Hierbij wat feedback over de proeftest van Software-ontwikkeling.
Om niet misbegrepen te worden: NIET alle code is VOORBEELDcode. Er staat ook TEGENVOORBEELDcode: zo moet het NIET. Dus lees aandachtig de tekst die bij de code hoort.

Klasse Krijger

1. Eerste opmerking: jullie kregen de klasse `Krijgers` (MET EEN 's' OP HET EINDE), en deze klasse bevatte de main-methode. Dan is het niet zo'n goed idee om in DEZE klasse een Krijger te gaan definiëren. Maak daarvoor een nieuwe klasse aan, met naam `Krijger` (ZONDER 's' OP HET EINDE).
2. Gezien elke Krijger ofwel een Ninja, ofwel een Samoerai moet zijn, maak je de klasse Krijger abstract. Het heeft immers geen zin toe te laten dat er een object van de klasse Krijger wordt aangemaakt.
3. Je moet Krijgers kunnen vergelijken; dus implementeer je de methode `compareTo(Object o)`. Of je kiest ervoor om enkel de methode `compareTo(Krijger k)` te implementeren. In het eerste geval komt er dan bij de hoofding van de klasse `implements Comparable`, in het tweede geval `implements Comparable<Krijger>`.
   Merk op:
   - de tweede versie, die type-safe is, bestond voor JDK 1.5 niet. Tot JDK 1.4 was enkel de eerste optie mogelijk.
   - er is een verschil tussen `implements Comparable` en `implements Comparator`!!!

   Oplossing:

```
@Override
public int compareTo(Object o) {
    return this.compareTo((Krijger) o);
}
public int compareTo(Krijger k){
    ... // geen testen op null
        // (API verwacht nullpointerException in plaats van een rangschikking)
        // evenmin testen op klasse
        // (API verwacht ClassCastException in plaats van een rangschikking)
}
```

   Indien de parameter `o` dan geen Krijger blijkt te zijn, zal er een exceptie opgeworpen worden. De Java API van de interface `Comparable<T>` geeft duidelijk aan welke excepties geworpen moeten worden; zie deze link.

   In de unitTest kan je ook testen of de juiste excepties opgeworpen werden. Een mogelijke oplossing daarvan:

```
@Test
public void testCompareToObject() {
    System.out.println("compareTo(Object)");
    Krijger krijger = new Ninja(20,"Ninja");
    try {
        krijger.compareTo(null);
        fail("Moet NullPointerException gooien");
    } catch (NullPointerException e) {
        assertTrue(true);
    }
    try {
        krijger.compareTo(new Object());
        fail("Moet ClassCastException gooien");
    } catch (ClassCastException e) {
        assertTrue(true);
    }
}
```

   Of, je kan de test in stukjes knippen, en aangeven in de hoofding van de testmethode dat er een exceptie verwacht wordt (en welke).

```
@Test(expected = NullPointerException.class)
public void testNullptrs() {
    System.out.println("compareTo(Object)");
    Krijger krijger = new Ninja(20,"Ninja");
    krijger.compareTo(null);
}
```

4. Let op efficiëntie. De code die hieronder staat, doorloopt sowieso x, ook als getal!=0. Tijdverlies. (Voor de computer. Maar ook een beetje voor de lezer, die zich afvraagt welke constructie (en waarom) daar bovengehaald wordt.)

```
@Override                                                          // INEFFICIENTE CODE
public int compareTo(Object o){                                    // INEFFICIENTE CODE
    Krijger k = (Krijger) o;                                       // INEFFICIENTE CODE
    int getal = naam.compareTo(k.getNaam());  // hier bereken je 'getal' // INEFFICIENTE CODE
    int x = 0;                                // hier bereken je 'x'     // INEFFICIENTE CODE
```

```
        if(this.levend() && k.levend()){                                    // INEFFICIENTE CODE
            x = 0;                                                          // INEFFICIENTE CODE
        } else if (this.levend()){                                          // INEFFICIENTE CODE
            x = -1;                                                         // INEFFICIENTE CODE
        } else if(k.levend()){                                              // INEFFICIENTE CODE
            x = 1;                                                          // INEFFICIENTE CODE
        }                                                                   // INEFFICIENTE CODE
        if(getal==0){ // hier beslis je dat je 'x' misschien helemaal niet  // INEFFICIENTE CODE
            return x; // nodig hebt...                                      // INEFFICIENTE CODE
        } else {                                                            // INEFFICIENTE CODE
            return getal;                                                   // INEFFICIENTE CODE
        }                                                                   // INEFFICIENTE CODE
    }
```

5. Let op! De test `if(this==null)` is hier NIET op zijn plaats! Uiteraard kan het huidige object niet null zijn, want dan zou je de huidige methode niet kunnen oproepen. Totaal overbodige test, dus.
6. In de opgave stond duidelijk 'verdedigings-/aanvalskrachten worden telkens opnieuw berekend'. Dan is het expliciet een slecht idee om deze gegevens als instantievariabele bij te houden. Niet in de bovenklasse Krijger, niet in de afgeleide klassen Ninja of Samoerai! Herberekenen is herberekenen, en niet ergens opslaan. En uiteraard heeft het geen zin om deze krachten ergens in de constructor te herberekenen. Ten eerste heeft niemand ze daar nodig (je mag ze toch niet opslaan), en ten tweede veranderen ze toch tijdens de 'levensloop' van het object.
7. Een afgeleide klasse kan niet aan de instantievariabelen van de bovenklasse, tenzij die `protected` in plaats van `private` gedeclareerd zijn. Doe dat dan: maak de nodige instantievariabelen protected in de bovenklasse. Wat uiteraard helemaal not done is: diezelfde instantievariabelen nogmaals declareren in de afgeleide klasse.
8. Als één van de methodes in een klasse een randomgenerator nodig heeft, mag je die als instantievariabele declareren en meteen initialiseren. Stel dit niet uit tot in de methode zelf, want dan maak je telkens nieuwe generatoren aan per getal dat je random moet genereren.
9. Houd je code overzichtelijk. Vergelijk hieronder leesbaarheid en lengte.

```
    public boolean isAlive() {          // NIET ERG PROPERE CODE
        boolean levend = true;          // NIET ERG PROPERE CODE
        if (this.getLevens() <= 0) {    // NIET ERG PROPERE CODE
            levend = false;             // NIET ERG PROPERE CODE
        }                               // NIET ERG PROPERE CODE
        return levend;                  // NIET ERG PROPERE CODE
    }

    public boolean isAlive(){           // NETJES OPGEKUIST
        return aantalLevens > 0;        // NETJES OPGEKUIST
    }
```

   Heb je ook gemerkt dat het gebruik van de getter `.getLevens()` overbodig is? Omdat de methode `isAlive` tot dezelfde klasse behoort als de instantievariabele `aantalLevens`.
10. Het is niet de bedoeling dat je naar lieverlee een heleboel extra getters (oei), setters (foei) en constructoren bijmaakt. Houd je aan de opgave, zodat je op tijd aan de volgende klassen kan beginnen - en daar kan bewijzen dat je het OOK ZONDER die extra setters/constructoren kan. Een voorbeeld ter illustratie:

```
public class Krijgers {
    String naam;
    int levens,gewonnen,aKracht,vKracht; // teveel!

    public Krijgers(String naam, int levens, int gewonnen, int aKracht, int vKracht){...} // niet gevra
    public Krijgers(String naam, int levens, int gewonnen){...} // niet gevraagd!

    public boolean isLevend(){...}     // ok
    public String isLevendString{...}  // niet gevraagd
    public int getAantalLevens(){...}  // ok
    public void setAantalLevens(){...} // niet gevraagd
    public String getNaam(){...}       // ok
    public void incGewonnen(){...}             // ok, maar naamgeving kon beter
    public void decGewonnen(){...}             // niet gevraagd
    public void incLevens(){...}               // niet gevraagd
    public void decLevens(){...}               // ok, maar naamgeving kon beter
    public int getAanvalsKracht(){...}         // ok - mits BEREKEND
    public void setAanvalsKracht(){...}        // niet gevraagd
    public int getVerdedigingsKracht(){...}    // ok - mits BEREKEND
    public void setVerdedigingsKracht(){...}   // niet gevraagd
    public int getGewonnen(){...}              // ok
    public boolean isGewonnen(Boolean b){...} // niet gevraagd
    public void setGewonnen(int gewonnen){...}// niet gevraagd
    public void gevechtMet(Krijger){...}       // niet gevraagd
    @Override public int hashCode(){...}       // niet gevraagd, maar is je vergeven
    @Override public boolean equals(Object obj){...} // idem
```

11. Nog een tip om tijd te besparen: het is edelmoedig om alle mogelijke verkeerde input op te proberen vangen. Maar doorgaans verlies je daarmee de belangrijke zaken uit het oog - en je tijd. Ook hier een voorbeeld:

```
public Krijger(String naam, int levens) {                                    // ok
    this.naam = new String(naam);                                            // ok
    if( levens < 0){                                                         // niet doen
        throw new RuntimeException("Krijger levens kunnen niet negatief zijn!"); // niet doen
    }                                                                        // niet doen
    this.levens = levens;                                                    // ok
}
```

(Ja, het is properder en veiliger. Maar je moet keuzes maken als je een test aflegt... en dan laat je misschien beter zien dat je tot het einde van de opgave raakt, voor je dergelijke extra's inbouwt.)

Testklasse

12. Deze test heeft niet zoveel zin: `assertEQuals(3,4,0.0001)`. Want dan ga je na of de twee GEHELE getallen 3 en 4 al dan niet gelijk zijn, op een fractie na. Als je het zo ziet staan is het logisch, maar hier is het al wat subtieler - doch even fout:

    ```
    double EPS=0.00001;
    assertEquals(1,k.getGewonnenGevechten(),EPS);
    ```

13. Gegeven onderstaande testcode. De methodes `getAanvalskracht()` en `getVerdedigingskracht()` zouden een getal binnen bepaalde grenzen moeten genereren. In deze test wordt dat gecontroleerd. Welke redeneerfout werd er gemaakt?

    ```
    @Test
    public void TestAanvalVerdediging(){
        Assert.assertTrue(n.getAanvalskracht()>=0 && n.getAanvalskracht()<450);
        Assert.assertTrue(n.getVerdedigingskracht()>=0 && n.getVerdedigingskracht()<480);
        Assert.assertTrue(s.getAanvalskracht()>=0 && s.getAanvalskracht()<164);
        Assert.assertTrue(s.getVerdedigingskracht()>=0 && s.getVerdedigingskracht()<200);
    }
    ```

    Antwoord: het heeft weinig zin om te testen of random-gegenereerde getallen inderdaad binnen de onderstelde grenzen liggen. Er is immers veel kans dat, óók indien je de grenzen per ongeluk verkeerd instelde, dat éne geval dat je hier test, toch binnen de verhoopte grenzen blijft. Beter idee: laat deze test zo dikwijls lopen, dat er statistisch gezien een grote kans is dat je het hele bereik bestreken hebt. Dus zet de test in een `for(int i=0; i<100000000; i++)`-lus.

    Toegespitst op de context waarin deze code geschreven werd: er waren testen die voorrang verdienden, bijvoorbeeld de test of `compareTo` in orde was.

KrijgerVergelijker

14. Deze klasse hebben we nodig omdat we, NAAST de 'natuurlijke' sorteermethode die we in de klasse `Krijger` zelf al implementeerden, een bijkomende sorteermethode willen hebben. Hoofding van deze klasse:

    ```
    public class KrijgerComp implements Comparator {
    ```

    Ook hier kan je kiezen voor `implements Comparator` (ook voor vergelijking met niet-Krijgers) dan wel `implements Comparator<Krijger>`. Je editeeromgeving wijst je dan verder de weg voor de hoofding van de methode `compare(...,...)` die je moet overloaden.

    Raadpleeg ook hier de API (en merk op dat er voor `null` keuze gelaten wordt).

15. Merk op: bij het implementeren van de methode `compare(Krijger,Krijger)` uit de klasse `KrijgerVergelijker`, kan je aan de protected instantievariabelen van de klasse `Krijger`. Je hoeft dus niet via methodes van `Krijger` te gaan als je de info ook rechtstreeks uit de instantievariabelen kan halen. Dit komt omdat beide klassen (`KrijgerVergelijker` en `Krijger`) in dezelfde package staan. (Merk op: in `C#` zou dit niet het geval zijn. Zie cursus Softwareontwikkeling II.)

DAODummy

16. De bestaansreden van een Dummyklasse: je zorgt dat je hier een aantal (hardgecodeerde) objecten hebt klaarstaan, waarop je dan testen kan uitvoeren. Je plaatst de hardgecodeerde namen dus ook binnen die klasse; mag gerust als klassevariabelen (`static`) en niet wijzigbaar (`final`).

    ```
    public class DAODummy ...{
        private final String[] namen = {"Donatello", "Mich", "Raf", "Schreder", "Bruce", "Arnold"};
        private final int[] gewonnen = {2, 3, 2, 3, 3, 3};
        private final int[] levens = {2, 4, 2, 0, 3, 3};
        ...
    }
    ```

17. Een tussendoortje. Toon aan dat, met enkele minieme ingrepen, het drie keer zo kort (én drie keer zo leesbaar) kan. (Voor de rest geen uitspraken over de juistheid. Tenzij dit: er werd een setter gebruikt die verondersteld werd niet te bestaan.)

```
    public KrijgerDAODummy() {                                                  // NIET GOED
        int i=0;                                                                // NIET GOED
        int a;                                                                  // NIET GOED
        Ninja nj;                                                               // NIET GOED
        lijst = new ArrayList();                                                // NIET GOED
        String naam;                                                            // NIET GOED
        int leven;                                                              // NIET GOED
        int gewonnenDuel;                                                       // NIET GOED
        while(i<6){                                                             // NIET GOED
            naam = namen[i];                                                    // NIET GOED
            leven = levens[i];                                                  // NIET GOED
            gewonnenDuel = gewonnen[i];                                         // NIET GOED
            nj = new Ninja(naam, leven);                                        // NIET GOED
            nj.setAantalGewonnen(gewonnenDuel);                                 // NIET GOED
            lijst.add(nj);                                                      // NIET GOED
            i++;                                                                // NIET GOED
//            System.out.println(nj.getNaam()+ " aantal gewonnen: "             // NIET GOED
//            +nj.getAantalGewonnen() + " aantal levens: " + nj.getLevens());   // NIET GOED
            }
    }
```

KrijgersDAO

18. De klasse `KrijgersDAO` is de klasse die verschillende krijgers tegelijk beheert, en sorteert op verschillende manieren. Belangrijk: gevraagd werd de klasse `abstract` te maken. Uiteindelijk moet de klasse drie dingen kunnen: krijgers in oorspronkelijke volgorde uitschrijven, in natuurlijke volgorde, in speciale volgorde.
HOE de krijgers bewaard worden (in een set, list, array,...) ligt niet vast in deze klasse. Deze klasse heeft dus geen instantievariabelen. De afgeleide klasse (DAODummy) heeft dat wel - daar had je trouwens al besloten (hopelijk...) dat een array of list als instantievariabele de enige juiste keuze was. (Bewaar je de krijgers meteen in een set, dan is hun oorspronkelijke volgorde verloren.)
We verkrijgen dus:

```
public abstract class KrijgerDAO {                                              // OK

    public abstract Krijger[] getKrijgers();                                    // OK
    // dit VOORAL NIET HIER IMPLEMENTEREN, zodat je echt een abstracte
    // klasse hebt! (Je KAN het trouwens niet implementeren, omdat je geen
    // instantievariabelen hebt.)

    public SortedSet<Krijger> getKrijgersGesorteerd(){...}                  // OK
    // deze methode IMPLEMENTEER JE WEL hier:
    // omdat je getKrijgers() kan oproepen, heb je de array met alle krijgers.
    // Deze array steek je in een SortedSet, en het sorteren gebeurt automatisch.


    public SortedSet<Krijger> getKrijgersGesorteerdSpeciaal(){...}          // OK
    // idem
}
```

19. Het is niet omdat een klasse abstract is, dat je implementatie van ALLE methodes MOET uitstellen tot in de afgeleide klassen. Hieronder staat wat code, met een vraag bij geformuleerd.

```
public abstract class KrijgersDAO {

    public SortedSet<Krijgers> getKrijgers(){                   // NIET OK, moet abstract zijn
        return null;
    }
    public SortedSet<Krijgers> getKrijgersAlfa(){               // OK mits implementatie
        return null;
    }
    public SortedSet<Krijgers> getKrijgersComp(){               // OK mits implementatie
        return null;
    }
//    Dit is volgens mij raar dat er hier moet gereturnd worden.
//    Dit is een abstracte klasse dus de body's moeten niet gedeclareerd worden.
}
```

    Het antwoord op de vraag: de methode `getKrijgers()` maak je abstract, omdat je ze niet kán implementeren zolang je niet weet hoe de afgeleide klasse de Krijgers bewaart (in een list of een array of nog anders). De andere twee methodes laat je niet-abstract (zoals ze er nu staan) en je implementeert ze onmiddellijk - je kan de methode `getKrijgers()` gebruiken om aan de (niet-gesorteerde) collectie krijgers te geraken.
20. Hieronder wat code die we in vorige studentenversies tegenkwamen, met commentaar.

```
(1)
    ArrayList<Krijger> a = returnKrijgers();    // ok
    a.sort((Comparator)new KrijgerVergelijker());  // niet goed
    Collections.sort(a,new KrijgerVergelijker());  // wel goed

(2)
    Krijgers[] krrr= krr.clone(); // waarom klonen?
                // allicht omdat je beseft dat de sort hieronder ervoor zorgt
                // dat je de krijgers niet meer in oorspronkelijke volgorde hebt
    sort(krrr); // waarom sorteer je de array? dat gebeurt automatisch bij objecten
                // toevoegen aan TreeSet.
                // Dus was het klonen ook niet nodig.
    TreeSet set=new TreeSet();
    set.addAll(Arrays.asList(krr));
    return set;
```

21. Merk op: als je start met 10 krijgers, en je vraagt aan de klasse `DAO` om deze op drie verschillende manieren uit te schrijven, dan heb je nog altijd maar 10 objecten van de klasse Krijger, en GEEN 30! Enkel de verwijzingen naar 10 bestaande objecten worden 'ontdubbeld': er zijn 10 verwijzingen die in een array zitten en de oorspronkelijke volgorde onthouden, er zijn 10 verwijzingen die in een SortedSet zitten en diezelfde 10 objecten aanwijzen in natuurlijke volgorde, en analoog voor de laatste SortedSet. Dit zou in C++ (gegeven dat je OBJECTEN bewaart in de containers, en geen pointers...) wel even anders zijn!

    Je mag de implementatie echter NIET uitwerken zoals in onderstaande code, maar je moet wel degelijk een nieuwe set (of collectie) opbouwen aan de hand van een andere.

    ```
    public List geefKrijgers(){
        return krijgers;
    }

    public List geefKrijgersInNatuurlijkeVolgorde(){ // FOUT
        Collections.sort(krijgers);                  // FOUT
        return krijgers;                             // FOUT
    }
    ```

22. Er werd uitdrukkelijk gevraagd om de methode(s) `getKrijgersGesorteerd()` een `SortedSet` te laten teruggeven. De declaratie van je variabele gebruikt de `SortedSet` (de interface), de initialisatie gebruikt `TreeSet` (de klasse die de interface implementeert).

    ```
    public SortedSet<Krijger> getCustomGesorteerdeKrijgers(){
        SortedSet<Krijger> ts = new TreeSet<>(new KrijgerVergelijker());
        // mag ook:
        // TreeSet<Krijger> ts = new TreeSet<>(new KrijgerVergelijker());
        ts.addAll(krijgers);
        return ts;
    }
    ```

Main

23. In de gegeven klasse `Krijgers` stond de main-methode al klaar. Je moet drie keer iets uitprinten: een list of een array, en twee keer een sorted set. Uiteraard vermijd je duplicated code door een print-methode te schrijven. Deze moet dan zowel een list en/of array als een set aanvaarden als parameter. Gebruik daarvoor `Iterable`.