12. Генерички типови

Један од главних мотивишућих примера за увођење генеричких типова је развој генеричких алгоритама односно алгоритама који се могу примењивати за различите типове података. Познати пример у програмском језику C је алгоритам за сортирање `qsort` који као аргументе поред низа и његовог броја елемената добија и величину сваког елемента у низу као и показивач на функцију поређења. Ова два аргумента омогућавају прилагођавање уопштеног алгоритма за сортирање тако да буде применљив за конкретан тип података. Ова, и још многе друге примене, у Јави су реализоване путем концепта генеричког типа, односно типа података који се не фиксира приликом реализације класе, интерфејса или методе, већ се понаша попут променљиве.

12.1. Сирови тип

У досадашњем тексту користили су се искључиво сирови типови података који подразумевају конкретно навођење специфичног типа податка.

Пример 1. Описати класу која представља кутију у коју може да се убаци произвољни објекат. Потом тестирати рад кутије тако што се различити типови убацују у кутију, а потом из ње извлаче. □

```
package rs.math.oop.g12.p01.genericiKutijaSirova;

public class KutijaSirova {
	// можемо убацити било који објектни тип овде
	private Object vrednost;

	// било који објектни тип се прихвата
	// и имплицитно конвертује у Object
	void postaviVrednost(Object vrednost) {
		this.vrednost=vrednost;
	}

	// изгубили смо информацију о правој природи објектног типа
	// тако да једино што можемо да вратимо је тип Object
	Object uzmiVrednost() {
		return this.vrednost;
	}

	public static void main(String[] args) {
		KutijaSirova kutija1 = new KutijaSirova();
		kutija1.postaviVrednost("Текст");
		// мора експлицитна конверзија
		String tekst1 = (String) kutija1.uzmiVrednost();
		System.out.println(tekst1);
		// тако да је одговорност на програмеру да зна шта је у кутији
		Integer broj1 = (Integer) kutija1.uzmiVrednost(); // изузетак
	}
}
```

Приметимо да се у кутију може убацити било који објектни тип који се путем методе `postaviVrednost` имплицитно конвертује у најопштији могући сирови тип `Object`.

Будући да се на овај начин губи информација о правом типу поља `vrednost`, она се не може повратити те метода за узимање вредности `uzmiVrednost` враћа такође објекат типа `Object`. Овај механизам са употребом најопштијег сировог типа `Object` омогућава флексибилност по питању типова пошто се сваки други објектни тип може конвертовати у општији. Међутим, проблем настаје у другом смеру, јер је програмер дужан да зна праву природу објекта у кутији и да изврши експлицитну конверзију у складу са тим. Овде се види да компајлер омогућава експлицитну конверзију садржаја кутије у било који објектни тип тако да ће се проблем са конверзијом у погрешан тип `Integer` евидентирати тек у фази извршавања у виду изузетка `ClassCastException`.

```
Текст
Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be
cast to class java.lang.Integer (java.lang.String and java.lang.Integer are in module
java.base of loader 'bootstrap')
        at
rs.math.oop.g12.p01.genericiKutijaSirova.KutijaSirova.main(KutijaSirova.java:26)■
```

12.2. Појам, дефинисање и предности генеричког типа

Генерички тип омогућава да при дефинисању класа, интерфејса и метода сами типови (тј. класе и интерфејси) буду параметри. Дакле, генерички типови се понашају слично као формални параметри при дефинисању метода. Програмирање коришћењем генеричких типова има следеће предности:

1. строжија контрола типа приликом превођења Јава програма;
2. елиминација експлицитне конверзије типа (кастовања);
3. омогућавање да се имплементирају генерички алгоритми.

Генеричка класа се дефинише на следећи начин:

```
imeKlase<T1,T2,...,Tn>{/* ... */}
```

Секција са параметрима који представљају типове, ограничена са знацима < и > следи непосредно иза имена класе. У тој секцији се специфицирају параметри који представљају типове `T1`, `T2`, ..., и `Tn`. По конвенцији, параметри су означени једним великим словом. Најчешће се користе следеће ознаке:

- E -Елеменат (енг. Element)
- K –Кључ (енг. Key)
- N –Број (енг. Number)
- T –Тип (енг. Type)
- V –Вредност (енг. Value)
- S,U,W итд. –други, трећи, четврти итд.

Пример 2. Описати класу која представља генеричку кутију у коју може да се убаци произвољни објекат, али да се притом може имлицитно закључити шта је унутра. □

```
package rs.math.oop.g12.p02.genericiKutijaGenericka;

public class KutijaGenericka<T> {
        // можемо убацити било који објектни тип овде
        private T vrednost;

        // прихвата се онај објектни тип за кога је направљена кутија
        // одлука о типу се доноси приликом инстанцирања кутије
        public void postaviVrednost(T vrednost) {
```

```
                this.vrednost=vrednost;
        }

        // нисмо изгубили информацију о правој природи објектног типа
        // тако да враћамо прецизан тип, а не његово уопштење
        public T uzmiVrednost() {
                return this.vrednost;
        }

        public static void main(String[] args) {
                KutijaGenericka<String> kutija1 = new KutijaGenericka<String>();
                kutija1.postaviVrednost("Текст");
                // не треба експлицитна конверзија
                // тако да је програмер растерећен одговорности да то мора да зна
                String tekst1 = kutija1.uzmiVrednost();
                System.out.println(tekst1);
                // компајлер не допушта да у кутију убацимо нешто што није String
                // па нам тиме помаже у провери семантичке коректности програма
                //kutija1.postaviVrednost(45);
        }
}
```

Генеричка кутија је параметризована са `T`. Параметар има опсег важења у целом телу класе, а то се види на 3 места у овом примеру: 1) код декларације типа променљиве `vrednost`, 2) код типа аргумента методе `postaviVrednost` и 3) код повратне вредности методе `uzmiVrednost`. За разлику од класе `KutijaSirova`, објекти класе `KutijaGenericka` кроз конструктор подешавају тип података за поље `vrednost`. Ово се спроводи тако што се у генеричком типу `KutijaGenericka<T>`, `T` замењује са конкретним типом `T` (тзв. генерички позив типа) тако да и цео генерички тип добија своју конкретизацију. Ово надаље елиминише потребу за експлицитном конверзијом података који се узима из кутије, а такође спречава програмера да за вредност у кутији постави нешто што није `String`.

```
Текст■
```

Пример 3. Описати генеричку класу за представљање уређеног пара при чему координате уређеног пара не морају нужно припадати истом домену односно бити исти тип. □

```
package rs.math.oop.g12.p03.genericiUredjeniPar;

public class UredjeniPar<T, S>{
        private T vrednost1;
        private S vrednost2;

        public UredjeniPar(T vrednost1, S vrednost2) {
                this.vrednost1 = vrednost1;
                this.vrednost2 = vrednost2;
        }

        public T getVrednost1() {
                return vrednost1;
        }

        public S getVrednost2() {
                return vrednost2;
```

```
	}

	@Override
	public String toString() {
		return "("+vrednost1+", "+vrednost2+")";
	}

	public static void main(String[] args) {
		UredjeniPar<Integer, Integer> par1 =
				new UredjeniPar<Integer, Integer>(10, 20);
		UredjeniPar<Integer, String> par2 = new UredjeniPar<>(30, "Пример текст");
		// не пролази компилацију због неслагања очекиваног и прослеђеног типа
		//UredjeniPar<Integer, Integer> par3 = new UredjeniPar<>(30, 14.0);
		System.out.println(par1);
		System.out.println(par2);
	}
}
```

Навођењем два различита параметра `T` и `S` омогућено је креирање уређених парова који могу имати различите типове координата, али ово не искључује могућност да `T` и `S` буду исти тип. Приликом креирања другог уређеног пара `par2`, примећује се да у позиву конструктора изостају подешавања вредности параметара. Јава компајлер ово допушта, јер је из контекста односно декларације објектне променљиве јасно да то морају бити цео број и текст. Креирање уређеног пара `par3` није дозвољено, јер се тип другог аргумента прослеђеног у конструктор не поклапа са декларисаним типом.

```
(10, 20)
(30, Пример текст)■
```

12.2.1. Стек као генерички интерфејс и његова имплементација

Генерички интерфејси омогућавају навођење генеричких метода. Надаље, приликом њихове имплементације од стране класа та генеричност се може задржати.

Пример 4. Дефинисати генерички интерфејс за стек структуру података. Након тога дефинисати класу која имплементира генерички стек интерфејс помоћу низа. □

```
package rs.math.oop.g12.p04.genericiStek;

interface StekInterfejs<T> {
	boolean jePrazan(); // празан стек
	int velicina(); // број елемената у стеку
	void dodaj(T element); // додај на врх
	T vrh();	// елемент на врху стека
	void ukloni(); // скини елемент са врха
}
```

```
package rs.math.oop.g12.p04.genericiStek;

public class StekPrekoNiza<T> implements StekInterfejs<T> {
	private T[] stekNiz;
	private int stekIndeks; // показује на врх стека
	private int kapacitet;

	public StekPrekoNiza() {
```

```
        kapacitet = 5;
        stekIndeks = -1;
        stekNiz = alocirajStekNiz();
    }

    /*
     * ова наредба (тзв. анотација) елиминише упозорење компајлера због експлицитне
     * конверзије низа објеката у низ T елемената
     */
    @SuppressWarnings("unchecked")
    private T[] alocirajStekNiz() {
        System.out.println("Повећавам капацитет стека на "+kapacitet);
        /*
         * будући да ће сва интеракција између низа T[] и "остатка света"
         * ићи преко метода dodaj и ukloni који гарантују поштовање
         * генеричког типа овде свесно вршимо експлицитну конверзију
         * јер креирање низа генеричких типова није подржано
         */
        return (T[]) new Object[kapacitet];
    }

    @Override
    public boolean jePrazan() {
        return stekIndeks == -1;
    }

    @Override
    public int velicina() {
        return stekIndeks + 1;
    }

    @Override
    public void dodaj(T element) {
        System.out.println("Додајем "+element);
        if (stekIndeks + 1 == kapacitet) {
            // нема више места па дуплирамо капацитет
            kapacitet *= 2;
            T[] noviStekNiz = alocirajStekNiz();
            // преписујемо податке у новокреирани стек низ
            for (int i = 0; i <= stekIndeks; i++)
                noviStekNiz[i] = stekNiz[i];
            stekNiz = noviStekNiz;
        }
        stekNiz[++stekIndeks] = element;
    }
```

```
    @Override
    public T vrh() {
        if(!jePrazan())
            return stekNiz[stekIndeks];
        else {
            // овде ћемо уместо изузетка вратити null,
            // али има смисла и да се избаци изузетак
            System.out.println("Стек је празан па нема смисла гледање врха.");
            return null;
```

```
		}
	}

	@Override
	public void ukloni() {
		if(!jePrazan()) {
			System.out.println("Уклањам "+stekNiz[stekIndeks]);
			// бришемо референцу да би GC могао да почисти меморију
			stekNiz[stekIndeks]=null;
			stekIndeks--;
		}else {
			// овде нећемо ништа урадити,
			// али има смисла и да се избаци изузетак
			System.out.println("Стек је празан па нема смисла уклањање.");
		}
	}

	public static void main(String[] args) {
		StekPrekoNiza<Integer> stek = new StekPrekoNiza<>();
		System.out.println("Врх: "+stek.vrh());
		stek.dodaj(34);
		stek.dodaj(23);
		stek.dodaj(11);
		System.out.println("Врх: "+stek.vrh());
		stek.ukloni();
		System.out.println("Врх: "+stek.vrh());
		stek.dodaj(112);
		stek.dodaj(-134);
		stek.dodaj(111);
		stek.dodaj(345);
		System.out.println("Величина: "+stek.velicina());
		stek.ukloni();
		stek.ukloni();
		stek.ukloni();
		stek.ukloni();
		stek.ukloni();
		stek.ukloni();
		stek.ukloni();
	}
}
```

Најпре је дефинисан генерички стек интерфејс са методама карактеристичним за стек структуру: додавање на врх стека, уклањање са врха и провера тренутног врха стека. Поред тога, у интерфејсу су наведене још две додатне методе за проверу да ли је стек празан и за тренутну величину стека. Након тога је реализована класа која имплементира стек интерфејс уз употребу низа и додатних поља за представљање тренутног капацитета стека као и за позицију (индекс) врха стека. Иницијално стање врха стека је -1 што значи да је стек празан. Како се додају елементи у стек, индекс се повећава тако да означава врх стека. Приметити да је за потребе алоцирања иницијалног (и даље реалокације) низа са генеричким елементима искоришћено заправо креирање низа објеката уз експлицитну конверзију у низ генеричких типова. Ово је урађено будући да Јава не допушта конструкцију `new T[...]`. Имајући у виду да се низ `T[]` надаље користи опрезно и мења искључиво кроз позиве метода `dodaj` и

`ukloni`, које се обе ослањају на генеричност, ова експлицитна конверзија је оправдана. Испис из горе описаног програма је следећи:

```
Повећавам капацитет стека на 5
Стек је празан па нема смисла гледање врха.
Врх: null
Додајем 34
Додајем 23
Додајем 11
Врх: 11
Уклањам 11
Врх: 23
Додајем 112
Додајем -134
Додајем 111
Додајем 345
Повећавам капацитет стека на 10
Величина: 6
Уклањам 345
Уклањам 111
Уклањам -134
Уклањам 112
Уклањам 23
Уклањам 34
Стек је празан па нема смисла уклањање.■
```

12.3. Локално-генерички метод

Опсег дејства генеричког параметра не мора нужно обухватати читаву класу или читав интерфејс. У Јави је могућа и локалнија употреба генеричког понашања на нивоу појединачних метода. То наравно не значи да до сада уведене методе у којима се користио генерички параметар из класе или интерфејса нису генеричке. Како бисмо разликовали ова два типа генеричких метода, они код којих параметар за тип није потекао из класе или интерфејса ћемо звати локално-генерички методи.

Пример 5. Реализовати статички локално-генерички метод за претрагу над низом генеричких типова. □

```
package rs.math.oop.g12.p05.genericiPretragaNiza;

public class PretragaNiza {

	public static <T> int pretrazi(T[] niz, T element) {
		for(int i=0; i<niz.length; i++)
			if(niz[i].equals(element))
				return i;
		return -1;
	}

	public static void main(String[] args) {
		Integer[] nizCelih = new Integer[] {2,43,22,11,243,253,64};
		int element1 = 34;
		int element2 = 243;
		System.out.printf("Позиција елемента %d је %d.%n", element1,
				pretrazi(nizCelih, element1));
		System.out.printf("Позиција елемента %d је %d.%n", element2,
```

```
                                pretrazi(nizCelih, element2));
        }
}
```

Приметимо да је параметар `T` сада први пут декларисан непосредно пред повратни тип и да он нема утицај на остатак чланица класе `PretragaNiza`. Такође, линеарне алгоритам претраге нема никакве специјалне захтеве по питању карактеристика објеката `T`, једино што је неопходно је поређење на једнакост које је неупитно подржано за све `T` будући да је `equals` дефинисан на највишем нивоу, односно у класи `Object`. Испис након извршавања је:

```
Позиција елемента 34 је -1.
Позиција елемента 243 је 4.■
```

12.4. Ограничења за типове

Тип понекад мора испуњавати одређене додатне критеријуме зарад реализације генеричког алгоритма те не може увек представљати произвољну подкласу класе `Object` као што је то био случај у досадашњим примерима. На пример, за потребе реализације алгоритам бинарне претраге, сортирања, тражења минимума, максимума, итд. јасно је да типови морају бити упоредиви. Нотација

```
<T extends TipKojiOgranicava>
```

изражава да параметарски тип `T` треба да буде подтип типа који га ограничава. При томе, и параметарски тип `T` и тип који ограничава могу бити и класе и интерфејси. Кључна реч `extends` описује дакле ограничавање одозго, а не нужно наслеђивање. Иако би боље решење било да је за ово ограничавање уведена нова кључна реч (главни кандидат је била реч `sub`) дизајнери Јаве су се ипак одлучили да је једноставније искористити већ постојећу кључну реч која је најближа по семантици циљном механизму.

Тип може истовремено бити ограничен са највише једном класом и произвољним бројем интерфејса. У том случају се ограничења набрајају на следећи начин.

```
<T extends TipKojiOgranicava1 & TipKojiOgranicava2 & ...>
```

Пример 6. Реализовати статички локално-генерички метод за тражење минималног елемента у низу. Након тога применити тражење минимума над низом објеката типа `String`, али и над низом објеката раније уведеног генеричког типа уређени пар. С тим у виду потребно је "дорадити" класу за уређени пар тако да имплементира генерички интерфејс `Comparable` при чему се поређење врши по принципу да се најпре пореди по првој координати па потом, ако има потребе, и по другој. □

```
package rs.math.oop.g12.p06.genericiMinimalniElementNiza;

public class UredjeniParUporediv<S extends Comparable<S>, T extends Comparable<T>>
        implements Comparable<UredjeniParUporediv<S, T>>{
        private T vrednost1;
        private S vrednost2;

        public UredjeniParUporediv(T vrednost1, S vrednost2) {
                this.vrednost1 = vrednost1;
                this.vrednost2 = vrednost2;
        }
```

```
	public T getVrednost1() {
		return vrednost1;
	}

	public S getVrednost2() {
		return vrednost2;
	}

	@Override
	public String toString() {
		return "("+vrednost1+", "+vrednost2+")";
	}

	@Override
	public int compareTo(UredjeniParUporediv<S, T> o) {
		int uredjenjeS = this.vrednost1.compareTo(o.vrednost1);
		if(uredjenjeS!=0)
			return uredjenjeS;
		return this.vrednost2.compareTo(o.vrednost2);
	}
}
```

У прилагођеној верзији генеричког уређеног пара наведене су две нове битне ствари: 1) да генерички уређени пар имплементира интерфејс за упоређивање са другим генеричким паровима `Comparable<UredjeniParUporediv<S, T>>` и 2) да је сваки од типова `S` и `T` такође упоредив тип, нпр. за `S` је то `S extends Comparable<S>`. Ставка 2) стога ограничава одозго генерички параметар интерфејсом `Comparable`. Због тога је надаље могуће реализовати методу compareTo на тражени начин па се најпре позива `compareTo` над првим елементом уређеног пара, па ако је по том критеријуму нејасно уређење, онда се користи и други критеријум.

```
package rs.math.oop.g12.p06.genericiMinimalniElementNiza;

public class MinimalniElementNiza {

	public static <T extends Comparable<T>> T nadjiMinimum(T[] niz) throws Exception{
		if(niz.length==0)
			throw new Exception("Низ је празан - минимум нема смисла.");
		T minimum = niz[0];
		for(T element: niz)
			if(element.compareTo(minimum)<0)
				minimum = element;
		return minimum;
	}

	public static void main(String[] args) {
		String[] stringovi = new String[] {"Паја", "Ана", "Мика",
				"Марија", "Пера"};
		// није могуће користити оператор new за креирање низа генеричких типова
		// па вршимо експлицитну конверзију и наговештавамо компајлеру
		// да не треба да исписује упозорење због тога
		@SuppressWarnings("unchecked")
		UredjeniParUporediv<Integer, Integer>[] parovi =
			(UredjeniParUporediv<Integer, Integer>[]) new UredjeniParUporediv[]
			{
```

```
                        new UredjeniParUporediv<Integer, Integer>(46, 21),
                        new UredjeniParUporediv<Integer, Integer>(10, 21),
                        new UredjeniParUporediv<Integer, Integer>(10, 19),
                        new UredjeniParUporediv<Integer, Integer>(15, 21),
                };
        // уређени пар који садржи нешто што и само није упоредиво као координату
        // се не може креирати због ограничења на тип по обе координате
        // у дефиницији упоредивог уређеног пара
        // UredjeniParUporediv<Color, Integer> par; // не компајлира се
        try {
                String minString = nadjiMinimum(stringovi);
                System.out.println(minString);
                UredjeniParUporediv<Integer, Integer> minPar =
                                nadjiMinimum(parovi);
                System.out.println(minPar);
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
    }

}
```

Генеричка метода за тражење минималног елемента такође користи ограничавање параметра типа одозго интерфејсом `Comparable<T>` будући да је за тражење минимума потребно искористити упоређивање елемената низа. За демонстрацију рада методе искоришћен је низ стрингова, а након тога и низ уређених парова за које је показано на који начин реализују упоредивост. Треба приметити да није могуће инстанцирати низ генеричких типова `new UredjeniParUporediv<Integer, Integer>[...]` - слично као што је раније показано да није могуће позвати ни конструкцију `new T[...]`. Да би се то заобишло, употребљен је конструктор за креирање сировог низа уређених парова `new UredjeniParUporediv[...]` а након тога је извршена експлицитна конверзија у `UredjeniParUporediv<Integer, Integer>[]`. Испис који производи овај програм је:

```
Ана
(10, 19)■
```

12.5. Генерици и виртуелна машина

Кад год се дефинише генерички тип, у позадини (на нивоу виртуелне машине) аутоматски бива обезбеђен и одговоарајући сирови тип за њега. Име сировог типа је исто као име генеричког типа, само што су уколоњени параметри који представљају типове. Променљиве које представљају типове су просто замењене са типовима који их ограничавају или са `Object` типом (ако за те променљиве није било ограничења). На пример, за раније уведену класу `KutijaGenericka<T>` где `T`, између осталог, параметризује тип поља `vrednost`, у позадини се креира сиров тип попут `KutijaSirova` где је тип поља `vrednost` једнак `Object`. Додатно, компајлер убацује експлицитну конверзију ка одговарајућем типу `T` на свим местима где је то потребно. Другим речима, компајлер у фази препроцесирања преводи генерички код у код са сировим типовима на сличан начин на који би то и добар програмер урадио. Када се користи бар једно ограничење за параметар `T`, уместо замењивањем појављивања типа `T` са `Object` врши се замењивање првим наведеним ограничењем. За преостала ограничења, ако их има

више од једног, у коду се на одговарајућим местима спроводе експлицитне конверзије како би се добио жељени тип.

Пример 7. Класу за упоредиви уређени пар реализовати кроз употребу сирових типова. Такође на сличан начин реализовати и статичку методу за тражење минималног елемента низа и онда је применити над низом уређених парова. □

```
package rs.math.oop.g12.p07.genericiUredjeniParUporedivSirov;

// због ограничења на тип уместо класом Object сва појављивања типа се замењују
// негенеричким интерфејсом Comparable
public class UredjeniParUporedivSirov implements Comparable{
		private Comparable vrednost1;
		private Comparable vrednost2;

		public UredjeniParUporedivSirov(Comparable vrednost1,
				Comparable vrednost2) {
			this.vrednost1 = vrednost1;
			this.vrednost2 = vrednost2;
		}

		public Comparable getVrednost1() {
			return vrednost1;
		}

		public Comparable getVrednost2() {
			return vrednost2;
		}

		@Override
		public String toString() {
			return "("+vrednost1+", "+vrednost2+")";
		}

		@Override
		public int compareTo(Object o) {
			// поређење је гарантовано и у овој варијанти једино што оно
			// није реализовано кроз генеричку compareTo методу
			UredjeniParUporedivSirov par = (UredjeniParUporedivSirov) o;
			int uredjenjeS = this.vrednost1.compareTo(par.vrednost1);
			if(uredjenjeS!=0)
				return uredjenjeS;
			return this.vrednost2.compareTo(par.vrednost2);
		}
}
```

Будући да је циљ био да се генеричност потпуно замени сировим типовима, али да се притом задржи функционалност, изведене су следеће замене типова. Уместо досадашњег потписа класе

```
public class UredjeniParUporediv<S extends Comparable<S>, T extends Comparable<T>>
	implements Comparable<UredjeniParUporediv<S, T>>
```

нови потпис изгледа овако:

```
public class UredjeniParUporedivSirov implements Comparable
```

уз додатну експлицитну конверзију унутар методе `compareTo`

```
UredjeniParUporedivSirov par = (UredjeniParUporedivSirov) o;
```

и наравно замене свих појављивања типова `S` и `T` типом `Comparable`.

```
package rs.math.oop.g12.p07.genericiUredjeniParUporedivSirov;

public class MinimalniElementNizaSirov {

	public static Comparable nadjiMinimum(Comparable[] niz) throws Exception{
		if(niz.length==0)
			throw new Exception("Низ је празан - минимум нема смисла.");
		Comparable minimum = niz[0];
		for(Comparable element: niz)
			if(element.compareTo(minimum)<0)
				minimum = element;
		return minimum;
	}

	public static void main(String[] args) {
		UredjeniParUporedivSirov[] parovi =new UredjeniParUporedivSirov[]
			{
				new UredjeniParUporedivSirov(46, 21),
				new UredjeniParUporedivSirov(10, 21),
				new UredjeniParUporedivSirov(10, 19),
				new UredjeniParUporedivSirov(15, 21),
			};
		try {
			UredjeniParUporedivSirov minPar =
					(UredjeniParUporedivSirov) nadjiMinimum(parovi);
			System.out.println(minPar);
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

Сличне измене је претрпео и код за тражење минималног елемента низа. Резултат извршавања сада сирове методе за тражење минималног елемента над низом уређених парова је исти као и раније:

```
(10, 19)■
```

12.6. Генерици и наслеђивање

Поставља се такође питање у каквој су релацији две генеричка објекта уколико су њихови параметри у релацији наслеђивања? На пример, ако су параметри типови `Integer` и `Object` и јасно је да је `Integer` подкласа од `Object`, да ли је онда и `KutijaGenericka<Integer>` подкласа од `KutijaGenericka<Object>`? Одговор је: не, објекти ових класа нису ни у каквој релацији те се објектна променљива типа `KutijaGenericka<Integer>` не може сакрити (уопштити) имплицитном конверзијом у објектну променљиву типа `KutijaGenericka<Object>` за разлику од сценарија у којем се објектна променљива типа `Integer` може имплицитно конвертовати у објектну променљиву типа `Object`.

Са друге стране уколико не посматрамо наслеђивање између параметара већ између класа које се параметризују (у овом случају `KutijaGenericka<T>`) онда се може успоставити релација наслеђивања што демонстрира наредни пример.

Пример 8. Из претходно уведене класе `KutijaGenericka<T>` извести генеричку подкласу која омогућава и прослеђивање боје кутије кроз конструктор. Након тога тестирати имплицитну конверзију између објектних променљивих из смера подкласе ка надкласи. □

```
package rs.math.oop.g12.p08.genericiKutijaGenerickaObojena;

import java.awt.Color;
import rs.math.oop.g12.p02.genericiKutijaGenericka.KutijaGenericka;

public class KutijaGenerickaObojena<T> extends KutijaGenericka<T>{
	private Color boja;

	public KutijaGenerickaObojena(Color boja){
		super();
		this.boja=boja;
	}

	public Color getBoja() {
		return boja;
	}

	public static void main(String[] args) {
		KutijaGenerickaObojena<String> kutijaObojena =
				new KutijaGenerickaObojena<String>(Color.red);
		kutijaObojena.postaviVrednost("Текст");
		// имплицитна конверзија у општији тип
		KutijaGenericka<String> kutija = kutijaObojena;
		System.out.println(kutija.uzmiVrednost());
	}
}
```

Пример се компајлира и уредно се исписује садржај кутије те константујемо да је релација наслеђивања задржана у овом сценарију.

```
Текст■
```

12.7. Резиме

Генерички типови су моћан механизам који омогућава писање генеричких алгоритама те растерећивање програмера од писања истог или сличног кода велики број пута. Јава омогућава да се генеричност дефинише на глобалном или локалном нивоу, односно на нивоу целих класа или интерфејса или само на нивоу појединачних метода. Кроз механизам ограничавања могућих типова програмер може додатно прецизирати које предуслове тип мора да испуњава како би се класа/интефејс/метода могла њиме параметризовати. Иако делује да је систем генеричких типова врло чврсто спрегнут са Јава језиком и Јава компајлером, испоставља се да генерички типови заправо представљају синтаксну олакшицу програмеру (слично као и набројиви типови) те се кроз фазу препроцесирања овакав код трансформише у сирови (негенерички) код који даље улази у прави процес компилације. Без обзира на то, овако постављен систем генеричких типова је од изузетне користи, јер омогућава бољу контролу типова и избегавање (не и потпуну елиминацију) експлицитних конверзија које очигледно пребацују одговорност за појаву грешка на програмера и могу се детектовати тек у фази извршавања.

12.8. Питања и задаци