                }
                // Swap
                temp = arr[i];
                arr[i] = arr[minindex];
                arr[minindex] = temp;
        }
}
```

I detta fall kompliceras bilden något av att interfacet java.lang.Comparable i sig är generiskt. Deklarationen av interfacet är deklarerat som nedan (T ska anges till samma typ som implementerande klassen, t.ex. i klassen String ska argumentet ”other” vara av typen String).

```
public interface Comparable<T> {
        abstract public int compareTo(T other);
}
```

Det extra T:et i
T extends Comparable<T>
står alltså för att samma typ som metoden sortObjects instansieras för ska användas i interfacet Comparable.

Bakom kulisserna i Java

När man i Java kompilerar en generisk klass genereras endast en version av kompilerad kod, oavsett hur många olika typer den sedan kommer att instansieras för.
Den kompilerade koden innehåller, på de ställen en typparameter refereras i källkoden, referenser till Javas supertyp Object. I Java är alltså generics inte en del av den kompilerade koden utan bara ett sätt möjliggöra för kompilatorn att kontrollera att källkoden är korrekt vad gäller ingående typer, samt naturligtvis att undvika down-casts (som ju är potentiella felkällor och dessutom irriterande).

[2] Detta anges som ”T *extends* Comparable” eftersom typen är att anse som en subtyp till Comparable.

I Språket C++ är förvisso syntaxen för generisk kod liknande den i Java, men kompileringen sker på helt annat sätt[3]. I C++ genereras nämligen separat maskinkod för varje separat typ som den generiska koden instansieras för. Detta är ett helt annat sätt att implementera generics än det sätt man valt i språket Java.

En generisk klass för en kö

Vi kan nu förbättra den kö-klass som du tidigare implementerat i en laboration. Klassen ska kunna instansieras för godtycklig typ.

```
public class Queue<T> {

    private T[] queue;
    private int noOfElements;

    public Queue(int size) {
        queue = (T[]) new Object[size];
        noOfElements = 0;
    }

    …

    /** Add an object to the end of the queue.
     */
    public void add(T element) {
        if(noOfElements >= queue.length) {
                this.resize();
        }
        queue[noOfElements++] = element;
    }

    /** Remove the head of the queue.
     */
    public T remove() throws EmptyQueueException {
        if(noOfElements <= 0) throw new EmptyQueueException();

        T temp = queue[0];
        this.pack(0);
        return temp;
    }


    …
}
```

Notera att man i Java inte kan skapa en array av godtycklig typ via en typparameter. Istället får man skapa en array av typen Object (ja just det) och göra ett down-cast. Detta görs internt i kö-klassen (i detta fall i konstruktorn) och syns inte för användaren av klassen. Anledningen att till att just arrayer inte kan skapas via typparametrar ligger utanför denna introduktion, vill du fördjupa dig i detta rekommenderar jag att studerar någon av referenserna nedan.

Alla klasser i Javas Collections Framework är implemeterade genersikt (sedan version 1.5).

[3] I C++ används beteckningen template (mall) för generics.

Läsa mer om generics

Ortacles tutorial om generics i Java:
http://docs.oracle.com/javase/tutorial/java/generics/