## Апстрактне класе. Универзална суперкласа Object. Сва значења кључне речи final. Интерфејси.

### Апстрактне класе

У примеру са површима, на ком смо учили полиморфизам, полиморфни метод `povrsina()` је у базној класи `Povrs` имао празно тело, из простог разлога што је присуство тог метода у базној класи неопходно, јер то захтева полиморфизам, а не знамо како бисмо га имплементирали јер не постоји универзални начин за рачунање површине површи када не знамо која површ је у питању.

То је чест случај у објектно оријентисаном програмирању: креира се суперкласа из које се изводе потклассе само да би се користиле предности полиморфизма, при чему полиморфни метод нема значење у контексту базне класе, те његово имплементирање у њој нема смисла.

Такав метод могуће је прогласити апстрактним. То се чини навођењем кључне речи `abstract` испред повратног типа метода и тачка-запете (;) уместо тела метода:

```
public abstract double povrsina();
```

`public abstract` ⇔ `abstract public` (битно је да буде испред повратног типа)

Чим класа има бар један апстрактни метод и сама постаје апстрактна, те се и у првом реду њене дефиниције, испред речи `class`, мора навести иста кључна реч, `abstract`.

Није могуће креирати конкретан објекат типа апстрактне класе, али могуће је декларисати променљиву типа апстрактне класе. Та променљива се, даље, може користити на исти начин као у примеру за полиморфизам.

Апстрактни метод не може бити `private`, пошто се `private` метод не може наследити, а тиме ни предефинисати у изведеној класи.

Приликом извођења потклассе из апстрактне базне класе, уколико се не предефинише неки од апстрактних метода базне класе и сама потклассе биће апстрактна, јер поседује апстрактни метод наслеђен од базне класе. У том случају, нећемо моћи да креирамо ни објекте те изведене апстрактне класе.

Ако је класа апстрактна, морамо користити кључну реч `abstract` када је дефинишемо, чак и када она само наслеђује апстрактни метод своје суперкласе. Пре или касније, морамо имати потклассу која има имплементацију свих метода које је наследила од своје (апстрактне) базне класе. Онда можемо креирати објекте те класе.

Једине измене у примеру са површима које је неопходно извршити како би базна класа била апстрактна су:

```
public abstract class Povrs{

      public abstract double povrsina();
      // sve ostalo isto
}
```

### Сва значења кључне речи `final`

Кључна реч `final` има различита значења у зависности од тога на шта је примењена.

`final` променљива (атрибут, параметар метода, локална променљива)
: вредност те променљиве не може да се мења (константна је)

`final` метод
: није га могуће предефинисати у изведеној класи
: Апстрактни метод не може бити дефинисан као `final` јер мора бити дефинисан у некој потклассе

`final` класа
: није могуће извести потклассу из ње
: Апстрактна класа не може бити дефинисана као `final` пошто ће то спречити да апстрактни методи класе икада буду дефинисани
: Декларисање класе као `final` је драстичан корак, који спречава да функционалност класе буде проширена извођењем, па треба да будемо потпуно сигурни да је то оно што желимо.

### `Object` – универзална суперкласа

Уколико се експлицитно не наведе да је класа изведена из неке друге, подразумева се да је изведена из класе `java.lang.Object`, тако да је `Object` директна или индиректна суперкласа сваке друге класе.

Последице:

- променљива типа `Object` може да чува референцу на објекат произвољног класног типа, што је корисно када желимо да пишемо метод који рукује објектима непознатог типа. Можемо да дефинишемо параметар метода типа `Object`, па се референца на објекат произвољног типа може проследити методу.
- наше класе наслеђују чланове класе `Object`. То су методи ( 7 `public` и 2 `protected`):
  - `public: toString(), equals(), hashCode(), getClass(), notify(), notifyAll(), wait()`
  - `protected: clone(), finalize()`

  О методима `toString()` и `getClass()` је већ било речи.

  Методи `getClass()`, `notify()`, `notifyAll()` и `wait()` су дефинисани као `final`, те се не могу предефинисати у изведеним класама.

  Метод `equals()` пореди референцу на текући објекат и референцу прослеђену као аргумент и враћа `true` акко су те две референце једнаке, тј. обе реферишу на исти објекат у меморији. Иначе враћа `false` (сматра се да су два објекта различита чак и када имају исте вредности одговарајућих атрибута)

  Метод `hashCode()` предефинише се у изведеним класама онда када се објекти тих класа користе као кључеви у хеш-табелама. Више речи о томе биће у наставку курса.

  Методи `notify()`, `notifyAll()` и `wait()` служе за синхронизацију нити. Више речи о нитима и њиховој синхронизацији биће у наставку курса.

  Метод `clone()` креира објекат који је копија текућег објекта, без обзира на његов тип.

  Метод `finalize()` позива се приликом уништавања објекта. Може се предефинисати како би се додао сопствени кôд за ослобађање ресурса које је објекат користио.

### Интерфејси

Када је све што желимо скуп једног или више метода које треба имплементирати у различитим класама тако да можемо полиморфно да их зовемо, можемо користити тзв. интерфејс.

Интерфејс је колекција константи и/или апстрактних метода и, у већини случајева, садржи само методе.
За методе се подразумева да су `public` и `abstract`, па се ове кључне речи не наводе. Методи у интерфејсу не могу бити статички.

Интерфејс се дефинише као и класа, али користећи кључну реч `interface` уместо `class`.
Дефиниција интерфејса треба да се налази у фајлу са екстензијом .java и именом које се поклапа са именом интерфејса.
У Eclipse: New > Interface, дамо име интерфејсу, Finish.

Раније су се интерфејси интензивно користили за дефинисање константи, међутим, по увођењу могућности импортовања статичких чланица класе (почев од Јава 5), такав приступ је застарео.

Имплементирање интерфејса

Да бисмо користили интерфејс, „имплементирамо“ га у класи, тј. декларишемо да класа имплементира интерфејс:

```
public class MojaKlasa implements Interfejs1{
      // da klasa ne bi bila apstraktna, mora da implementira sve metode interfejsa
}
```

Сваки метод декларисан у интерфејсу мора имати дефиницију унутар класе ако желимо да креирамо конкретне објекте те класе.

Пошто су методи у интерфејсу подразумевано `public`, неопходно је коришћење кључне речи `public` приликом њиховог дефинисања у класи која имплементира интерфејс.

Делимична имплементација интерфејса

Могуће је изоставити дефиницију једног или већег броја метода интерфејса у класи која га имплементира, али у том случају, класа наслеђује те апстрактне методе од интерфејса, па мора и сама бити декларисана као апстрактна и неће бити могуће креирање конкретних објеката типа те, апстрактне, класе. Да би се дошло до корисне класе, мора се дефинисати потклаcа која имплементира преостале методе интерфејса.

Имплементирање већег броја интерфејса

Класа може имплементирати више од једног интерфејса. У том случају имена свих интерфејса које класа имплементира раздвајају се запетама:

```
public class MojaKlasa implements Interfejs1, Interfejs2, Interfejs3 {...}
```

Наслеђивање (извођење једног интерфејса из другог)

Могуће је извести један интерфејс из другог, коришћењем кључне речи `extends` за идентификовање имена базног интерфејса. То је истог облика као када изводимо једну класу из друге. Изведени интерфејс добија све методе (и константе) интерфејса из кога се изводи:

```
public interface MojInterfejs extends TvojInterfejs{...}
```

Интерфејси и вишеструко наслеђивање

За разлику од класе, која може да наследи само једну класу, интерфејс може да наследи произвољан број интерфејса. Имена свих тих интерфејса раздвајају се запетама:

```
public interface MojInterfejs extends TvojInterfejs, NasInterfejs{...}
```

Ово се назива вишеструким наслеђивањем.
У Јави, класе не подржавају, а интерфејси подржавају вишеструко наслеђивање.

Ако два или више супер-интерфејса декларишу метод са истим потписом, тај метод мора имати исти повратни тип у свим тим интерфејсима. Само тако могуће је да једна имплементација метода у класи задовољи све интерфејсе.

Коришћење интерфејса

Интерфејс декларише стандардни скуп операција. Објекти разних класа које имплементирају исти интерфејс имају заједнички скуп операција. Наравно, дата операција у једној класи може бити имплементирана посве различито од начина на који је имплементирана у другој класи, али начин на који се операција позива је исти за објекте свих класа које имплементирају интерфејс.

Најважнија употреба интерфејса (полиморфизам):
пошто интерфејс дефинише тип, могуће је користити полиморфизам за скуп класа које имплементирају исти интерфејс.

**интерфејси и полиморфизам**

Могуће је декларисати променљиву типа интерфејса, а у њој се може чувати референца на објекат произвољне класе која имплементира тај интерфејс.

- у интерфејсу се наведу декларације полиморфних метода
- све класе за које хоћемо да користе полиморфизам напишемо тако да имплементирају тај интерфејс
- декларишемо променљиву типа интерфејса у којој чувамо референце на објекте типа тих класа
- и полиморфизам функционише

Коришћење више интерфејса

Променљива типа неког интерфејса може се користити само за позив оних метода који су декларисани у том интерфејсу. Ако класа објекта чија референца се чува у променљивој имплементира и неки други интерфејс, онда би за позив метода декларисаних у том интерфејсу требало или користити променљиву типа тог интерфејса за чување референце на објекат, или кастовати референцу на објекат у тип његове стварне класе.

Нпр. уколико имамо случај:

```
public interface Obim{
      double obim();
}

public interface Povrsina{
      double povrsina();
}

public class Krug implements Obim, Povrsina{
      …
      public Krug(){
            …
      }

      public double obim(){
            …
      }
      public double povrsina(){
            …
      }
}

public class Test{
      public static void main(String[] args){
            …
            Povrsina p = new Krug();
            double pov = p.povrsina();
            …
            if(p instanceof Obim){   // kastovanje u tip interfejsa
                  double ob  = ((Obim)p).obim(); // polimorfni poziv metoda obim()
                  …
            }
            ...
            if(p.getClass() == Krug.class){
                  Krug k = (Krug)p; // kastovanje u stvarni tip objekta
                  double ob = k.obim(); // nije polimorfni poziv
            }
      }
}
```

Након кастовања у стварни тип, могуће је позивати произвољан метод класе објекта, али није могуће добити полиморфно понашање. Компајлер одређује који метод се позива у време компајлирања.

Да бисмо полиморфно позивали методе из другог интерфејса, неопходно је да имамо референцу сачувану као тај тип.

Чак и ако интерфејси нису ни у каквој вези, кастовање је могуће јер реферисани објекат је типа класе која имплементира оба интерфејса.

Параметри метода типа интерфејса

Параметар метода може бити типа интерфејса.
У том случају, приликом позива метода, као аргумент се може проследити референца на објекат произвољне класе која имплементира тај интерфејс.

Ова техника дефинисања параметара интерфејсног типа се интензивно користи унутар Јавине библиотеке.
Нпр. `CharSequence` је интерфејс, а имплементирају га класе `String` и `StringBuffer`.
Постоји неколико њихових метода који имају параметре типа овог интерфејса:

класа `String`

```
boolean contains(CharSequence s)
boolean contentEquals(CharSequence cs)
String replace(CharSequence target, CharSequence replacement)
```

класа `StringBuffer`

```
StringBuffer append(CharSequence s)
StringBuffer append(CharSequence s, int start, int end)
StringBuffer insert(int dstOffset, CharSequence s)
StringBuffer insert(int dstOffset, CharSequence s, int start, int end)
```