# Generičko programiranje u Javi

# Uvod u generičko programiranje

- Generičkim programiranjem moguće je napisati metodu koja može primiti različite tipove parametara bez potrebe za *overloadanjem*
- *Overloadane* metode se često koriste kako bi omogućile slične funkcionalnosti s različitim tipovima ulaznih parametara
- Pomoću generičkih metoda moguće je smanjiti količinu programskog koda i smanjiti broj potreba za obavljanjem *cast* operacija
- Bez korištenja generičkih principa programiranja nužno je korištenje "univerzalne" klase "java.lang.Object", koja se onda pomoću *cast* operacije pretvara u ostale tipove

# Primjer metoda bez korištenja generičkog programiranja

- Kako metode moraju primati različite tipove, potrebno je napisati više različitih *overloadanih* metoda:

```
public void ispisiPolje(String[] polje) {
    for (String str : polje) {
        System.out.println(str);
    }
}

public void ispisiPolje(Integer[] polje) {
    for (Integer str : polje) {
        System.out.println(str);
    }
}

public void ispisiPolje(Double[] polje) {
    for (Double str : polje) {
        System.out.println(str);
    }
}
```

# Primjer klase bez generičkog programiranja (1/5)

- Ako je potrebno napisati klasu "Par" koja će sadržavati dva ista predmeta, pri čemu je jedan lijevi, a drugi desni, to je potrebno napraviti na sljedeći način:

```
public class Par {
   private Object lijeva, desna;
   public Par(Object lijeva, Object desna) {
      this.lijeva = lijeva;
      this.desna = desna;
   }
   public Object getLijeva() {return lijeva;}
   public Object getDesna() {return desna;}
   public String toString() {
      return "(lijeva, desna) = (" + lijeva + "," + desna + ")";
   }
}
```

# Primjer klase bez generičkog programiranja (2/5)

- Ako bi klase čiji objekti se dodaju u objekte klase "Par" izgledale ovako:

```
public class Cipela {
   public String toString() {
      return "Cipela";
   }
}
```

```
public class Nausnica {
   public String toString() {
      return "Naušnica";
   }
}
```

- Onda bi umetanje objekata klase "Cipela" u objekte klase "Par" izgledalo ovako:

```
Cipela c1 = new Cipela(); Cipela c2 = new Cipela();
Par parCipela = new Par(c1, c2);
System.out.println("1. Par: " + parCipela);
```

# Primjer klase bez generičkog programiranja (3/5)

- Slično i za objekte klase "Nausnica":

```
Nausnica n1 = new Nausnica();
Nausnica n2 = new Nausnica();
Par parNausnica = new Par(n1, n2);
System.out.println("2. Par: " + parNausnica);
```

- Ako bi bilo potrebno dohvatiti npr. "lijevu naušnicu", to je moguće napraviti na sljedeći način:

```
Nausnica ln = (Nausnica) parNausnica.getLijeva();
System.out.println("Lijevi član 2. para je :"
 + ln);
```

- Kako metoda "**getLijeva**" vraća objekt tipa "Object", potrebno je operacijom *cast* naznačiti u kojem se tipu objekta radi, kako bi ispis u konzolu bio ispravan

# Primjer klase bez generičkog programiranja (4/5)

- Ispis u konzolu izgleda ovako:

  **1. Par: (lijeva, desna) = (Cipela, Cipela)**

  **2. Par: (lijeva, desna) = (Nausnica, Nausnica)**

  **Lijevi član 2. para je : Naušnica**

- Osim potrebe za korištenjem *cast* operacije, loša strana navedenog rješenja je mogućnost dodavanja različitih članova u klasu "Par":

```
Nausnica n1 = new Nausnica();
Cipela c2 = new Cipela();
Par neobicanPar = new Par(n1, c2);
System.out.println("Neobičan par: " + neobicanPar);
```

- Kod dohvaćanja članova para iz objekta "neobicanPar" potrebno je biti vrlo oprezan, jer su različitih tipova

# Primjer klase bez generičkog programiranja (5/5)

- Ako se želi dohvatiti lijevi član, potrebno je koristiti sljedeću *cast* naredbu:

  **Nausnica n1 = (Nausnica) neobicanPar.getLijeva();**
- Međutim, ukoliko programer slučajno pogriješi i umjesto u tip "Nausnica" upotrijebi cast operaciju u tip Cipela, dogoditi će se iznimka ***ClassCastException***:

  **Cipela c1 = (Cipela) neobicanPar.getLijeva();**
- Pogreška u programskom kodu dolazi do izražaja tek kod izvođenja programskog koda, a prevoditelj u fazi prevođenja same klase ne javlja nikakvu pogrešku

# Primjer generičke metode

- Metodu "ispisiPolje" moguće je napisati tako da može primiti polje s bilo kojim tipovima podataka, pri čemu se tip označava generičkom oznakom "E":

```
public <E> void ispisiPolje(E[] polje) {
    for (E element : polje) {
        System.out.println(element);
    }
}
```

  - Navedena generička metoda može biti pozvana s različitim tipovima ulaznih parametara (polja)
  - Pozivi generičkih metoda su identični pozivima *overloadanih* metoda

# Svojstva generičkih metoda

- Sve deklaracije generičkih metoda definiraju parametar koji se nalazi između uglatih zagrada „<“ i „>“ i stavlja se ispred povratnog tipa generičke metode
- U uglate zagrade moguće je staviti više tipova odvojenih zarezima
- Jedan parametar se može pojaviti samo jednom unutar uglatih zagrada, ali se može pojaviti više puta među ulaznim parametrima
- Na primjer, ako je potrebno napisati metodu koja će spajati dva polja proizvoljnog tipa i vratiti treće polje tog tipa, to je moguće implementirati na sljedeći način:

```
public <E> E[] spojiPolja(E[] polje1,
                          E[] polje2)
```

# Primjer definiranja generičke klase (1/2)

- Ako je već spomenutu klasu "Par" potrebno preoblikovati na način da bude generička i da može primati samo dva objekta istog tipa, to je moguće napraviti na sljedeći način:

```
public class GenerickiPar<T> {
   private T lijeva, desna;
   public GenerickiPar(T lijeva, T desna) {
      this.lijeva = lijeva;
      this.desna = desna;
   }
   public T getLijeva() {return lijeva;}
   public T getDesna() {return desna;}
   public String toString() {
      return "(lijeva, desna) = (" + lijeva + "," + desna + ")";
   }
}
```

# Primjer definiranja generičke klase (2/2)

- U tom slučaju kreiranje objekta klase "GenerickiPar" izgledalo bi ovako:

```
Nausnica n1 = new Nausnica();
Nausnica n2 = new Nausnica();
GenerickiPar<Nausnica> parNausnica =
   new GenerickiPar<>(n1, n2);
System.out.println("2. Par: " + parNausnica);
```

- Ukoliko se u objekt "parNausnica" pokušaju dodati dva objekta različitih tipova, npr. "Nausnica" i "Cipela", prevoditelj dojavljuje pogrešku

# Nedefinirani tipovi

- engl. *Raw types*
- Ukoliko se kod instanciranja objekta klase **`GenerickiPar`** ne navede tip podatka u uglatim zagradama, kompajler će implicitno koristiti tip **`Object`** na svim mjestima gdje je naveden generički tip **`<T>`**
- Time je omogućena unazadna kompatibilnost (engl. *backward compatilibity*) s prošlim verzijama Jave

```
GenerickiPar generickiPar = new GenerickiPar(n3, n1);
```

- U tom slučaju javlja se upozorenje (engl. *warning*) koje naznačava da se radi o parametriziranom tipu:

**GenerickiPar is a raw type. References to generic type GenerickiPar<T> should be parameterized**

# Zamjenski simbol (1/2)

- engl. *Wildcard*
- Ako je potrebno napisati metodu koja prima listu brojeva različitih tipova (npr. **`Integer`**, **`Double`**...) čiju sumu je potrebno odrediti, potrebno je nekako ograničiti tipove podataka koje polje može sadržavati
- U tom slučaju potrebno je koristiti zamjenski simbol „?“ pomoću kojeg se definira klasa koja može biti unutar liste:

```
ArrayList< ? extends Number >
```

```
public static double sum(ArrayList<? extends Number> list) {
    double total = 0;
    for (Number element : list) {
        total += element.doubleValue();
    }
    return total;
}
```

# Zamjenski simbol (2/2)

- U metodu „sum“ moguće je predati samo liste koje sadržavaju tipove objekata čije klase su izravno ili neizravno naslijeđene iz klase **`Number`**
- Ukoliko je unutar metode potrebno koristiti tip podatka koji se nalazi u listi, umjesto „?“ potrebno je koristiti „T“:

```
public static <T extends Number> double sum(
                                    ArrayList< T > list )
```

# Konvencija imenovanja parametara za definiranje tipova

- Prema konvenciji imenovanja, imena tipova parametara su velika slova:
  - E – označava element i često se koristi kod parametriziranih zbirki
  - K – označava ključ
  - N – označava broj
  - T – označava tip
  - V – označava vrijednosti
  - S, U... - označavaju preostale tipove

# Primjer definiranja generičke klase "Stog" (1/3)

```
public class Stog<E> {

        private final int VELICINA_STOGA = 10;
        private int vrh = 0;
        E[] elementi;

        @SuppressWarnings("unchecked")
        public Stog() {
          elementi = (E[]) new Object[VELICINA_STOGA];
        }

        public void push(E element) {
           if (vrh == VELICINA_STOGA) {
              throw new RuntimeException("Stog je pun!");
            }
           elementi[vrh++] = element;
        }

        public E pop() {
           if (vrh == - 1) {
              throw new RuntimeException("Stog je prazan");
            }
           return elementi[--vrh];
        }

        public int getStackSize() {
             return vrh;
        }
    }
```

# Primjer definiranja generičke klase "Stog" (2/3)

- Iz navedene klase “Stog” kreiraju se dva različita objekta koji primaju različite parametre: Double i Integer

```
double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9};

Stog<Double> stog1 = new Stog<Double>();
Stog<Integer> stog2 = new Stog<Integer>();

for (Double broj : doubleElements) {
   stog1.push(broj);
}

for (Integer broj : integerElements) {
   stog2.push(broj);
}
```

# Primjer definiranja generičke klase "Stog" (2/3)

```
double[] doubleElements = { 1.1, 2.2, 8.8, 9.9, 10.0};
int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9};

Stog<Double> stog1 = new Stog<Double>();
Stog<Integer> stog2 = new Stog<Integer>();

int kolicina = 0;

for (Double broj : doubleElements) {
     stog1.push(broj);
     ++kolicina;
}

System.out.println("Stavili smo " + kolicina +
                   " elemenata na stog!");

for(int i = stog1.getStackSize(); i > 0; i--) {
    System.out.println(stog1.pop());
}

for (Integer broj : integerElements) {
    stog2.push(broj);
}
```

# Primjer definiranja generičke klase s parametrom tipa (1/6)

- U slučaju da je potrebno definirati generičku klasu koja će primati samo određene vrste parametara, potrebno je ograničiti podskup tipova pomoću ključne riječi **extends**
- Na primjer, neka je zadana klasa "Spremiste" koja može sadržavati objekte podklasa klase "Stvar":

```
public class Spremiste<T extends Stvar> {...}
```

- Objekte klase "Spremiste" moguće je kreirati samo ukoliko se za parametar definira klasa koja nasljeđuje klasu "Stvar"
- Unutar same klase "Spremiste" je u tom slučaju moguće koristiti tip objekta koji je definiran pomoću naziva "T" i na taj način koristiti metode koje su specifične samo za taj određeni tip

# Primjer definiranja generičke klase s parametrom tipa (2/6)

- Neka su zadane klase, npr:

```
public abstract class Stvar {

    public String toString() {
        return "Stvar";
    }

}
public abstract class Odjeca extends Stvar {
    private String boja;

    public String getBoja() {
        return boja;
    }

    public void setBoja(String boja) {
        this.boja = boja;
    }
}
```

# Primjer definiranja generičke klase s parametrom tipa (3/6)

```
public class Hlace extends Odjeca {

    private boolean jeans;

    public String toString() {
        return "Hlače";
    }

    public boolean isJeans() {
        return jeans;
    }

    public void setJeans(boolean jeans) {
      this.jeans = jeans;
    }
}
```

# Primjer definiranja generičke klase s parametrom tipa (4/6)

```
public class Majica extends Odjeca {

    private boolean dugiRukavi;

    public String toString() {
        return "Majica";
    }

    public boolean isDugiRukavi() {
        return dugiRukavi;
    }

    public void setDugiRukavi(boolean dugiRukavi) {
        this.dugiRukavi = dugiRukavi;
    }

}
```

# Primjer definiranja generičke klase s parametrom tipa (5/6)

```
public class Spremiste<T extends Stvar> {
   private List<T> listaOdjece;
   public Spremiste(int kapacitet) {
      listaOdjece = new ArrayList<T>(kapacitet);
   }
   public void dodajOdjecu(T komadOdjece) {
      listaOdjece.add(komadOdjece);
   }
   public void isprazni() {
      listaOdjece.clear();
   }
   public String toString() {
      String opisOdjece = "";
      for (T komadOdjece : listaOdjece) {
         opisOdjece += " " + komadOdjece;
         if (komadOdjece instanceof Hlace) {
            opisOdjece += " jeans : " + ((Hlace)komadOdjece).isJeans();
         }
         else if (komadOdjece instanceof Majica) {
            opisOdjece += " dugi rukavi : "+ ((Majica)komadOdjece).isDugiRukavi();
         }
         opisOdjece += "\n";
      }
      return opisOdjece;
   }
}
```

# Primjer definiranja generičke klase s parametrom tipa (6/6)

- Ako se kreira objekt klase "Spremiste" s parametrom "Odjeca", znači da je time ograničen skup tipova objekata koji mogu biti dodani u "spremište":

```
Spremiste<Odjeca> ormar = new Spremiste<>(2);
Hlace hlace1 = new Hlace();
hlace1.setJeans(true);
Majica majica1 = new Majica();
majica1.setDugiRukavi(true);
ormar.dodajOdjecu(hlace1);
ormar.dodajOdjecu(majica1);
```

- Osim toga moguće je kreirati i objekt klase "Spremiste" koji prima samo objekte podklase "Stvar":

```
Spremiste<Stvar> kutija = new Spremiste<Stvar>(10);
```

# Primjer korištenja zamjenskog simbola

- U slučaju da je potrebno napisati metodu koja će isprazniti svako "spremište", bez obzira koje objekte ono sadržavalo, to je moguće napraviti na sljedeći način:

```
private static void isprazniSpremiste(
                    Spremiste<? extends Stvar> spremiste) {
    spremiste.isprazni();
}
```

- Pomoću znaka "?" omogućeno je metodi predati sve objekte klase "Spremiste" koji u sebi sadrže objekte podklase klase "Stvar"
- Pomoću te metode moguće je "isprazniti" objekte "ormar" i "kutija"
- Ako bi umjesto "? extends Stvar" napisali "? extends Odjeca", objekt "kutija" ne bi bilo moguće "isprazniti"

# Korištenje generičkih metoda

- Moguće je napisati i generičku metodu koja će ispravno funkcionirati samo sa zadanim tipovima, npr:

```
private static <T extends Odjeca> void popuniOdjecom(
                    Spremiste<T> spremiste, List<T> listaOdjece) {
   for (T odjeca : listaOdjece) {
      spremiste.dodajOdjecu(odjeca);
   }
}
```

- Tu metodu moguće je koristiti samo s podklasama klase "Odjeca":

```
Spremiste<Odjeca> ladica = new Spremiste<>(2);
List<Odjeca> odjeca = new ArrayList<Odjeca>();
listaOdjece.add(hlace1);
listaOdjece.add(majica1);
popuniOdjecom(ladica, odjeca);
```