



Programazioa

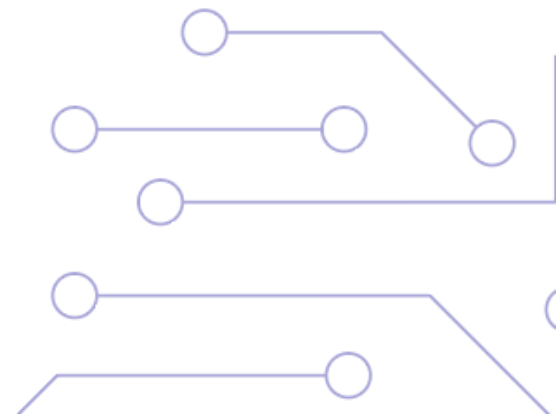
OBP Aurreratua

OBParen lau printzipioak: kapsulatzea

- Programa batean elkarreragiten duten objektuen artean ikuspena edo erabilgarritasuna mugatzea edo kontrolatzea da
 - Objektu batek beste objektu baten atributuak eta metodoak erabili ahal izango ditu bakarrik azken honetan adierazten den moduan
 - Atzipen baimenak minimora mantendu behar dira

Katua
- energia
- umorea
- loEgin()
- elikatu()
- miaukaEgin()

elikatu()
- energia++
- umorea++
- miaukaEgin()





OBParen lau printzipioak: abstrakzioa

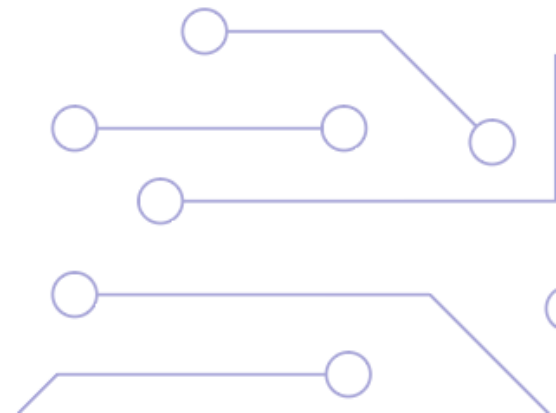
- Kapsulatzearen hedapen bat bezala uler daiteke
- Programa bat oso handia izan daiteke eta objektu askoren arteko elkarreraginak izan ditzake
 - Aldaketak egitea eta kodea mantentzea nekeza bihurtzen da
- Abstrakzioa aplikatzeak objektu bakoitzak maila altuko mekanismo bat baino ez duela azaldu behar erabiltzeko esan nahi du.
 - Mekanismo horrek barne-inplementazioaren xehetasunak ezkutatu beharko lituzke.
 - Beste objektuetarako garrantzitsuak diren eragiketak baino ez ditu erakutsi behar.

OBParen lau printzipioak: herentzia

- Batzuetan objektuak beraien artean oso antzekoak izan daitezke, bakoitzaren logika klase batean isolatuko dugu?

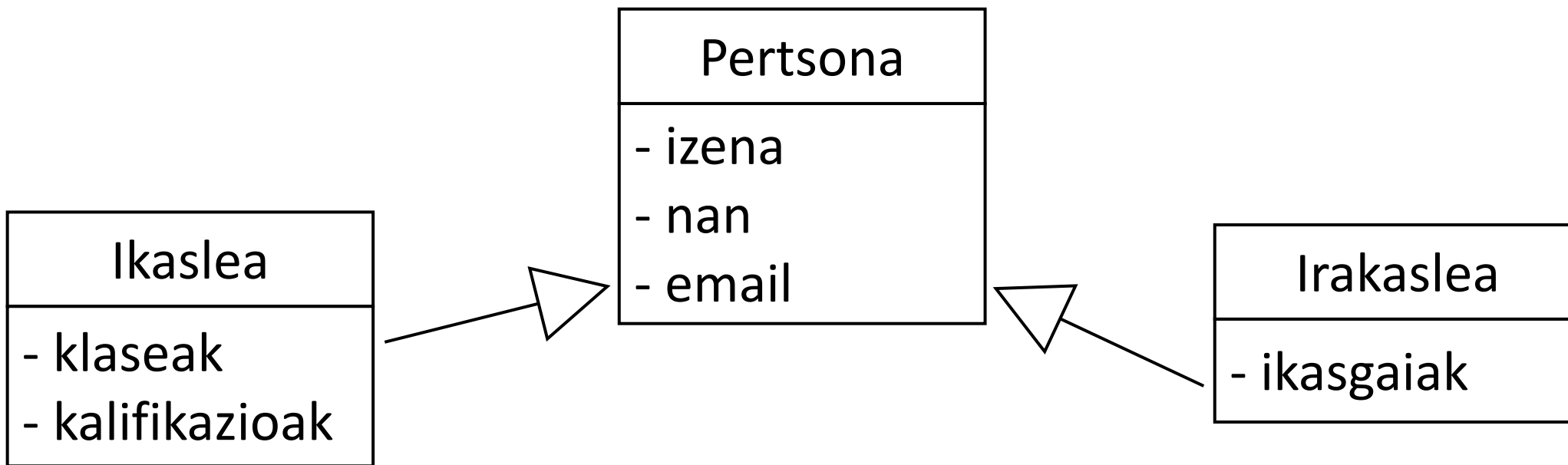
Ikaslea
<ul style="list-style-type: none">- izena- nan- email- klaseak- kalifikazioak

Irakaslea
<ul style="list-style-type: none">- izena- nan- email- ikasgaiak



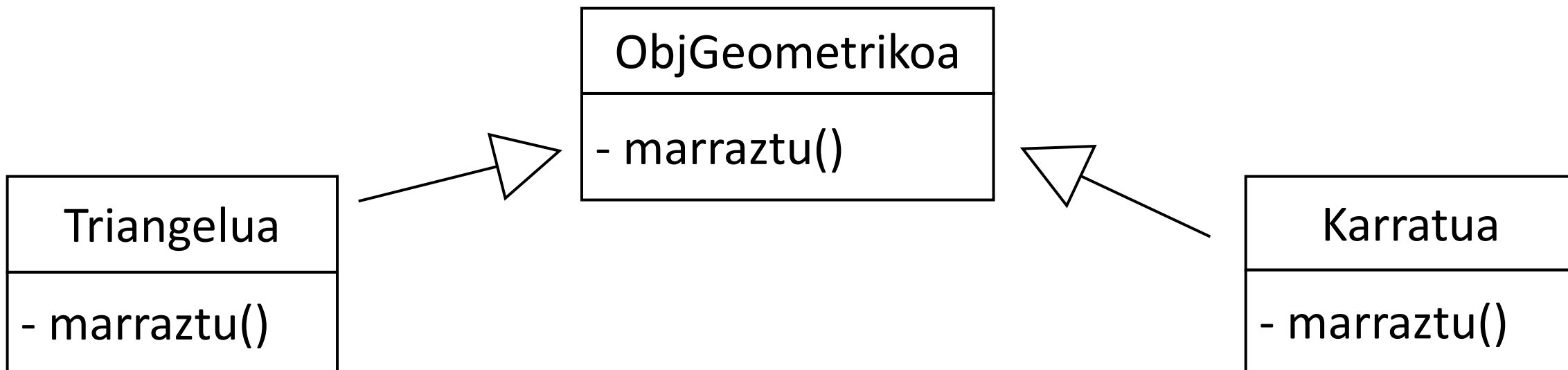
OBParen lau printzipioak: herentzia

- Amankomunak diren atributu eta metodoak klase batean definitu daitezke eta ondoren beste klaseek espezializatu daitezke
 - Atributu eta metodo berriak definitu ditzakete
 - Guraso klaseko metodoak berriatzi ditzakete



OBParen lau printzipioak: polimorfismoa

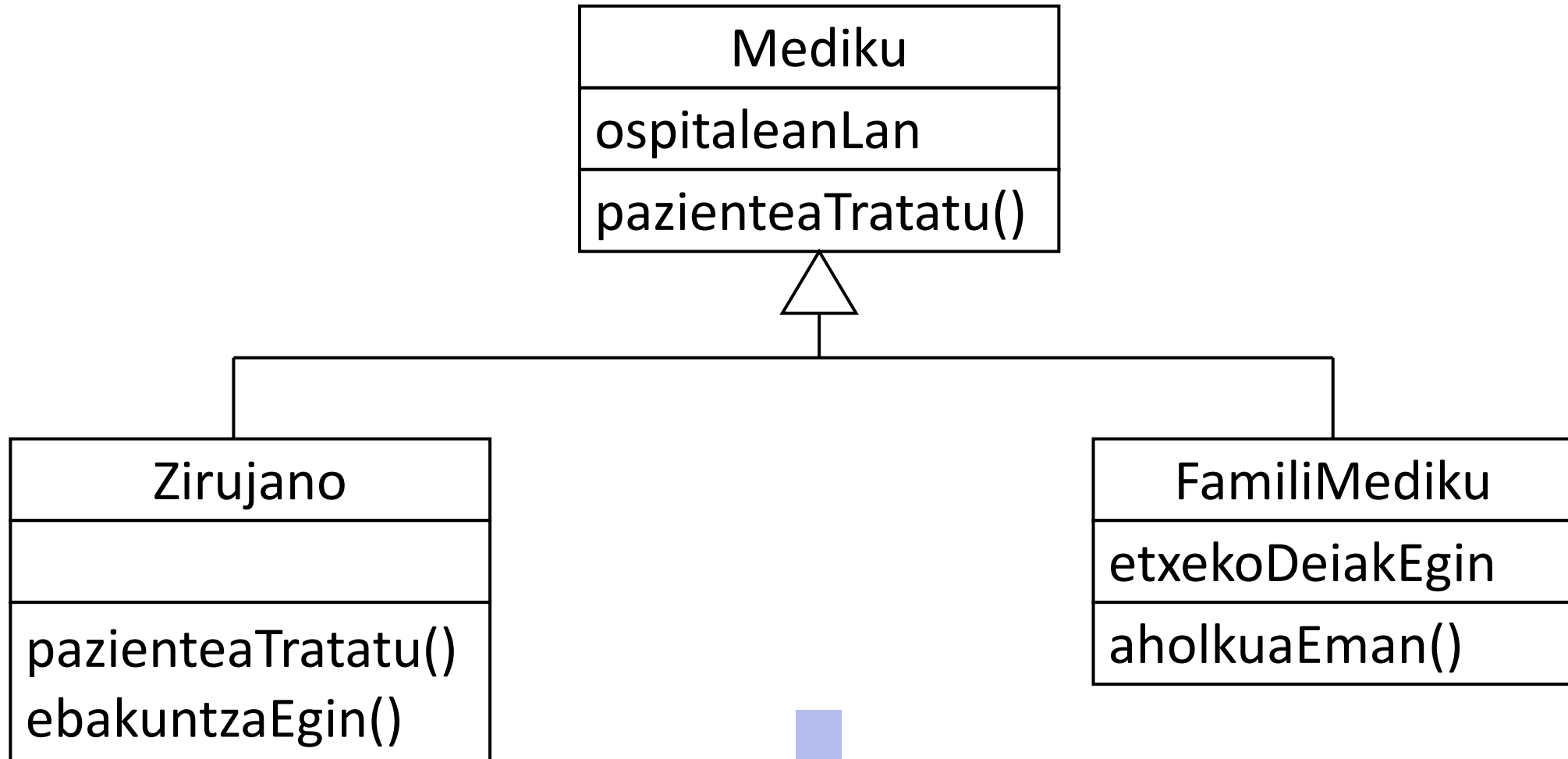
- Metodo baten deia deitu den hierarkiaren klasearen independentea da
 - Berdin zaigu guraso edo ondorengoen klase batean deitzea
 - Izen berdineko metodoek hierarkia bateko klaseetan modu desberdinetan funtzionatu dezakete



Herentzia

- Klase batean (**superklasea**) zehatzagoak diren klaseetako (**azpiklaseak**) ezaugarri amankomunak definitzeko erabiltzen da
- Azpiklase bakoitzak superklasea hedatzen (***extends***) du, honela:
 - Superklasearen atributu eta metodoak heredatzen ditu
 - Heredatutakoez gain bere atributu eta metodoak izan ditzake
 - Superklasetik heredatutako metodoak berridatzi ditzake, hurrengoa betetzen bada:
 - Metodo izen eta parametro berdinak izan behar ditu
 - Itzulera balioaren mota (baldin badu) mota berdinekoa edo mota horren azpiklasea izan behar du
 - Ezin du ikuspen (edo atzipen) maila txikiagoa izan (honen inguruan gehiago aurrerago)

Herentzia



Herentzia

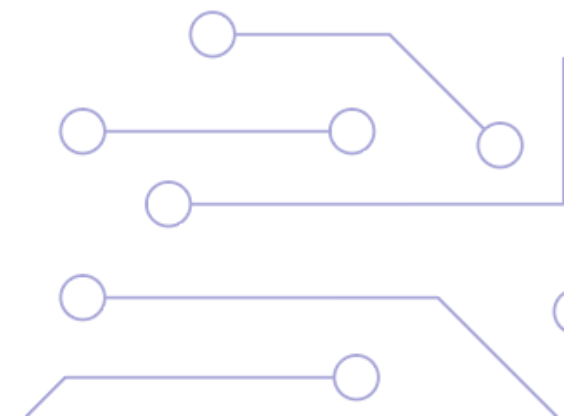
```
public class Mediku{
    boolean ospitaleanLan;
    public void pazienteaTratatu() {}
}
public class FamiliMediku extends Mediku{
    boolean etxekoDeiakEgin;
    public void aholkuaEman() {}
}
public class Zirujano extends Mediku{
    public void pazienteaTratatu() {}
    public void ebakuntzaEgin() {}
}
```

Atributu eta metodo
bat gehitzen ditu

Metodo bat gehitzen du
eta bestea berridazten du

Metodoen gainkarga


- Parametro desberdineko eta izen berdineko hainbat metodo izateari deitzen zaio
 - Erabiltzaileak behar duen arabera metodo bat edo beste erabiliko da
- Ezaugarri batzuk ditu:
 - Parametro desberdinak izan behar dituzte
 - Itzulera balioaren mota aldatu daiteke, baina ezin da izan aldatzen den gauza bakarra kasu horretan
 - Ikuspena aldatu daiteke





Metodoen gainkarga


```
public class gainkarga1{  
  
    public int batu(int a, int b){  
        return a + b;  
    }  
  
    public double batu(double a, double b){  
        return a + b;  
    }  
}
```





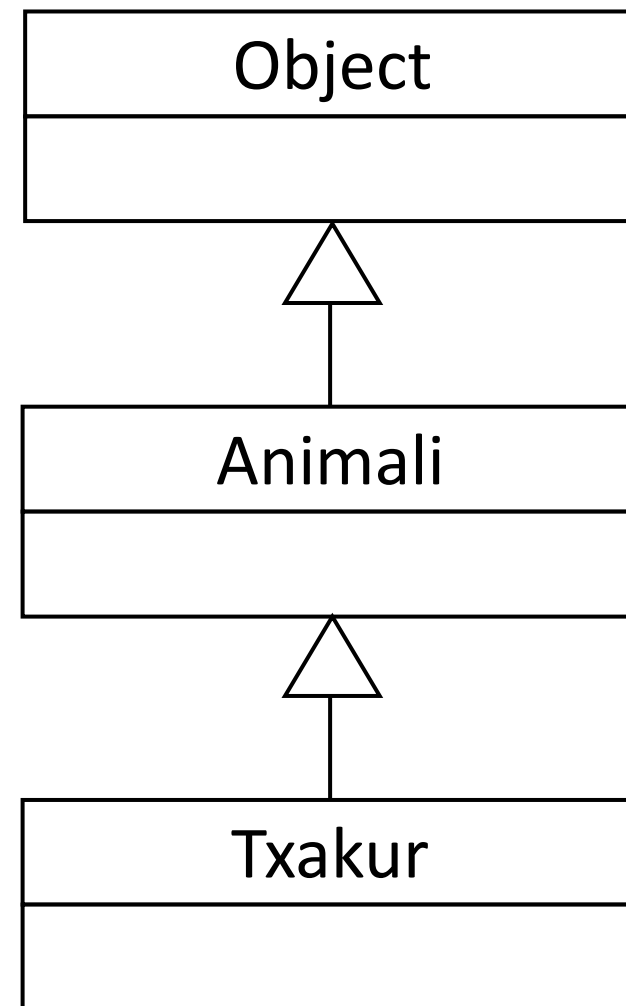
Metodoen gainkarga

```
public class gainkarga2{  
    String id;  
    public void ezarriId(String nireId) {  
        this.id = nireId;  
    }  
    public void ezarriId(int zenb) {  
        this.id = String.valueOf(zenb);  
    }  
}
```



Herentzia eta eraikitzaileak

- Klase bakoitzak eraikitzaile bat du
- Objektu baten instantzia sortzen denean (*new* hitza erabilita) hierarkia osoko klaseen eraikitzaileei deitzen zaie






Herentzia eta eraikitzaileak

```
public class Animali{
    public Animali(){
        System.out.println("Animali baten sorrera");
    }
}

public class Txakur extends Animali{
    public Txakur(){
        System.out.println("Txakur baten sorrera");
    }
}

public class ProbaTxakur{
    public static void main(String[] args){
        Txakur tx = new Txakur();
    }
}
```



Herentzia eta eraikitzaileak

- Zuzenean superklase baten eraikitzailea erabili nahi bada, **super** hitz erreserbatua erabiltzen da, ez superklasearen eraikitzailearen izena
 - *super* metodoaren deia eraikitzailearen lehen agindua izan behar da, eta jartzen ez bada konpiladoreak gehitzen du

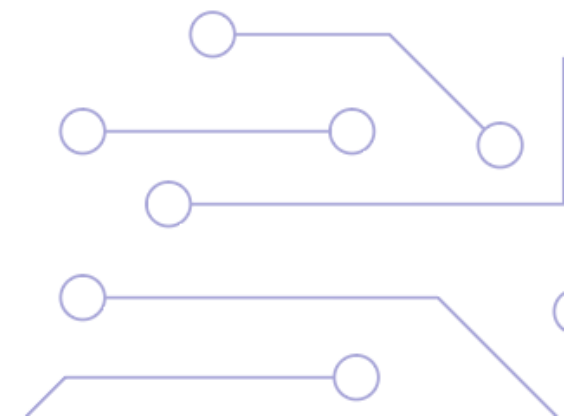
```
public class Ahate extends Animali{  
    int tamaina;  
    public Ahate(int tam){  
        Animali();  
        tamaina = tam;  
    }  
}
```

```
public class Ahate extends Animali{  
    int tamaina;  
    public Ahate(int tam){  
        super();  
        tamaina = tam;  
    }  
}
```

Herentzia eta eraikitzaileak

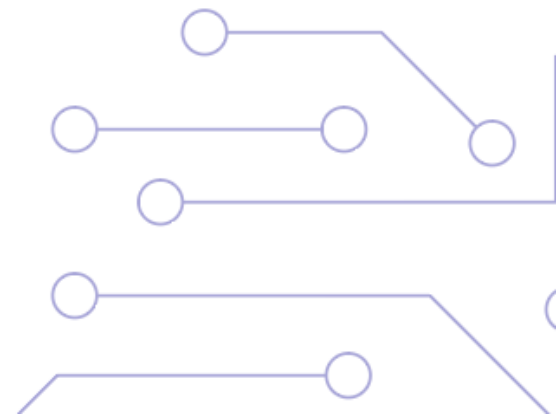
- Superklasearen eraikitzaileak parametroak baditu, hauek *super* deian pasatu behar zaizkio

```
public class Animali{  
    String izen;  
    public Animali(String iz){  
        this.izen = iz;  
    }  
}  
  
public class Txakur extends Animali{  
    public Txakur(String iz){  
        super(iz);  
    }  
}
```



Herentzia eta eraikitzaileak

- Klase batean parametro desberdineko eraikitzaile bat baino gehiago badago (gainkarga) **this** metodoa erabili daiteke kodea berrrerabiltzeko
 - Dei hau eraikitzaileko lehena izan behar da, eta erabiltzen bada, ezingo da *super* deia erabili
 - Metodo honek *super* metodoaren antzera parametroak jaso ditzake



Herentzia eta eraikitzaileak

```
public class Mini extends Kotxe{
```

```
    String kolore;
```

```
    public Mini() {
```

```
        this("gorria");
```

← Hemen ez dago super() deiarik

```
    }
```

```
    public Mini(String kol) {
```

```
        this.kolore = kol;
```

← Hemen super() deia inplizituki egiten da

```
    }
```

```
}
```

Superklaseen metodoen berridazketa

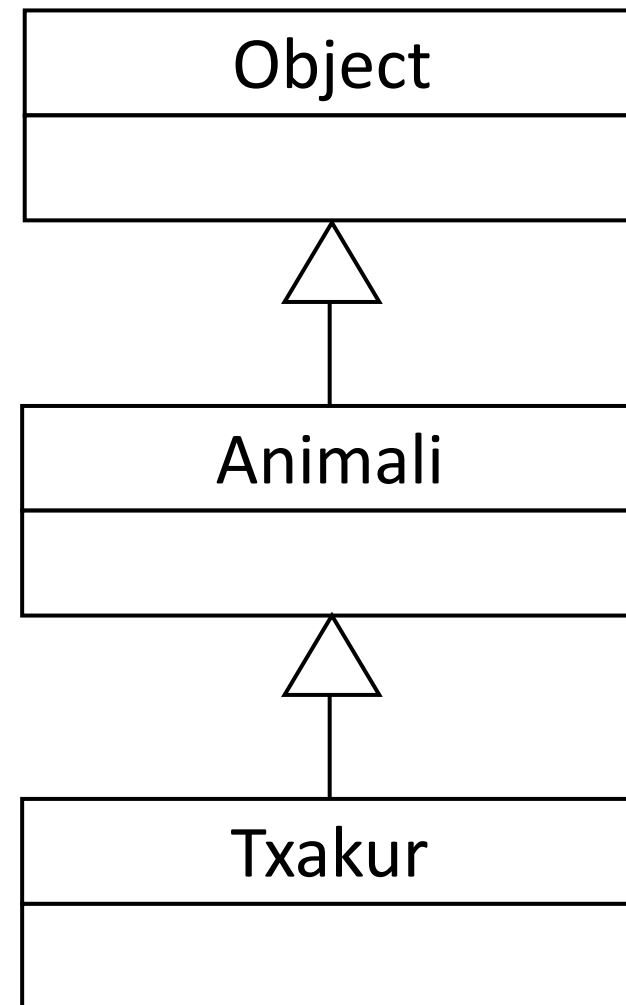
- Azpiklase batean superklase baten metodo bat erabili eta egiten duenaz gain funtzionalitate berri bat gehitu nahi bada (guztiz berridatzi nahi ez bada) *super* metodoa erabili daiteke

```
public class Txosten{  
    public prestatuTxostena(){  
        // Txostena prestatzen du  
    }  
    public inprimatuTxostena(){  
        // Txostena inprimatzen du  
    }  
}
```

```
public class TxostenZient extends Txosten{  
    public prestatuTxostena(){  
        super.prestatuTxostena();  
        konprobatuDatuk();  
    }  
    public konprobatuDatuk(){  
        // Datuk ondo dauden konprobatzen du  
    }  
}
```

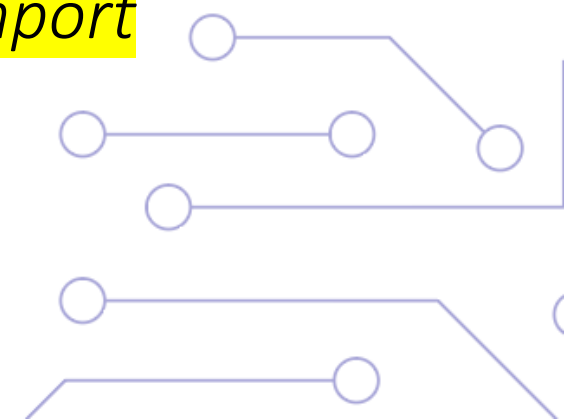
Object klasea

- Klase guztien superklasea
- Esplizituki beste klase bat hedatzen ez duen klase batek inplizituki *Object* klasea hedatuko du
- *Object* klasearen metodo batzuk:
 - `boolean equals()`: bi objektu berdinak diren adierazten du
 - `Class getClass()`: deitzen den objektuaren klasea itzultzen du
 - `int hashCode()`: objektua errepresentatzen duen hash kodea (identifikatzaile bat) itzultzen du
 - `String toString()`: klasearen izena eta zenbaki bat duen karaktere-kate bat itzultzen du
- Metodo hauek norberaren beharren arabera berridatzi daitezke



Java paketeak eta import agindua

- Beraien artean erlazionatutako klaseak paketeetan antolatzen dira
- Paketeak fitxategi karpetak bezala uler daitezke
- Bi pakete mota desberdintzen dira:
 - Pakete integratuak: JDKn dauden aurreprestatutako liburutegiak
 - Erabiltzailearen paketeak
- Klase bat pakete berdinean dagoen beste klase bat erabili behar badu ez da ezer egin behar, baina beste pakete batean badago **import** agindua erabili behar da



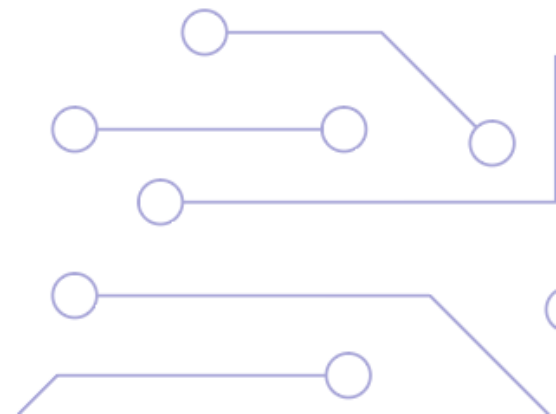
Java paketeak eta import agindua

- Adibidez, Scanner klasea erabili ahal izateko honakoa idazten da:

```
import java.util.Scanner;
```

- Kasu honetan java.util pakete bat da eta Scanner pakete horretako klase bat
- Klase bat, interfaze bat edo pakete oso batean dauden elementuak inportatu daitezke

```
import java.util.Scanner;  
import java.util.*;  
import java.util.List;
```

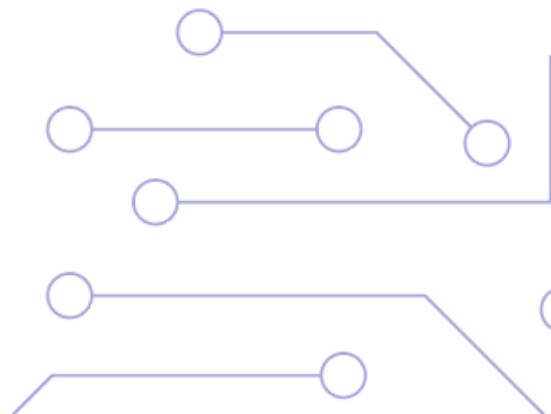
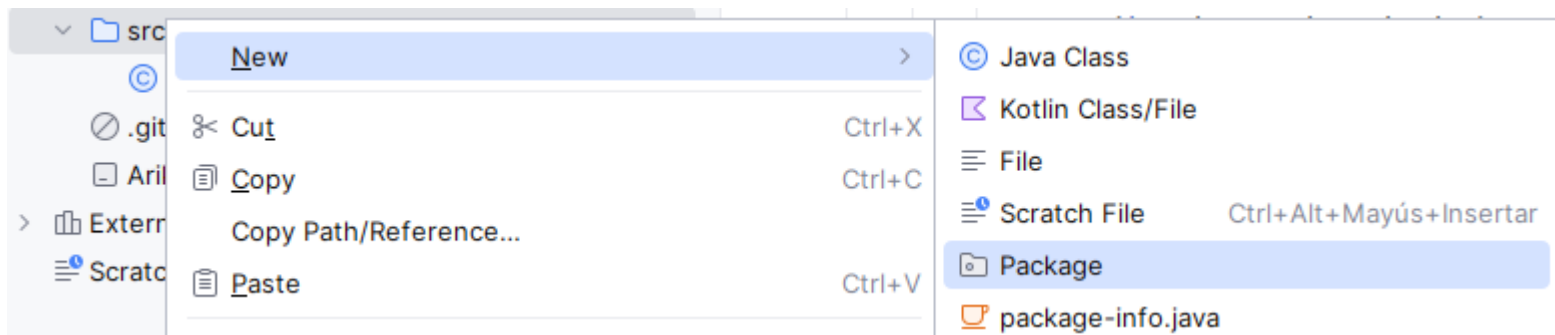


Java paketeak eta import agindua

- Erabiltzaileak sortzen dituen paketeak *package* aginduarekin definitzen dira:

```
package zubiri.java.liburutegi;
```

- Garapen inguruneak ere paketeak sortzeko aukera ematen du



Ikuspen maila eta atzipen aldagailuak

- Orain arte klase guztiak *public* bezala definitu dira
 - *public* hitza atzipen aldagailu bat da, beste klaseek hau ikus dezaketen adierazten du
- Klase mailako atzipen aldagailuak:

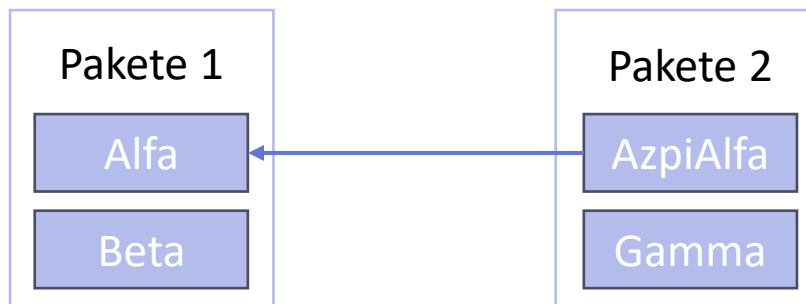
Aldagailua	Deskribapena
public	Edozein klasetik atzitu daiteke
lehenetsia (aldagailu gabe)	Pakete berdineko klaseetatik bakarrik atzitu daiteke

Ikuspen maila eta atzipen aldagailuak

- Klase barneko elementuek (atributu eta metodoek) lau atzipen aldagailu desberdin dituzte:

Aldagailua	Deskribapena
public	Elementua beste edozein kasetik atzitu daiteke
protected	Elementua pakete berdineko klase eta klasearen azpiklaseetatik atzitu daiteke
lehenetsia (aldagailu gabe)	Elementua pakete berdineko klaseetatik atzitu daiteke
private	Elementua klase berdinetik atzitu daiteke

Ikuspen maila eta atzipen aldagailuak



Aldagailua	Alfa	Beta	AzpiAlfa	Gamma
public	Bai	Bai	Bai	Bai
protected	Bai	Bai	Bai	Ez
lehenetsia (aldagailu gabe)	Bai	Bai	Ez	Ez
private	Bai	Ez	Ez	Ez

Ikuspen maila eta atzipen aldagailuak

```
class A{
    private datu = 40;
    private void mezua(){System.out.println("Kaixo");}
}
public class Proba{
    public static void main(String args[]){
        A obj = new A();
        System.out.println(obj.datu); // Konpilazio errorea
        obj.mezua(); // Konpilazio errorea
    }
}
```

Ikuspen maila eta atzipen aldagailuak

```
package pakete;  
class A{  
    void mezua(){System.out.println("Kaixo");}  
}  
package pakete2;  
class B{  
    public static void main(String args[]){  
        A obj = new A(); // Konpilazio errorea  
        obj.mezua(); // Konpilazio errorea  
    }  
}
```

Ikuspen maila eta atzipen aldagailuak

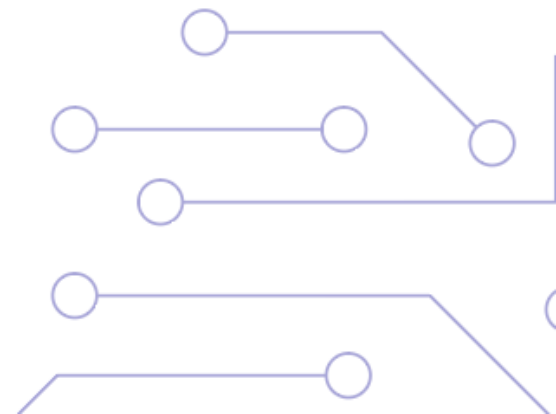
```
package pakete;  
public class A{  
    protected void mezu(){System.out.println("Kaixo");}  
}  
  
package pakete2;  
class B extends A{  
    public static void main(String args[]){  
        B obj = new B();  
        obj.mezu();  
    }  
}
```

Ikuspen maila eta atzipen aldagailuak

```
package pakete;  
public class A{  
    public void msg(){System.out.println("Hello");}  
}  
package pakete2;  
class B{  
    public static void main(String args[]){  
        A obj = new A();  
        obj.msg();  
    }  
}
```

Ikuspen maila eta atzipen aldagailuak

- Segurtasunagatik eta informazioa sentikorra babesteko ikuspen maila ahalik eta murriztatzaileena izan behar du (kapsulatzea)
 - Klase barneko elementuak *private* bezala definitu hori ez egiteko oso arrazoi on bat izan ezean
- *public* atzipen aldagailua ekidin ahal den heinean, atributu konstanteetan izan ezik






Getter eta setter

- Atributuak normalean private bezala definituko direnez, beste klaseetatik ezingo dira atzitu
- Balio hauek atzitzeko metodoak eskaini daitezke, *getter* eta *setter* metodoak
 - Segurtasuna hobe da
 - Atzipen maila (irakurketa/idazketa) mugatu daiteke
- Metodo hauek bakarrik atributuaren balioa itzuli edo idazten dute



Getter eta setter

```
public class Pertsona{  
    private String izena;  
    // Getter  
    public String getIzena() {  
        return izena;  
    }  
    // Setter  
    public void setIzena(String izenBerri) {  
        this.izena = izenBerri;  
    }  
}
```

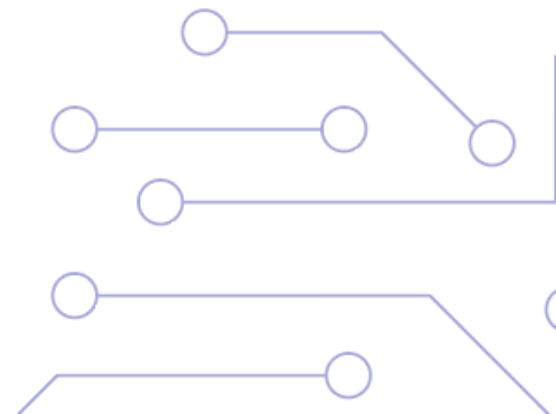


Getter eta setter

```
public class Main {  
    public static void main(String[] args){  
        Pertsona p = new Pertsona();  
        p.izena = "Unai"; // Konpilazio errorea  
        System.out.println(p.izena); // Konpilazio errorea  
    }  
}  
  
public class Main {  
    public static void main(String[] args){  
        Pertsona p = new Pertsona();  
        p.setIzena("Unai");  
        System.out.println(p.getIzena());  
    }  
}
```

Klase abstraktuak

- Batzuetan ez du zentzurik klase batetik objektuak instantziatzea
 - *Animali* klase baten azpiklase diren *Katu* eta *Txakur* klaseetatik agian bai, baina *Animali* klaseko objektuak?
- Superklasea izatea interesgarri izan daiteke herentzia eta polimorfismoa erabiltzeko, baina superklase horren instantziak egin ahal izatea mugatu behar da
- Klasea abstraktu bezala definitzen da
 - Ezingo da instantziarik egin
 - Azpiklaseek hedatu ahal izango dute



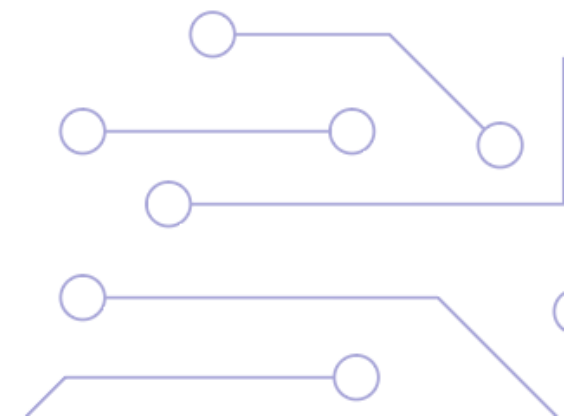
Klase abstraktuak

```
public abstract class Animali {  
    int hankaKop;  
    public void ibili(){...}  
    ...  
}  
public class Katu extends Animali {  
    public Katu(){...}  
    ...  
}  
public class Txakur extends Animali {  
    public Txakur(){...}  
    ...  
}
```

Metodo abstraktuak

- Klase abstraktuetan metodo abstraktuak sor daitezke
- Metodo hauek klase abstraktuak hedatzen dituzten azpiklaseetan **berridatzi behar dira**
 - Azpiklase guztientzat egokia den inplementaziorik aurkitzen ez den kasuetarako
- Metodo abstraktuak bakarrik klase abstraktuetan egon daitezke
 - Klase abstraktu batean abstraktuak ez diren metodoak ere egon daitezke
 - Abstraktua ez diren klaseetan ezin dira metodo abstraktuak egon
- Metodo abstraktu batek ez du gorputzik

```
public abstract void ibili();
```





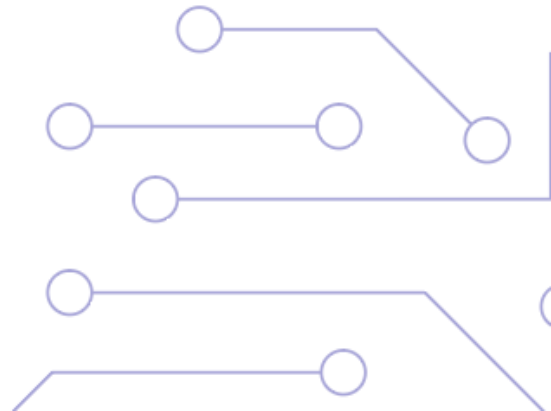
Metodo abstraktuak
bakarrik klase
abstraktuetan definitu
daitezke



Klase abstraktuetan
abstraktuak ez diren
metodoak definitu
daitezke

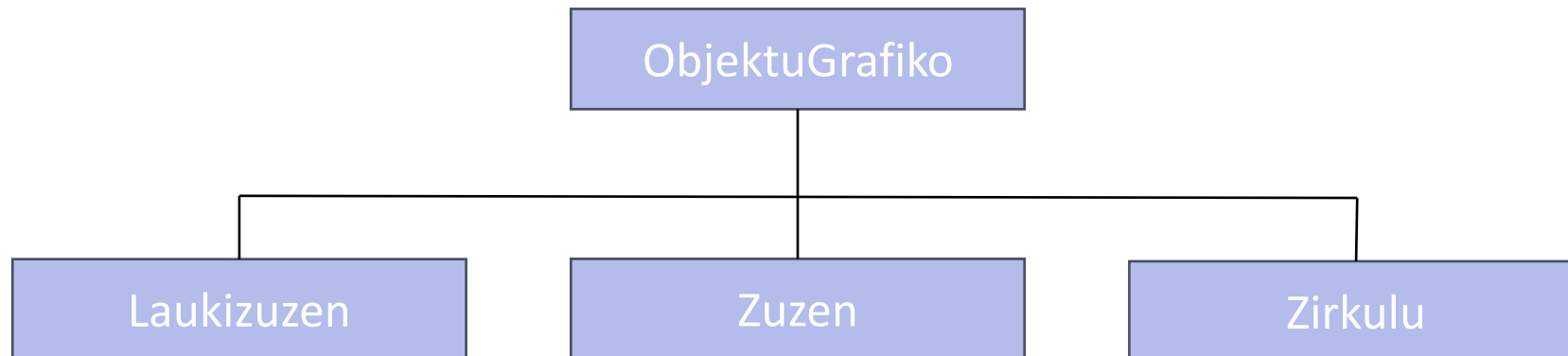
Metodo abstraktuak

- Klase abstraktu bat hedatzen duten azpiklase guztiek klase abstraktuaren metodo abstraktu guztiak berridatzi behar dituzte
 - Bestela, azpiklase hau abstraktu bezala definitu beharko da eta beste klase batek hedatu beharko du azpiklase berri hauek metodo abstraktu hori berridatzi dezaten



Metodo abstraktuak

- Suposa dezagun ondorengo klase egitura:
 - *Laukizuzen*, *Zuzen* eta *Zirkulu* azpiklaseek amankomunak diren zenbait ezaugarri (*posizioa*, *orientazioa*, *kolorea*) eta ekintza (*mugitu*, *rotatu*, *marraztu*) dituzte
 - Hauetako batzuk objektu guztientzat berdinak dira, hala nola, *posizioa*, *kolorea* eta *mugitu*
 - Beste batzuk, aldiz, inplementazio desberdina behar dute, adibidez, *marraztu* ekintza



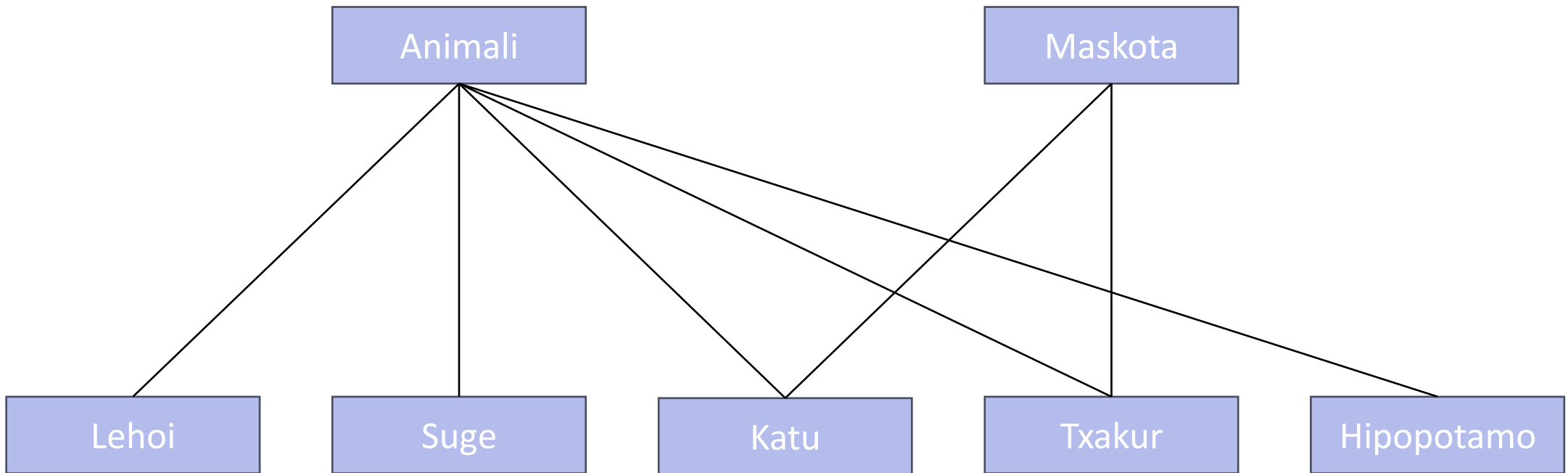
Metodo abstraktuak

```
public abstract class ObjektuGrafiko {  
    int x, y;  
    ...  
    public void mugitu(int xBerri, yBerri){  
        ...  
    }  
    public abstract void marraztu();  
}
```

```
public class Zuzen extends ObjektuGrafiko{  
    public void marraztu(){...}  
    ...  
}  
  
public class Zirkulu extends ObjektuGrafiko{  
    public void marraztu(){...}  
    ...  
}
```

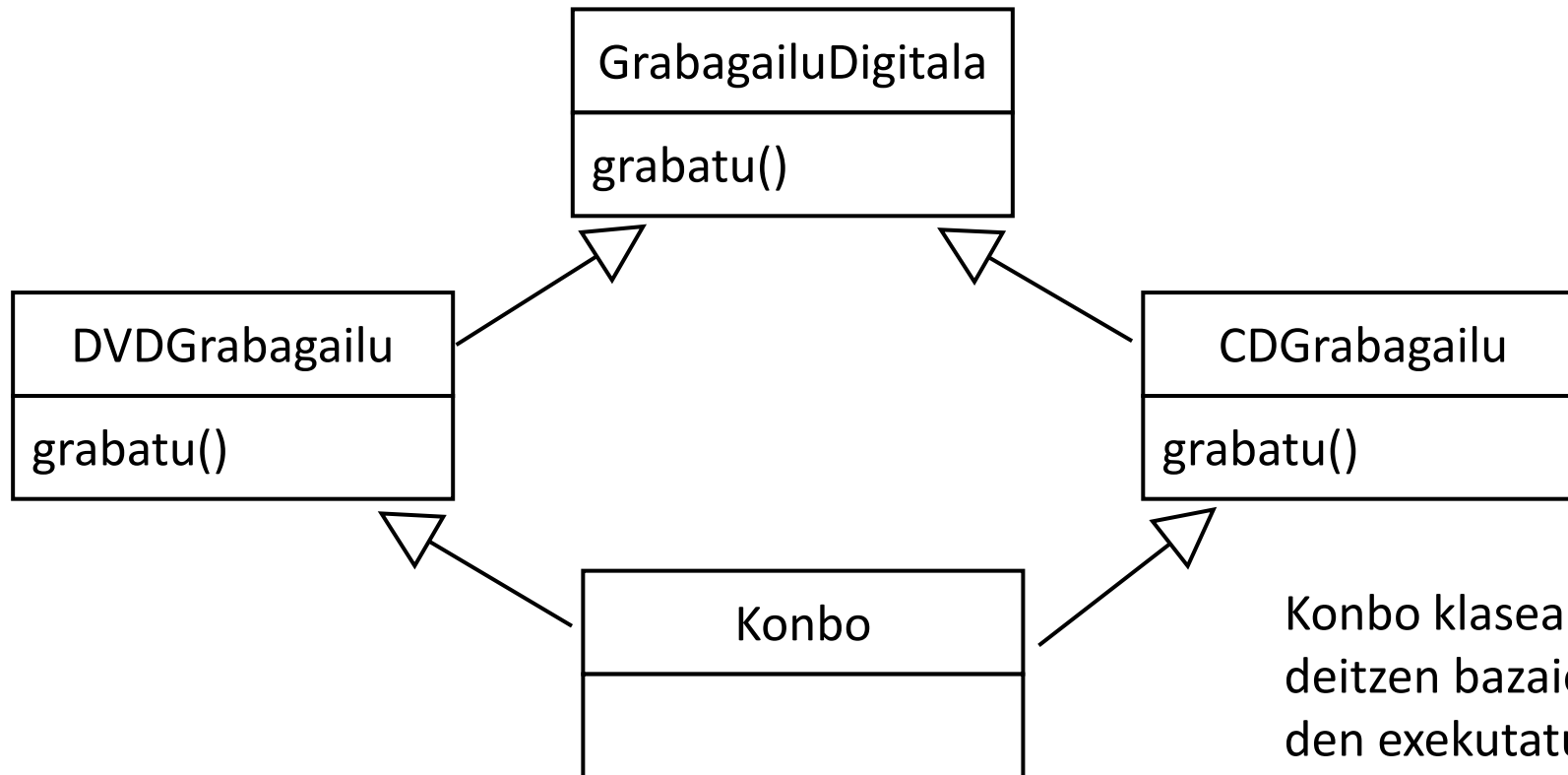
Interfazeak

- Batzuetan klase batzuk bi superklase hedatu behar dituzte
 - *Katu* eta *Txakur* klaseek herentzia bat baino gehiago behar dute



Interfazeak

- Honek beste arazo bat sor dezake: heriotzaren erronboa



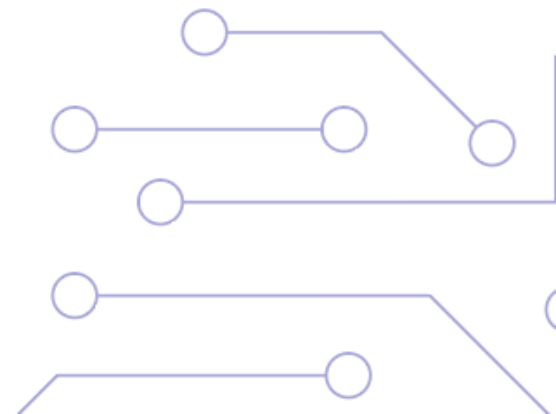
Konbo klasean grabatu metodoari deitzen bazaio ezin da jakin zein den exekutatu behar den metodoa



Javan herentzia
anizkoitza debekatuta
dago

Interfazeak

- Herentzia anizkoitza interfazeen erabilerarekin lor daiteke
- Interfazeak klase guztiz abstraktuak dira
 - Metodo guztiak abstraktuak dira, ez dute gorputzik
 - Ezin da instantziatu, ez du eraikitzailerik
- Klaseek interfazeak inplementatzen dituzte (*implements*)
 - Interfazeen metodo guztiak berridatzi behar dituzte



Interfazeak

```
public abstract class Animali {  
    int hankaKop;  
    public void ibili(){...}  
    public abstract void soinuaEgin();  
    ...  
}
```

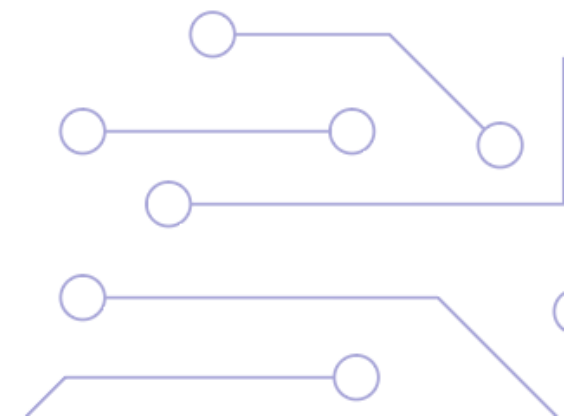
```
public class Katu extends Animali implements Maskota {  
    public Katu(){...}  
    public void soinuaEgin(){...}  
    public void jolastu(){...}  
    public void mimoakjaso(){...}  
    ...  
}
```

```
public interface Maskota {  
    public abstract void jolastu();  
    public abstract void mimoakJaso();  
    ...  
}
```

Interfazeak

- Klase batek interfaze bat baino gehiago ere implementa dezake
 - Herentzia anizkoitza lortzen da
 - Gogoratu: superklase bakarra hedatu dezake

```
public class txakur extends Animali implements Maskota, Salbagarri{  
    ...  
}
```



Polimorfismoa

- Orain arte ikusitako adibideetan objektu bat sortzen denean sortzen den klasearen motako aldagai batean gorde izan da

```
Txakur nireTxakur = new Txakur();
```

- Polimorfismoa erabilita helburu aldagaia sortzen den objektuaren klasearen edozein arbaso klasearen motakoa izan daiteke

```
Animali nireTxakur = new Txakur();
```

- Polimorfismoaren erabilerarekin azpiklase berriak sortzen diren kasurako egon daitezkeen arazoak ekiditen dira

Polimorfismoa

```
public class Animali {  
    public void soinuaEgin(){  
        System.out.println("Soinua egiten dut");  
    }  
}  
  
public class Katu extends Animali {  
    public Katu(){...}  
    public void soinuaEgin(){  
        System.out.println("Miau, miau");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args){  
        Animali an = new Katu();  
        an.soinuaEgin();  
    }  
}
```

```
"C:\Program Files\Java\jdk-17\bin\ja'  
Miau, miau
```

```
Process finished with exit code 0
```



Polimorfismoa erabiltzean
bakarrik helburu
aldagaiaren motako
klaseko metodoak erabili
ahal izango dira

Polimorfismoa

```
public class Animali {  
    public void soinuaEgin(){  
        System.out.println("Soinua egiten dut");  
    }  
}  
  
public class Katu extends Animali {  
    public Katu(){...}  
    public void soinuaEgin(){  
        System.out.println("Miau, miau");  
    }  
    public void jolastu(){  
        System.out.println("Jolasten ari naiz");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args){  
        Animali an = new Katu();  
        an.soinuaEgin();  
        an.jolastu();  
    }  
}
```

Klase eta metodo konstanteak

- Klaseak eta metodoak konstante bihurtu daitezke **final** hitz erreserbatua gehitzen bazaie
 - Konstante diren klaseak **ezin dira hedatu**
 - Aldatu behar ez diren klaseak sortzeko erabiltzen dira, hala nola, String klasea
 - Konstante diren metodoak **ezin dira berridatzi**
 - Objektuaren integrazioa edo logika babesteko erabiltzen dira

```
public class Txakur {  
    public final void zaunkaEgin() {  
        System.out.println("Guau, guau");  
    }  
    ...  
}
```

```
public class Txakurtxo extends Txakur {  
    public void esneaHartu() {...}  
    public void zaunkaEgin() {  
        System.out.println("Miau, miau");  
    }  
}
```

Klase eta metodo konstanteak

- Bereziki eraikitzaileetan erabiltzen diren metodoak konstante bezala definitu behar dira
 - Azpiklase batek metodo hauek berridatzi ez ditzan (bestela espero ez den portaera bat lor daiteke)

```
public class Ausazko {  
    double ausaz;  
    public Ausazko() {  
        this.ausaz = sortuAusazko();  
    }  
    public double sortuAusazko() {  
        return Math.random();  
    }  
}
```

```
public class BesteAusazko extends Ausazko {  
    public BesteAusazko() {  
        super();  
    }  
    public double sortuAusazko() {  
        return 5;  
    }  
}
```

Enumerazio klaseak

- Balio konstanteez osatutako egitura bat sortzeko erabiltzen diren klaseak dira
 - **enum** hitz erreserbatua erabiltzen da hauek sortzeko eta atributu estatikoak bezala atzitzen dira
 - Enumerazio barruko balioak letra nagusiz idazten dira

```
public enum Maila{  
    BAXUA,  
    ERTAINA,  
    ALTUA  
}
```

```
public class Main {  
    public static void main(String[] args){  
        Maila unekoMaila = Maila.BAXUA;  
    }  
}
```

Enumerazio klaseak

- Enumerazio klaseak beste klase baten barruan sor eta erabili daitezke

```
public class Main {  
    enum Maila{BAXUA, ERTAINA, ALTUA  
    }  
    public static void main(String[] args){  
        Maila unekoMaila = Maila.BAXUA;  
    }  
}
```

