

Metoder som alla objekt ärver

- Eftersom alla objekt är direkt eller indirekt subklasser från klassen Object ärver de 11 metoder.

clone()

equals(Object obj)

finalize()

getClass()

hashCode()

notify()

notifyAll()

toString()

wait()

wait(long timeout)

wait(long timeout, int nanos)

Metoden toString()

- Representerar en "textbeskrivning" av objektet

//Metod i klassen Object

```
public String toString()  
{  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

- API – om toString(): "Det rekommenderas att alla subklasser åsidosätter (override) denna metod"

```
public final class Dog  
{  
    private final String name;  
  
    public Dog(String name)  
    {  
        this.name = name;  
    }  
  
    @Override  
    public String toString()  
    {  
        return "Name: " + this.name;  
    }  
}
```

Metoden equals()

- Används för att se om två objekt är logiskt lika
- Grundutförandet kontrollerar bara om två objekt-referenser refererar till samma objekt

```
public boolean equals(Object obj)  
{  
    return (this == obj);  
}
```

- Brukligt är att åsidosätta (override) denna metod

```
public boolean equals(Object obj)  
{  
    //Obs! Kontrollera att det är en CartItem först  
    CartItem cartItem = (CartItem) obj;  
    return this.getID() == cartItem.getID();  
}
```

Förklaring:

Objekten är logiskt lika, har samma ID. Kan referera till två olika objekt

Abstrakta klasser

- Definierar en klass som i regel är så generell att den är gjord för att subclassas
- Definieras med hjälp av nyckelordet **abstract**
- Både metoder och klasser kan definieras som abstract
- Kan inte vara final
- Kan ha både abstrakta och icke abstrakta metoder
- Kan ha konstruktorer även om det inte går att skapa ett objekt av klassen med **new**
- Om en metod i en klass görs abstract måste också klassen vara abstract
- Kan vara abstract utan att ha någon abstract metod

Abstrakta klasser fortsättning

- Klasser som ärver från en abstrakt klass måste implementera eventuella abstrakta metoder **eller** deklarerar sig själv som **abstract**

```
//abstract superclass
public abstract class Animal
{
    public abstract void eat(); ← Abstrakt metod har ingen metodkropp
}
```

```
//subclass
public class Dog extends Animal
{
    public void eat()
    {
        System.out.println("Chew, chew"); ← Läger till metodskropp
    }
}
```

--- Eller ---

```
//subclass
public abstract class Dog extends Animal ← Deklarerar sig själv som abstract
{
}
```

Abstrakta klasser och subklasser

- En klass som är subklass till en klass som är abstract kan lagras i en objekt-referensvariabel av superklass-typ

```
//superclass
public abstract class Animal
{
    public abstract void eat();
}

//subclass
public class Dog extends Animal
{
    public void eat()
    {
        System.out.println("Chew, chew");
    }
}
```

Animal animal = new Dog(); ← **Lagra i en objekt-referensvariabel av superklass-typ**

Interface

- Kan ses som en 100% abstrakt klass
- Kan inte implementera några metoder
- Deklarerar beteende istället för att implementera funktionalitet
- Deklareras med nyckelordet **interface** istället för **class**
- Betyder alltså: interface definierar vad en klass kan göra inte hur den gör det

```
public interface Bounceable
{
    public void bounce();
}
```

- Klasser som vill vara "bounceable" implementerar interfacet **Bouncable** och tillhandahåller kod för metoden bounce()

Interface forts.

- Metoder i ett interface är **ALLTID** public abstract - behövs därför inte skrivas
- Datamedlemmar i ett interface är alltid public static final - behövs inte heller skrivas
- Kan inte vara final
- Kan bara ärva från andra interface
- Kan ärva från mer än ett interface
- Kan, precis som klasser, deklarereras som public eller default
- Metoder i ett interface kan vara static under Java 8

Vad gör interface för nytta?

- Kapslar in likheter mellan annars obesläktade klasser utan att påtvinga arv

- Exemplet **Bounceable**:

En fotboll och ett bildäck kan båda implementera **Bounceable** utan att ha något arvsförhållande

- Deklarerar metoder som en klass förväntas implementera
- Avslöjar ett objekts gränssnitt utan att avslöja vilken klass det är

Några skillnader mellan abstract och interface

- En klass kan implementera flera interface men bara ärva från en direkt abstrakt superklass
- Interface kan inte ha några implementerade metoder - abstrakta klasser kan ha både och (Detta gäller inte under Java 8 då interface kan ha en default implementation av en metod)
- Ett interface kan ärva från flera andra interface - abstrakta klasser kan bara ärva från en direkt superklass
- Metoder i ett interface kan bara var public - metoder i en abstrakt klass kan ha vilket åtkomstattribut som helst

Interface-implementation

- Exempel på en klass som implementerar ett interface:

```
//Interface
public interface Bounceable
{
    public void bounce();
}

//Implementerande klass
public class Football implements Bounceable
{
    public void bounce()
    {
        System.out.println("Bounce, bounce");
    }
}
```

Klassen **Football** garanterar att den har metoden **bounce()** men hur den implementationen ser ut vet bara klassen Football

Interface extends

- Ett interface kan ärva från ett eller flera andra interface:

```
//Ärver från ett interface
public interface InterfaceOne extends InterfaceTwo
{
    public void printMessage();
}
```

```
//Ärver från två interface
public interface InterfaceOne extends InterfaceTwo, InterfaceThree
{
    public void printMessage();
}
```

Notera: Ett interface kan bara ärva från andra interface **INTE** från andra klasser

Casting

- Används för att konvertera ett objekt av en viss klasstyp till en annan klasstyp i sin klasshierarki

```
//superklass  
public class Animal  
{  
}
```

```
//Subklass  
public class Dog extends Animal  
{  
}
```

Animal animal = new Dog(); ← Lagra ett Dog-objekt i en objekts-ref.variabel av superklasstyp

```
//Gör en casting  
Dog dog = (Dog) animal; ← Gör om animal till en Dog
```

instanceof

- Kontrollerar om ett objekt är av en viss klasstyp
- Kan **BARA** kontrollera objekt mot klasser som är i samma klasshierarki
- Kontrollerar om ett objekt implementerar ett visst interface (vilket som helst)

Exempel på instanceof

- För att kontrollera om ett objekt av klassen Dog är av superklasstypen Animal:

```
//Superklass  
public class Animal  
{  
}
```

```
//Subklass  
public class Dog extends Animal  
{  
}
```

```
//Testar instanceof  
Dog dog = new Dog();
```

```
boolean answer = dog instanceof Animal; ← Obs! Testar mot klassen Animal inte ett objekt av  
den klassen
```

```
answer ← Kommer att vara true
```

Vanligt fel med instanceof

- instanceof kan **BARA** kontrollera objekt mot klasser som är i samma klasshierarki:

```
public class ClassA
{
}
```

```
public class ClassB
{
}
```

```
//Testar instanceof
ClassA a = new ClassA();
```

```
boolean answer = a instanceof ClassB; ← Detta ger kompileringsfel eftersom ClassA och
                                         ClassB inte är i samma klasshierarki
```

Notera: Detta gäller **BARA** med instanceof mot en klass inte mot ett interface

Övning - instanceof

- Skapa en arvshierarki:

En **superklass**

En **subklass** till **superklassen**

Två **interface**

Låt **superklassen** implementera ett **interface** och **subklassen** det andra

Skapa objekt av båda klasserna

Testa nu **instanceof** på objekten mot **klasserna** och mot dina **interface**