Föreläsning 2

Innehåll

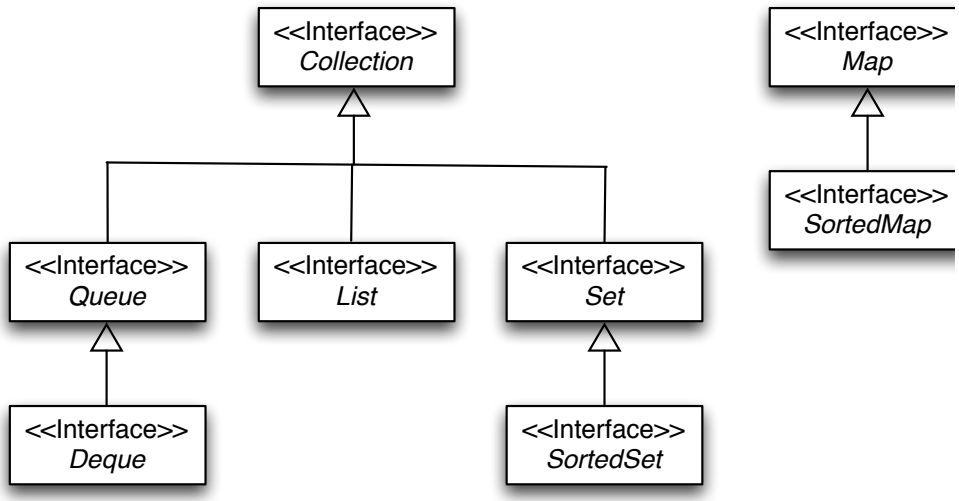
- Java Collections Framework (interface och klasser för samlingar av element)
- Använda generiska klasser
- autoboxing - och unboxing
- Iterera genom en samling element
- Jämföra element
  - skugga `equals`

Java Collections Framework

- Är en hierarki av interface, abstrakta klasser och konkreta klasser för samlingar av element.
- Dokumentation av Javas standardklasser och interface finns på Javas webbsida. Där finns även tuturials, bl.a. om Javas Collection-klasser.
- Basen i hierarkin är ett interface `Collection`:

```
interface Collection<E> {
   boolean add(E x);
   boolean contains(Object x);
   boolean remove(Object x);
   boolean isEmpty();
   int size();
   ...
}
```


Java Collections Framework – interface hierarki



Java Collections Framework – interface hierarki

`Collection`	en samling av element, där dubbletter tillåts
`List`	en samling element där positionering är möjlig (första, sista, element på plats i, ...)
`Queue`	en samling av element som utgör en kö
`Deque`	som `Queue` men man kan sätta in och ta ut element både i början och i slutet av kön
`Set`	en samling element där dubbletter är förbjudna
`SortedSet`	som `Set` men med krav att elementen går att jämföra
`Map`	en samling nyckel-värde-par (jfr. lexikon)
`SortedMap`	som `Map` men med krav att nycklarna går att jämföra

Java Collections Framework – några klasser

Interface	Klass
`Queue`	`ArrayDeque`, `LinkedList`, `PriorityQueue`
`Deque`	`ArrayDeque`, `LinkedList`
`List`	`ArrayList`, `LinkedList`
`Set`	`HashSet`
`SortedSet`	`TreeSet`
`Map`	`HashMap`
`SortedMap`	`TreeMap`

Generiska klasser

- Javas samlingsklasser är generiska.
- En generisk klass har en eller flera **typparametrar**. Ex:

```
public class ArrayList<E> {...}
```

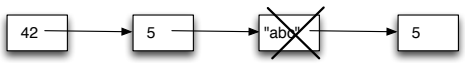
- Vid användning av en generisk klass anges ett **typargument**:

```
ArrayList<Integer> myList = new ArrayList<Integer>();
```


Varför generiska klasser

Bakgrund

- Klasser bör implementeras så att de blir generella d.v.s. går att använda i många olika sammanhang.
  - En klass som hanterar en lista av *element* ska inte skrivas så att den bara kan hantera listor med heltal.
- Men vi vill inte ha en lista där vi kan sätta in objekt hur som helst.
  - I en lista med heltal vill vi inte av misstag kunna sätta in `String`-objekt.



Generik i Java

- Generik hindrar oss att göra fel.
  - I en lista av typen `ArrayList<String>` kan man bara sätta in strängar.
- Kompilatorn kommer att upptäcka typfel. Ex:

```
ArrayList<String> myList = new ArrayList<String>();
myList.add(123);
```

ger nu kompileringsfel. Listan `myList` får enligt sin deklaration endast innehålla objekt av typen `String`.

Exempel på en generisk klass

`java.util.ArrayList`

Utdrag ur den generiska klassen `java.util.ArrayList`:

```
public class ArrayList<E> {
   public ArrayList() {...}
   public boolean add(E x) {...}
   public void add(int index, E x) {...}
   public E get(int index) {...}
   ...
}
```

Alla add-metoder har inparameter av typen `E`. Därför kan enbart objekt av klassen `E` (eller subklasser till denna) sättas in i listan.

Exempel på en generisk klass

`java.util.ArrayList, forts`

Observera att inte alla metoder i `ArrayList<E>` har `E` som typparameter. T.ex. har sökmetoderna `contains`, `indexOf` och `remove` följande signaturer:

```
public boolean contains(Object x);
public int indexOf(Object x);
public boolean remove(Object x);
```

Användare som inte känner till den exakta typen av ett element `x` ska kunna anropa metoderna. Dock kommer man att få resultat `true` om och endast om `x` finns i listan (och alltså även är av rätt typ).

Restriktioner för typargument

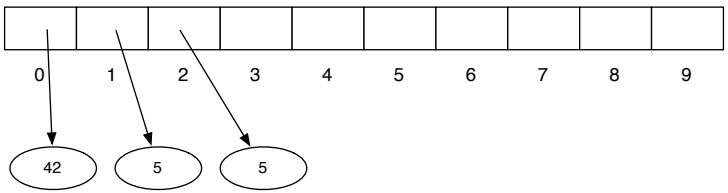
- Typargumentet får *inte* vara av primitiv typ:

```
List<int> list = new ArrayList<int>(); // Går inte!
```

- Istället får man använda motsvarande wrapperklass:

```
List<Integer> list = new ArrayList<Integer>();
```

- I en lista av typen `ArrayList<Integer>` lagras alltså elementen i en vektor med `Integer`-objekt:



Primitiva datatyper - wrapperklasser

Primitiva typer i Java:

```
boolean
short
int
long
char
byte
float
double
```

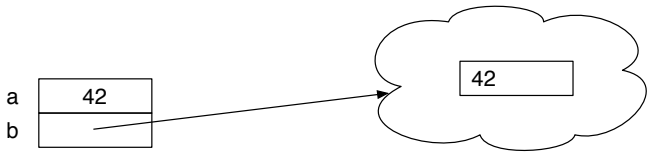
Motsvarande wrapperklasser:

```
Boolean
Short
Integer
Long
Character
Byte
Float
Double
```

Primitiva datatyper - wrapperklasser

Exempel

```
int a = 42;
Integer b = new Integer(42);
```



Variabeln a har värdet 42, medan variabeln b innehåller en referens till ett Integer-objekt.

Diskutera

Antag att vi lagrar heltal i en lista:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
int j = list.get(0);
```

Hur kan detta fungera?

- Listan innehåller Integer-objekt.
- På andra raden sätter vi in ett heltal (typen int).
- På tredje raden har vi deklarerat en variabel j av typen int. Men metoden get returnerar en referens till ett Integer-objekt.

Autoboxing – unboxing

Autoboxing automatisk konvertering från primitiv typ till objekt av motsvarande wrapperklass

Unboxing automatisk konvertering av objekt av wrapperklass till motsvarande primitiva typ

Exempel:

```
Integer k = 3;  // autoboxing
int j = k;      // unboxing
Integer i = new Integer(2);
i = i + 1;      // Unboxing av i för att beräkna i+1.
                // Därefter autoboxing av resultatet vid
                // tilldelningen.
```

I tidiga versioner av Java var man tvungen att skriva

```
i = new Integer(i.intValue() + 1);
```


Autoboxing – unboxing

i samband med generiska klasser

Praktiskt när man vill använda en generisk klass för att lagra element av primitiv typ. Ex:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);   // Autoboxing till Integer-objekt här.
...
int j = list.get(0);  // Unboxing till int här.
```

Generisk klass med flera typparametrar

- Exempel: Interfacet `Map<K, V>` och klassen `HashMap<K, V>` har två typparametrar.
  - K som i key, V som i value
- Exempel på användning:

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("June", 30);
```


Exempel på användning av klassen Map

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("Januari", 31);
map.put("Februari", 28);
map.put("Mars", 31);
map.put("April", 30);
map.put("Maj", 31);
...

System.out.println("Antal dagar i mars: " + map.get("Mars"));
```


Diskutera

Antag att vi lagrar ett antal tal i en samling element, t ex en lista eller en mängd:

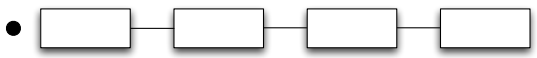
```
List<Integer> list = new ArrayList<Integer>();
// här sätts tal in i listan

Set<Integer> set = new HashSet<Integer>();
// här sätts tal in i mängden
```

Hur ska vi göra för iterera genom en samling element och behandla alla elementen (t.ex. summera talen i listan respektive mängden)?

Iteratorer

- För att iterera genom (traversera) listor eller andra samlingar kan man använda ett iteratorobjekt.
- Ett iteratorobjekt håller reda på en position i listan:



Iteratorer
Interfacet Iterator<E>

- Ett iterator-objekt är en instans av en klass som implementerar interfacet `Iterator<E>`.

```
public interface Iterator<E>  {
    /** Returns true if the iteration has more elements. */
    boolean hasNext();

    /** Returns the next element in the iteration. */
    E next();

    /** Removes from the underlying collection the last
        element returned by the iterator (optional). */
    void remove();

    ...
}
```


Användning av iterator
Exempel

Summera talen i listan `list` av typen `ArrayList<Integer>`:

- Iterera genom listan med explicit iterator:

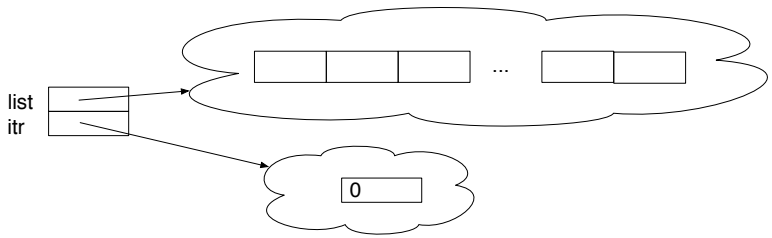
```
Iterator<Integer> itr = list.iterator();
int sum = 0;
while (itr.hasNext()) {
   int n = itr.next();
   sum += + n;
}
```

- Samma sak med for-each-sats:

```
int sum = 0;
for (int n : list) {
    sum += n;
}
```


Iteratorer – under huven

- Till `ArrayList<E>` hör en speciell iteratorklass (vars namn vi inte vet).
- Denna iteratorklass implementerar interfacet `Iterator<E>`. I iteratorklassen finns attribut som håller reda på iteratorns position.
- I klassen `ArrayList<E>` finns en metod `iterator()` som returnerar en referens till ett nyskapat iteratorobjekt.



Användning av iterator
Vanliga fel

- Iteratorobjektet skapas inuti metoden `iterator()`.

```
Iterator<Integer> itr = new Iterator();  // FEL
Iterator<Integer> itr = list.iterator(); // RÄTT
```

- För varje anrop av `next()` hoppar iteratorn fram ett steg i listan.

```
while (itr.hasNext()) {
    if (itr.next() > 0) {   // FEL - next anropas flera gånger
        sum = += itr.next();
    }
}

while (itr.hasNext()) {
    int n = itr.next();     // RÄTT
    if (n > 0) {
        sum = += n;
    }
}
```

Interfacet Iterable<E>

- I Java finns också följande interface:

```
public interface Iterable<E>  {
   /** Returns an iterator over a set of elements of type E *
   Iterator<E> iterator();

   ...
}
```

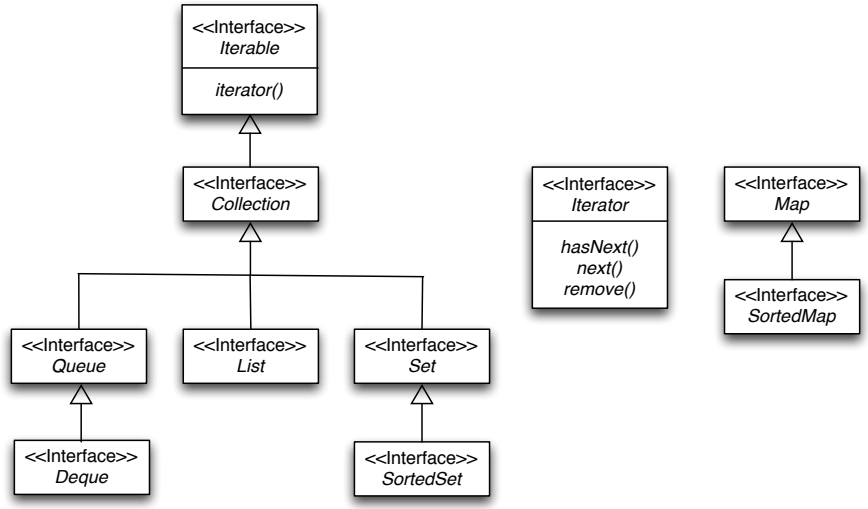
- Interfacet `Collection` ärver interfacet `Iterable`:

```
public interface Collection<E> extends Iterable<E> ...
```

Alla klasser som implementerar `Collection` måste alltså implementera metoden `iterator()`. Man kan alltså använda en iterator på objekt av alla dessa klasser.

Java Collections Framework – interface hierarki

med `Iterable<E>` och `Iterator<E>`



Interfacet ListIterator

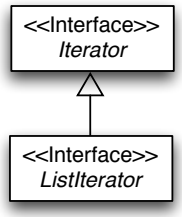
I klasser som implementerar interfacet `List` finns även dessa metoder:

```
/** Returnerar en listiterator som startar i pos i. */
public ListIterator listIterator(int i);

/** Returnerar en listiterator som startar i pos 0. */
public ListIterator listIterator();
```

`ListIterator<E>` är ett interface som ärver `Iterator<E>` och där man lagt till metoder för att röra sig även bakåt i listor samt för att sätta in element.

Interfacet ListIterator




```
public interface ListIterator<E> extends Iterator<E> {
   boolean hasPrevious();
   E previous();
   void add(E x);
   ...
}
```

foreach-sats i Java

Exempel

```
ArrayList<String> words = new ArrayList<String>();
// här sätts String-objekt in listan words
...
for (String s : words) {
    // behandla s
}
```

- Ett sätt att enkelt iterera över samlingar. Man slipper att explicit använda en iterator. Men `hasNext()` och `next()` anropas i bakgrunden.
- `for (String s : words) ...` kan läsas som "för varje `s` i `words`".
- Kan användas för objekt av klasser som implementerar `Iterable` och för vektorer.

foreach-sats

Vektorer

Kan också användas för att iterera över elementen i en vektor.

Ex: En metod som skriver ut alla strängarna i en vektor av typ `String[]`:

```
public void print(String[] a) {
   for (String s : a) {
      System.out.println(s);
   }
}
```


foreach-sats

Begränsningar

- Foreach-sats kan *inte* användas när man explicit behöver tillgång till iteratorn i koden. T.ex. vid borttagning av element under itereringen:

```
Iterator<Integer> itr = list.iterator();
while (itr.hasNext()) {
    if (itr.next() < 0) {
        itr.remove();
    }
}
```

- OBS! Man får *inte* använda listans egna metoder för att lägga till eller ta bort element under itereringen.
  - Använd iteratorns metod `remove` istället.
- Listans metod `removeIf` kan också användas.

foreach-sats

Begränsningar

Foreach-sats kan *inte* användas för att iterera parallellt över flera samlingar, motsvarande följande kod:

```
ArrayList<Person> list1, list2;
...
Iterator<Person> itr1 = list1.iterator();
Iterator<Person> itr2 = list2.iterator();
while(itr1.hasNext() && itr2.hasNext()) {
   System.out.println(itr1.next() + " " + itr2.next());
}
```

Nästlad foreach-sats

Däremot går det bra med nästlade loopar.

Ex: Skriv ut alla kombinationer av strängar `str1` `str2` där `str1` finns i vektorn `first` och `str2` finns i vektorn `second`. `first` och `second` är av typen `String[]` :

```
for (String f : first ) {
   for (String s : second) {
      System.out.println(f + " " + s);
   }
}
```


Traversera elementen i en samling

Utanför kursen, men kul att veta

- Fr.o.m. Java 8 finns metoden `forEach` i interfacet `Iterable`. Exempel:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add(11);
list.add(12);
list.forEach(e -> System.out.println(e));  // skriver ut alla tal
list.forEach(e ->  {                       // skriver ut alla jämna tal
    if (e % 2 == 0) {
        System.out.println(e);
    }
});
```

- Det som skicka med som argument till metoden `forEach` är ett *lambdauttryck* – ett stycke kod som, i det här fallet, ska utföras för varje element i listan. (Lambdauttryck behandlas senare under kursen.)

Filtrering med metoden `removeIf`

Utanför kursen, men kul att veta

I interfacet `Collection` finns metoden `removeIf` som kan användas för att filtrera bort element ur en lista.

Exempel 1: Filtrera bort alla negativa tal ur listan `list`:

```
list.removeIf(n -> n < 0);
```

Exempel 2: Filtrera bort alla udda tal ur listan `list`:

```
list.removeIf(n -> n % 2 != 0);
```

Argumentet är ett lambdauttryck (behandlas senare under kursen).

Strömmar

Utanför kursen, men kul att veta

Man kan även använda strömmar för att behandla alla element i en samling eller en vektor.

Exempel 1: Skriv ut alla personerna i listan `pList`:

```
pList.stream().forEach(p -> System.out.println(p));
```

Exempel 1: Skriv ut alla personer i listan `pList` som matchar `x`:

```
pList.stream().filter(p -> p.equals(x)).forEach(p ->
                                        System.out.println(p));
```

Du kan läsa mer om strömmar här:
https://docs.oracle.com/javase/tutorial/collections/streams/

Diskutera

Följande klass beskriver en person:

```
public class Person {
    private String name;
    private int id;

    public Person (String name, int id) {
        this.name = name;
        this.id = id;
    }
}
```

Vad skrivs ut när följande rader exekveras? Förklara varför?

```
ArrayList<Person> list = new ArrayList<Person>();
list.add(new Person("Fili", 1));
list.add(new Person("Balin", 2));
Person p = new Person("Balin", 2);
System.out.println(list.contains(p));
```


Jämföra likhet

Metoden equals

- Inuti `ArrayList` används metoden `equals` för att jämföra om två objekt är lika:

  ```
  if (o1.equals(o2)) {...}
  ```

- Metoden `equals` finns i superklassen `Object`. Den returnerar `true` om och endast om de jämförda objekten är identiska. Den testar referenslikhet och fungerar alltså som ==.

- Om vi istället vill att *innehållet* inuti objekten ska jämföras (här id-numren) måste vi skugga `equals` klassen `Person`.
  - Skiss, ej färdig equals-metod:

    ```
    public boolean equals(Object obj) {
       return id  == ((Person) obj).id;
    }
    ```


Söka i Javas listklasser

Sökmetoder som använder equals internt

```
// sök efter platsen för elementet med id-nummer 2
int index = list.indexOf(new Person(null, 2));

// tag reda på om det finns ett element med id-nummer 2 i listan
boolean found = list.contains(new Person(null, 2));

// tag bort elementet med id-nummer 2 ur listan
boolean removed = list.remove(new Person(null, 2));
```

- Använd dessa metoder istället för att skriva en egen sökloop.
- Observera att argumentet ska vara ett objekt av samma typ som elementen i listan.

Skugga equals – att tänka på

- Parametern till `equals` måste vara av typ `Object`, annars blir det inte skuggning och den ursprungliga metoden i klassen `Object` kommer att användas.

- De attribut som används i jämförelsen inuti `equals` bör inte gå att ändra. Deklarera dem `final`:

  ```
  public class Person {
      private String name;
      private final int id;
      ...
  ```

  Annars kan det bli svårt att hitta objektet när det satts in i en lista.

Skugga equals

Specifikation

```
public boolean equals(Object obj);
```

Ur equals specifikation:

- x.equals(x) ska returnera true (reflexivitet).
- Om x.equals(y) returnerar true så ska y.equals(x) returnera true (symmetri).
- Om x.equals(y) returnerar true och y.equals(z) returnerar true så ska x.equals(z) returnera true (transitivitet).
- Upprepade anrop av x.equals(y) ska ge samma resultat (konsistens).
- x.equals(null) ska returnera false.

Skugga equals – med instanceof

```
public boolean equals(Object obj) {
    if (obj instanceof Person) {
        return id == ((Person) obj).id;
    } else {
        return false;
    }
}
```

- Observera att parametern till equals måste vara av typ Object, annars blir det inte skuggning. Därför måste också typomvandling till Person ske när man ska använda obj:s id.
- Uttrycket obj instanceof Person returnerar true om obj:s typ är Person eller någon subklass till Person.
- Uttrycket obj instanceof Person returnerar false om obj har värdet null.

Skugga equals med instanceof – för- och nackdelar

Överkurs

- Denna lösningen tillåter att subklasser ärver equals-metoden.
- Man kan därför använda equals i en arvshiearki och jämföra "subklassobjekt" och "superklassobjekt".
- Kan leda till att equals inte uppfyller kraven i specifikationen om man skuggar equals i subklassen.
  - Därför är det lämpligt att deklarera metoden equals final:

    ```
    public final boolean equals(Object obj) {
    ...
    ```

    Nu kan inte equals skuggas i någon subklass till Person.
  - Detta undviks om man bara tillåter jämförelser mellan objekt av samma typ. Se nästa bild.

Skugga equals – med getClass

Överkurs

```
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (this.getClass() != obj.getClass()) {
        return false;
    }
    return id == ((Person) obj).id;
}
```

- Metoden getClass returnerar typen för det objekt obj refererar till under exekveringen.
- Bara metoder av exakt samma klass kan anses vara lika.

Skugga equals – med extra metod canEqual

Överkurs

- Ibland blir lösningen på förra bilden för sträng. Det kan man lösa genom att lägga till en extra metod i klassen:

```
public boolean equals(Object obj) {
    if (obj instanceof Person) {
        Person other = (Person) obj;
            return other.canEqual(this) && this.id == other.i
    } else {
        return false;
    }
}

public boolean canEqual(Object other) {
    return (other instanceof Person);
}
```

- Bägge metoderna ska skuggas i subklasser. För detaljer läs mer här: www.artima.com/lejava/articles/equality.html

Skugga hashCode

- När man skuggar equals bör man också skugga metoden hashCode. Metoderna equals och hashCode används när objekt sätts in i en hashtabell. (Behandlas senare i kursen).
- Metoden ska returnera ett heltal som representerar objektet. Exempel:

```
public class Person {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Person) {
            return id == ((Person) obj).id;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return id;
    }
}
```


Skugga toString

- I klassen Object finns också metoden toString() som bör skuggas.
- Metoden ska returnera en sträng som representerar objektet:

```
public class Person {
    private String name;
    private int id;
    ...

    public String toString() {
        return name + " " + id;
    }
}
```

- Metoden toString anropas bl.a. vid utskrift av objekt:

```
System.out.println(p);
```


Javas klasser skuggar equals

- I Javas färdiga klasser skuggas equals, hashCode och toString.
- Man behöver inte själv tänka på att skugga equals om man använder någon av Javas klasser som typargument:

```
List<String> names = new ArrayList<String>();
list.names("Kili"));
list.names("Balin");

// sök efter platsen för namnet "Balin"
int index = list.indexOf("Balin");   // index får värdet 1
```

Exempel på vad du ska kunna

- Ha kännedom om Java Collection Framework
- Förklara begreppet generik, kunna använda respektive implementera generiska klasser
- Förklara begreppen wrapperklass, autoboxing och unboxing
- Använda en iterator för att traversera en samling element
- Skugga metoden `equals`