Exercise 1: Opgave 9.1 på CodeJudge.

a) Lav klasserne Cirkel, Rektangel og Kvadrat, som implementerer vedhæftede interface From.java (se CodeJudge).

   Lav Rektangel før du laver Kvadrat. Kan du bruge nedarvning og spare dig selv noget arbejde?

b) Udvid din implementation af forme til at understøtte .equals.

   To objekter af samme form er ens hvis de har samme værdier i felterne.

End of Exercise 1

Exercise 2: Dit team i firmaet ÅsynTobindsværk er sat til at lave en database over klienters bygningers relevante information. Dit team kommer frem til at I ikke gider gemme et generelt `Bygning` objekt, men forskellige delkategorier af bygninger, som f.eks. en `Lejlighed`, `FirmaBygning` etc. . Du når lige akkurat at skrive følgende kode, før en kollega stopper dig og siger "Du kan ikke instansiere et abstrakt objekt, og du instansiere et helt array af dem i databasens konstruktør!". Er påstanden korrekt? Hvis ja omstrukturer koden til at imødekomme kravene, hvis nej så forklar hvad der er forkert ved påstanden.

```
class Database{
        Bygning[] bygninger;

        Database Database(){
                bygninger = new Bygning[100000];
                //...
        }
}

abstract class Bygning{
        int antal_folk;
        int samlet_ad_revenue;
        boolean indeholder_aktive_brugere();
}

class Boligblok extends Bygning{
        //...
}

class FirmaBygning extends Bygning{
        //...
}
```

End of Exercise 2

Exercise 3: Du skal lave en model af en zoologisk have. Kig på filen Zoo.java på CodeJudge og implementer klasser, så koden kan køre og printer følgende:

```
Snake is an animal with 0 legs that lays eggs.
Turtle is an animal with 4 legs that lays eggs.
Elephant is an animal with 4 legs that gives birth.
Giraffe is an animal with 4 legs that gives birth.
Zebra is an animal with 4 legs that gives birth.
Bull ant is an animal with 6 legs that lays eggs.
```

Tænk grundigt over, hvordan du kan benytte abstrakte klasser og nedarvning til at skrive så effektiv kode som muligt. Udnyt eksempelvis, at alle reptiler lægger æg, og at alle insekter har seks ben.

End of Exercise 3

Exercise 4: Du er i gambling-firmaet Den52ArmedeTyveKnægt blevet sat til at lave et kortspils-simulation i Java.

a Lav en class `Card` der har følgende konstruktør

```
public Card(String suit, int value){
    //...
}
```

Hvor `suit` er kuløren af kortet, "spades", "heart", "clubs" og "diamonds", og `value` er en værdi i intervallet $[0; 12]$ der hhv. repræsenterer 2, 3, 4, 5, 6, ... king, ace.

b Lav en passende `toString()` metode for `Card`, så den følger hvordan man vil læse kortet op, f.eks. "9 of hearts", "queen of diamonds" etc.

c Lav en class `Deck` hvori der indgår 52 kort, dvs. 13 af hver kulør uden jokere. Lad et helt deck blive skabt i constructoren.

```
public Deck(){
    Card[] deck = new Card[52];
    //...
}
```

d For at sikre sig at spillet er fair og folk ikke udnytter at constructoren skaber kortene i en bestemt rækkefølge, skal du implementere et blandings-metode "shuffle" i `Deck` classen, som constructoren benytter.

```
public void shuffle(){
    //...
}
```

e For at få simulationen til at køre helt perfekt, indså Den52ArmedeTyveKnægt at de nogle gange vil kunne sammenligne kortene med følgende prioriteringer:

1. $\heartsuit < \diamondsuit < \clubsuit < \spadesuit$. (heart mindre end diamonds, diamonds mindre end clubs og clubs mindre end spades.)
2. $2 < 3 < 4 < \ldots <$ king $<$ ace

Dvs. at hvis du sammenligner 2 of clubs og 4 of clubs, er 2 of clubs mindst. Men hvis du har 4 of clubs og 2 of spades, så er 4 of clubs mindst, idet kuløren er prioriteret højere.

Implementer Javas `Comparable` interface i `Card` klassen.

Bonus: nu kan kortene i `Deck` sorteres af `Arrays` klassen. Få den til at fungere med følgende kode:

```
public void SortDeck(){
        Arrays.sort(this.deck);
}
```

f (∗) Nogle klienter synes, at shuffle metoden ikke *føles* tilfældig nok. Selvfølgelig har kunden altid ret, så tilpas din shuffle-metode således at 5 på hinanden følgende kort ikke er i stigende rækkefølge i forhold til prioriteringerne givet.

End of Exercise 4

Exercise 5: Lav den abstrakte klasse Skakbrik:

```
public abstract class Skakbrik extends Point {
```

Der holder styr på brikkens farve, position og briktype (springer, løber, osv).
Med position menes her par af heltal (x, y) som ligger i $[0, 7] \times [0, 7]$.

- Skakbrikklassen skal have en passende `toString()`-funktion, der printer således:

  ```
  sort springer F6
  ```

  Her bliver værdien for `x` omsat til et bogstav mellem A og H, og værdien for `y` bliver omsat fra intervallet $[0, 7]$ til intervallet $[1, 8]$.

- Giv klassen en passende konstruktører, der initierer brikken hvis den er indenfor pladen, og kaster en exception ellers. Konstruktørmetoden må gerne, ud over koordinaterne x og y, tage farve og briktype som argument.

- Lav klasserne for løber, springer, dronning, tårn og konge, så de udvidder skakbrikklassen. (Loeber, Springer, Dronning, Taarn og Konge.)

  ```
  public class Springer extends Skakbrik {
  ```

  Lav passende konstruktører for dine brikker. Testeren i CodeJudge kalder dem med:

  ```
  Springer minLillePony = new Springer(5,3,"hvid");
  ```

- Implementer passende translate-metoder for alle brikker. Testeren i CodeJudge kalder dem med

  ```
  minLillePony.translate(1,2);
  ```

  Her kan du virkelig udnytte at du har en abstrakt klasse!

  For at teste om trækket er lovlig skal du nemlig teste to ting:

  1. trækket er passende for den givne briktype og
  2. trækket flytter ikke brikken udenfor brættet.

  Overvej flere løsningsmuligheder, diskuter dem gerne med læsegruppen, og implementer så en velovervejet og velvalgt løsning, der undgår redundans.

End of Exercise 5

Exercise 6:

- Nedenstående klasse har en konstruktør. Betyder det, at man kan oprette en instans af den (`Fun clownPerson = new Fun(10);`)?

```
public abstract class Fun {
        public int funAmount;
        public abstract String makeLaughter();
        public Fun(int funAmount) {
                this.funAmount = funAmount;
        }
}
```

- Gør det en forskel, hvis `makeLaughter();` bliver implementeret, så der ingen abstrakte metoder er?
- Hvad betyder det, at alle felter i et interface er public static final?
- Kan et interface have konkrete metoder?*
- Er metoderne i et interface altid public static?**

End of Exercise 6

Exercise 7: Hvad går galt i følgende program, og hvorfor?

```
public class AnimalShelter {

        public static void main(String[] args) {
                Dog pet1 = new Dog("Boris");
                Cat pet2 = new Cat("Dorit");

                pet1.eat(5000);

                System.out.println(pet2.pet());

                fourLeggedCreature[] pets = {pet1, pet2};

                for (int i = 0; i < pets.length; i++) {
                        System.out.println(pets[i].getName());
                }
        }
}


abstract class fourLeggedCreature {
        boolean hasFur;
        int weight;

        abstract void eat(int kilogram);
}
```

```
class Dog extends fourLeggedCreature{
        String name;
        void eat(int kilogram) {
                weight += kilogram;
        }

        Dog(String name){
                hasFur = true;
                weight = 30;
                this.name = name;
        }

        String getName() {
                return this.name;
        }
}

class Cat extends fourLeggedCreature{
        String name;
        void eat(int gram) {
                weight += gram/2;
        }

        Cat(String name){
                hasFur = true;
                weight = 5;
                this.name = name;
        }

        public String pet(){
                return "Puurrrr";
        }

        String getName() {
                return this.name;
        }
}
```

End of Exercise 7