DM503

Forelæsning 10

Indhold

- Abstrakte klasser og metoder
- Nedarvningsdetaljer
  - Polymorfisme
- Design med nedarvning for øje
- Generics

Abstrakte klasser

Abstrakte klasser

- En abstrakt klasse

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder
- En abstrakt metode

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder
- En abstrakt metode
  - En metode erklæret med `abstract` nøgleordet

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder
- En abstrakt metode
  - En metode erklæret med `abstract` nøgleordet
  - Erklæret uden implementering
    (uden tuborgklammer, med semicolon)

    ```
    public abstract void myfunc(...);
    ```

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder
- En abstrakt metode
  - En metode erklæret med `abstract` nøgleordet
  - Erklæret uden implementering
    (uden tuborgklammer, med semicolon)

    ```
    public abstract void myfunc(...);
    ```
- Hvis en klasse indeholder en abstrakt metode skal klassen også være abstrakt

Abstrakte klasser

- En abstrakt klasse
  - En almindelige klasse erklæret med `abstract` nøgleordet

    ```
    public abstract class MyClass {...}
    ```
  - Der kan ikke laves instanser af klassen
  - Klassen kan indeholde abstrakte metoder
- En abstrakt metode
  - En metode erklæret med `abstract` nøgleordet
  - Erklæret uden implementering
    (uden tuborgklammer, med semicolon)

    ```
    public abstract void myfunc(...);
    ```
- Hvis en klasse indeholder en abstrakt metode skal klassen også være abstrakt
  - (Det omvendte behøver ikke være tilfældet)

Abstrakte klasser

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig
  - Kan indeholde noget

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig
  - Kan indeholde noget
  - Men mangler andet

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig
  - Kan indeholde noget
  - Men mangler andet
  - Klasser der nedarver fra en abstrakt klasse kan så udfylde manglerne

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig
  - Kan indeholde noget
  - Men mangler andet
  - Klasser der nedarver fra en abstrakt klasse kan så udfylde manglerne
  - Oftest et abstrakt koncept (deraf navnet)

Abstrakte klasser

- Hvad kan vi nu bruge det til?!?
- Minder lidt om et interface
  - Kan dog indeholde ikke abstrakte metoder (altså metoder med en implementering)
  - Kan indeholde variable som ikke er konstanter
- Tænk på en klasse der ikke er helt færdig
  - Kan indeholde noget
  - Men mangler andet
  - Klasser der nedarver fra en abstrakt klasse kan så udfylde manglerne
  - Oftest et abstrakt koncept (deraf navnet)
- Eksempel...

Abstrakte klasser

```
public abstract class Shape {
    int x, y;

    public void setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void draw();

    public abstract void resize(int size);
}
```

Abstrakte klasser

```
public class Circle extends Shape {
    int diameter;

    public Circle(int diameter) {
        this.diameter = diameter;
        setPosition(0, 0);
    }

    public void draw() {
        // Do some drawing
    }

    public void resize(int size) {
        diameter = size;
        draw();
    }
}
```

Abstrakte klasser

```
public class Square extends Shape {
    int side;

    public Square(int side) {
        this.side = side;
        setPosition(0, 0);
    }

    public void draw() {
        // Do some drawing
    }

    public void resize(int size) {
        side = size;
        draw();
    }
}
```

Abstrakte klasser

- Abstrakte klasser (og metoder)
  - Mulighed for at angive delvist koncept
  - Kan udvides (læs: nedarves) til flere konkrete koncepter
  - Samler al den fælles funktionalitet i en fælles klasse (kodegenbrug)

Nedarvningsdetaljer

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse
- `obj instanseof Class`

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse
- `obj instanseof Class`
  - `true` hvis `obj` er en instans af klassen `Class`

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse
- `obj instanseof Class`
  - `true` hvis `obj` er en instans af klassen `Class`
  - `true` hvis `obj` er en instans af en underklasse til `Class`

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse
- `obj instanseof Class`
  - `true` hvis `obj` er en instans af klassen `Class`
  - `true` hvis `obj` er en instans af en underklasse til `Class`
  - `false` ellers

Nedarvningsdetaljer

- Det er muligt at undersøge om et objekt er en instans af en given klasse
- `obj instanseof Class`
  - `true` hvis `obj` er en instans af klassen `Class`
  - `true` hvis `obj` er en instans af en underklasse til `Class`
  - `false` ellers
- Eksempel...

Nedarvningsdetaljer

```
public class Car {
   // En bil
}
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
public class Main {
    public static void main(String[] args) {
        Audi bil1 = new Audi();
        Skoda bil2 = new Skoda();

        checkType(bil1);
    }
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
public class Main {
    public static void main(String[] args) {
        Audi bil1 = new Audi();
        Skoda bil2 = new Skoda();

        checkType(bil1);
    }

    public static void checkType(Car bil) {
        if (bil instanceof Car) {
            System.out.println("bil er en Car ");
        }
        if (bil instanceof Audi) {
            System.out.println("bil er en Audi");
        } else if (bil instanceof Skoda) {
            System.out.println("bil er en Skoda");
        }
    }
}
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
public class
     public s
          Audi
          Skod

          checkType(bil1);
     }

     public static void checkType(Car bil) {
          if (bil instanceof Car) {
               System.out.println("bil er en Car ");
          }
          if (bil instanceof Audi) {
               System.out.println("bil er en Audi");
          } else if (bil instanceof Skoda) {
               System.out.println("bil er en Skoda");
          }
     }
}
```

```
Terminal — bash — 66×5
MAC-SDU-00001:10 petersk$ javac Main.java
MAC-SDU-00001:10 petersk$ java Main
bil er en car
bil er en Audi
MAC-SDU-00001:10 petersk$
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
public class Main {
    public static void main(String[] args) {
        Audi bil1 = new Audi();
        Skoda bil2 = new Skoda();

        checkType(bil2);
    }

    public static void checkType(Car bil) {
        if (bil instanceof Car) {
            System.out.println("bil er en Car ");
        }
        if (bil instanceof Audi) {
            System.out.println("bil er en Audi");
        } else if (bil instanceof Skoda) {
            System.out.println("bil er en Skoda");
        }
    }
}
```

Nedarvningsdetaljer

```
public class Car {
   // En bil
}

public class Audi extends Car {
   // En Audi
}
public class Skoda extends Car {
   // En Skoda
}
public class
     public s
          Audi
          Skod

          checkType(bil2);
     }

     public static void checkType(Car bil) {
          if (bil instanceof Car) {
               System.out.println("bil er en Car ");
          }
          if (bil instanceof Audi) {
               System.out.println("bil er en Audi");
          } else if (bil instanceof Skoda) {
               System.out.println("bil er en Skoda");
          }
     }
}
```

```
Terminal — bash — 66×5
MAC-SDU-00001:10 petersk$ javac Main.java
MAC-SDU-00001:10 petersk$ java Main
bil er en car
bil er en Skoda
MAC-SDU-00001:10 petersk$
```

Nedarvningsdetaljer

- Forkert brug af `instanceof`

Nedarvningsdetaljer

- Forkert brug af `instanceof`

```
public class Car {
    private double mileage;

    public Car() {
        mileage = 0;
    }

    public void drive(double dist) {
        if (this instanceof Audi) {
            System.out.println("An Audi is driving: " + dist);
        } else if (this instanceof Skoda) {
            System.out.println("A Skoda is driving: " + dist);
        }
        mileage = mileage + dist;
    }
}
```

Nedarvningsdetaljer

- Korrekt (og uden brug af `instanceof`):

Nedarvningsdetaljer

- Korrekt (og uden brug af `instanceof`):

```
public class Car {
    private double mileage;

    public Car() {
        mileage = 0;
    }

    public void drive(double dist) {
        mileage = mileage + dist;
    }
}
```

Nedarvningsdetaljer

- Korrekt (og uden brug af `instanceof`):

```
public class Car {
    private double mileage;

    public Car() {
        mileage = 0;
    }

    public void drive(double dist) {
        mileage = mileage + dist;
    }
}
public class Audi extends Car {
    public void drive(double dist) {
        System.out.println("An Audi is driving: " + dist);
        super.drive(dist);
    }
}
```

Nedarvningsdetaljer

- Korrekt (og uden brug af `instanceof`):

```
public class Car {
    private double mileage;

    public Car() {
        mileage = 0;
    }

    public void drive(double dist) {
        mileage = mileage + dist;
    }
}
public class Audi extends Car {
    public void drive(double dist) {
        System.out.println("An Audi is driving: " + dist);
        super.drive(dist);
    }
}
public class Skoda extends Car {
    public void drive(double dist) {
        System.out.println("A Skoda is driving: " + dist);
        super.drive(dist);
    }
}
```

Nedarvningsdetaljer

Nedarvningsdetaljer

- Der gik faktisk mere for sig...

Nedarvningsdetaljer

- Der gik faktisk mere for sig...
- Var der nogen der bemærkede hvordan en Audi/Skoda kunne bruges som en Car?

Nedarvningsdetaljer

- Der gik faktisk mere for sig...
- Var der nogen der bemærkede hvordan en Audi/Skoda kunne bruges som en Car?
-

```
public class Main {
    public static void main(String[] args) {
        Audi bil1 = new Audi();
        Skoda bil2 = new Skoda();

        checkType(bil2);
    }

    public static void checkType(Car bil) {
        ...
    }
}
```

Nedarvningsdetaljer

- Der gik faktisk mere for sig...
- Var der nogen der bemærkede hvordan en Audi/Skoda kunne bruges som en Car?
-

```
public class Main {
    public static void main(String[] args) {
        Audi bil1 = new Audi();
        Skoda bil2 = new Skoda();

        checkType(bil2);
    }

    public static void checkType(Car bil) {
        ...
    }
}
```

- Husk at når Skoda nedarver fra Car, så er Skoda en Car!

Nedarvningsdetaljer

- Der gik faktisk mere for sig...
- Var der nogen der bemærkede hvordan en Audi/Skoda kunne bruges som en Car?
- ```
  public class Main {
      public static void main(String[] args) {
          Audi bil1 = new Audi();
          Skoda bil2 = new Skoda();

          checkType(bil2);
      }

      public static void checkType(Car bil) {
          ...
      }
  }
  ```
- Husk at når Skoda nedarver fra Car, så er Skoda en Car!
  - ```
    public class Main {
        public static void main(String[] args) {
            Skoda bil2 = new Skoda();
            Car bil;
            bil = bil2;
        }
    }
    ```

Nedarvningsdetaljer

Nedarvningsdetaljer

- Polymorfisme

Nedarvningsdetaljer

- Polymorfisme
  - Konceptet at en variabel kan refererer til objekter af flere forskellige typer

Nedarvningsdetaljer

- Polymorfisme
  - Konceptet at en variabel kan refererer til objekter af flere forskellige typer
- Polymorfisme ved nedarvning

Nedarvningsdetaljer

- Polymorfisme
  - Konceptet at en variabel kan refererer til objekter af flere forskellige typer
- Polymorfisme ved nedarvning
  - ```
    Car bil;
    bil = new Audi();
    bil = new Skoda();
    ```
    ```

Nedarvningsdetaljer

- Polymorfisme
  - Konceptet at en variabel kan refererer til objekter af flere forskellige typer
- Polymorfisme ved nedarvning
  - ```
    Car bil;
    bil = new Audi();
    bil = new Skoda();
    ```
- Polymorfisme ved brug af interfaces
    ```

Nedarvningsdetaljer

- Polymorfisme
  - Konceptet at en variabel kan refererer til objekter af flere forskellige typer
- Polymorfisme ved nedarvning
  - ```
    Car bil;
    bil = new Audi();
    bil = new Skoda();
    ```
- Polymorfisme ved brug af interfaces
  - ```
    Comparable c;
    c = new Integer(10);
    c = new Double(12.3);
    ```

Design med nedarvning for øje

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt
- Overskriv metoder i underklasser så det passer til klassen

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt
- Overskriv metoder i underklasser så det passer til klassen
- Tilføj nye variable til underklasser men overskriv (shadow) aldrig nedarvede variable

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt
- Overskriv metoder i underklasser så det passer til klassen
- Tilføj nye variable til underklasser men overskriv (shadow) aldrig nedarvede variable
- Lad hver klasse håndtere sine egne variable

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt
- Overskriv metoder i underklasser så det passer til klassen
- Tilføj nye variable til underklasser men overskriv (shadow) aldrig nedarvede variable
- Lad hver klasse håndtere sine egne variable
  - Brug `super` til at tilgå forældre-constructoren

Design med nedarvning for øje

- Nedarvning skal overholde et "er en"-forhold
  - Underklassen skal være en mere specifik version
- Design dit klassehierarki så det øger genbrug af kode
  - Også potentielt fremtidigt genbrug af kode
  - Put ens funktionalitet i en fælles forældreklasse
  - Skub ens funktionalitet så højt i klassehierarkiet som muligt
- Overskriv metoder i underklasser så det passer til klassen
- Tilføj nye variable til underklasser men overskriv (shadow) aldrig nedarvede variable
- Lad hver klasse håndtere sine egne variable
  - Brug `super` til at tilgå forældre-constructoren
  - Brug `super` til at tilgå metoder på forældren

Design med nedarvning for øje

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)
- Overskriv altid generelle metoder som `toString` og `equals`

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)
- Overskriv altid generelle metoder som `toString` og `equals`
- Brug abstrakte klasser til at beskrive klasser længere nede i hierarkiet

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)
- Overskriv altid generelle metoder som `toString` og `equals`
- Brug abstrakte klasser til at beskrive klasser længere nede i hierarkiet
- Brug `private`, `protected`, `public` på fornuftig vis

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)
- Overskriv altid generelle metoder som `toString` og `equals`
- Brug abstrakte klasser til at beskrive klasser længere nede i hierarkiet
- Brug `private`, `protected`, `public` på fornuftig vis
- Indkapsling af data

Design med nedarvning for øje

- Brug interfaces hvis en klasse skal have to eller flere roller (simuler multipel nedarvning)
- Overskriv altid generelle metoder som `toString` og `equals`
- Brug abstrakte klasser til at beskrive klasser længere nede i hierarkiet
- Brug `private`, `protected`, `public` på fornuftig vis
  - Indkapsling af data
  - Adgang for underklasser

Generics

Generics

- Lad os starte med et motiverende eksempel :-)

Generics

- Lad os starte med et motiverende eksempel :-)
- Vi vil lave en kasse som vi kan komme ting ned i og tage ud igen

Generics

- Lad os starte med et motiverende eksempel :-)
- Vi vil lave en kasse som vi kan komme ting ned i og tage ud igen
- Vi vil gerne have at kassen skal kun gemme hvad som helst

Generics

- Lad os starte med et motiverende eksempel :-)
- Vi vil lave en kasse som vi kan komme ting ned i og tage ud igen
- Vi vil gerne have at kassen skal kun gemme hvad som helst
- Hmm ... alle klasser nedarver fra `Object`

Generics

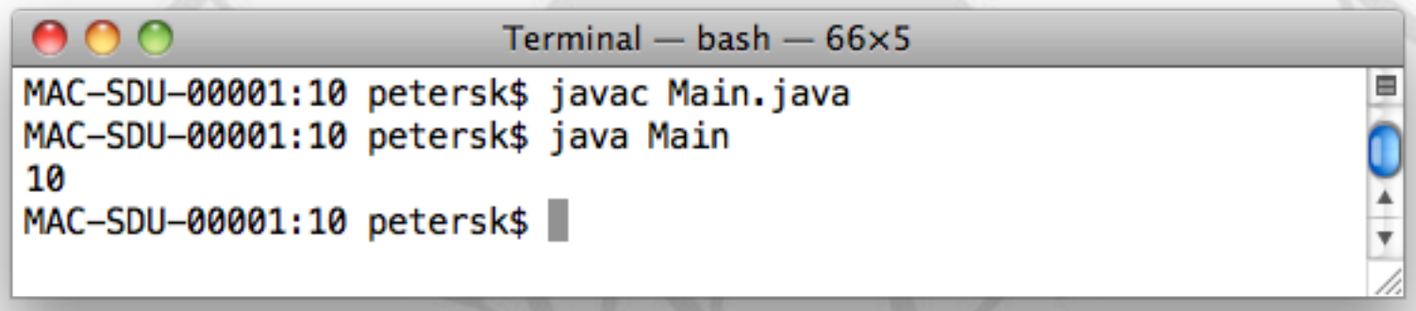
```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

Generics

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```



Generics

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large application
        integerBox.add("10"); // note how the type is now String

        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large application
        integerBox.add("10"); // note how the type is now String

        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```

```
Terminal — bash — 110×5
MAC-SDU-00001:10 petersk$ javac Main.java
MAC-SDU-00001:10 petersk$ java Main
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at Main.main(Main.java:9)
MAC-SDU-00001:10 petersk$
```

Generics

Generics

- Hvad er problemet?

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime
- Hvad gør vi så...

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime
- Hvad gør vi så...
- ... og ind kommer Generics som en redning

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime
- Hvad gør vi så...
- ... og ind kommer Generics som en redning
  - En måde at angive datatyper på i Java

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime
- Hvad gør vi så...
- ... og ind kommer Generics som en redning
  - En måde at angive datatyper på i Java
  - Typer kan også være variable i Java

Generics

- Hvad er problemet?
  - Vi har lavet en mundtlig "kontrakt" som ikke bliver overholdt
  - Java kan ikke tjekke den slags kontrakter for os
- Bemærk
  - Koden kompiler stadig korrekt
  - Fejlen sker først på runtime
- Hvad gør vi så...
- ... og ind kommer Generics som en redning
  - En måde at angive datatyper på i Java
  - Typer kan også være variable i Java
- Bedst illustreret med et eksempel...

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add(new Integer(10));

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add(new Integer(10));

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add(new Integer(10));

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

```
Terminal — bash — 110×5
MAC-SDU-00001:10 petersk$ javac Main.java
MAC-SDU-00001:10 petersk$ java Main
10
MAC-SDU-00001:10 petersk$
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add("10");

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add("10");

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

Generics

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();

        integerBox.add("10");

        Integer someInteger = integerBox.get(); // no cast!

        System.out.println(someInteger);
    }
}
```

```
Terminal — bash — 110×6
MAC-SDU-00001:10 petersk$ javac Main.java
Main.java:5: add(java.lang.Integer) in Box<java.lang.Integer> cannot be applied to (java.lang.String)
                integerBox.add("10");
                           ^
1 error
MAC-SDU-00001:10 petersk$
```

Generics

Generics

- Lidt notation

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang
- Eksempel: Box med to ting af samme type

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang
- Eksempel: Box med to ting af samme type
  - Forkert: `public class Box<T,T> {...}`

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang
- Eksempel: Box med to ting af samme type
  - Forkert: `public class Box<T,T> {...}`
  - Rigtig: `public class Box<T> {...}`

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang
- Eksempel: Box med to ting af samme type
  - Forkert: `public class Box<T,T> {...}`
  - Rigtig: `public class Box<T> {...}`
- Eksempel: Box med to ting af forskellig type

Generics

- Lidt notation
  - `Box` kaldes en parametriseret type (eller generisk type)
  - `T` kaldes en type-variabel
  - `<Integer>` kaldes et type-argument
- Bemærk
  - En generisk type kan have flere type-variable
  - Hver type-variabel kan kun specificeres én gang
- Eksempel: Box med to ting af samme type
  - Forkert: `public class Box<T,T> {...}`
  - Rigtig: `public class Box<T> {...}`
- Eksempel: Box med to ting af forskellig type
  - `public class Box<T,U> {...}`

Generics

Generics

- Generiske metoder

Generics

- Generiske metoder
  - Samme koncept bare kun for metoder

Generics

- Generiske metoder
  - Samme koncept bare kun for metoder
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U> void inspect(U u) {
        System.out.println("u: " + u); // toString på u
    }
}
```

Generics

- Generiske metoder
  - Samme koncept bare kun for metoder
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U> void inspect(U u) {
        System.out.println("u: " + u); // toString på u
    }
}
```

Generics

- Lidt mere realistisk eksempel

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for (Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.<Integer>fillBoxes(i, boxes);
    }
}
```

Generics

- Lidt mere realistisk eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
        for(Box<U> box : boxes) {
            box.add(u);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Box<Integer>> boxes = new ArrayList<Box<Integer>>();
        Box<Integer> b = new Box<Integer>();
        boxes.add(b);
        Integer i = new Integer(10);
        Box.fillBoxes(i, boxes);
    }
}
```

Generics

Generics

- Java kan selv se at typen `U` i `fillBoxes` skal være `Integer`

Generics

- Java kan selv se at typen `U` i `fillBoxes` skal være `Integer`
  - Kaldes type-inferens

Generics

- Java kan selv se at typen `U` i `fillBoxes` skal være `Integer`
  - Kaldes type-inferens
- Bemærk

Generics

- Java kan selv se at typen `U` i `fillBoxes` skal være `Integer`
  - Kaldes type-inferens
- Bemærk
  - T og U i har kunnet være alle typer

Generics

- Java kan selv se at typen `U` i `fillBoxes` skal være `Integer`
  - Kaldes type-inferens
- Bemærk
  - T og U i har kunnet være alle typer
  - Hvad hvis vi gerne vil bruge generiske typer men kun tillade nogle bestemte typer

Generics

Generics

- Metoder (og klasser) med begrænsede generiske typer

Generics

- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
```

Generics

- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
```

Generics

- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
```

- `<U extends Car>`
  - U kan være alle typer der er Car eller derunder i klassehierarkiet (Audi og Skoda)

Generics

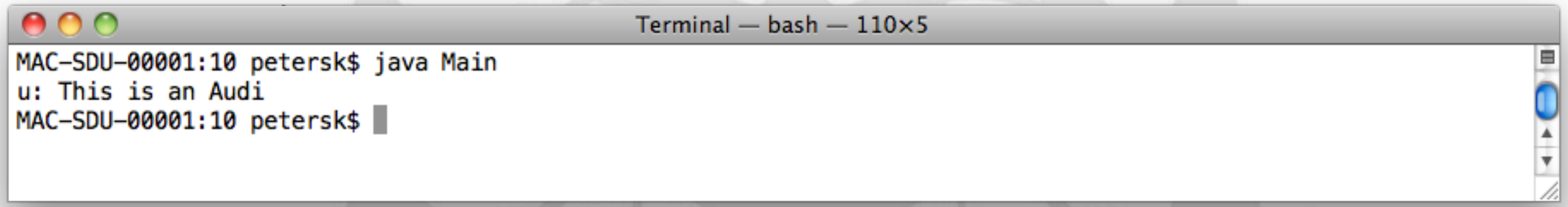
- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> b = new Box<Integer>();
        Audi bil = new Audi();
        b.inspect(bil);
    }
}
```

Generics

- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> b = new Box<Integer>();
        Audi bil = new Audi();
        b.inspect(bil);
    }
}
```



Generics

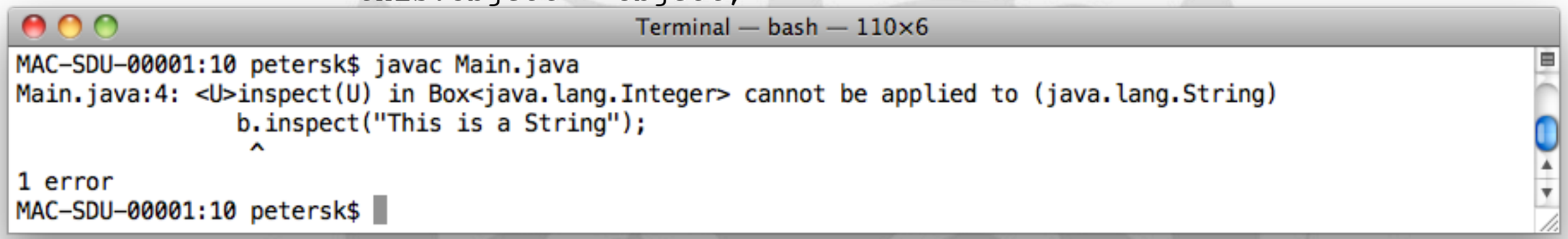
- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }

    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> b = new Box<Integer>();
        b.inspect("This is a String");
    }
}
```

Generics

- Metoder (og klasser) med begrænsede generiske typer
- Eksempel

```
public class Box<T> {
    private T object;

    public void add(T object) {
        this.object = object;
    }
    public <U extends Car> void inspect(U u) {
        System.out.println("u: " + u);
    }
}
public class Main {
    public static void main(String[] args) {
        Box<Integer> b = new Box<Integer>();
        b.inspect("This is a String");
    }
}
```



Generics

Generics

- `<U extends Something>`

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface
    - Betyder at `U` skal implementerer interfacet

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface
    - Betyder at `U` skal implementerer interfacet
    - Lille smule misbrug af nøgleordet `extends`

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface
    - Betyder at `U` skal implementerer interfacet
    - Lille smule misbrug af nøgleordet `extends`
- begrænsede generiske typer kan selvfølgelig også bruges på klasser

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface
    - Betyder at `U` skal implementerer interfacet
    - Lille smule misbrug af nøgleordet `extends`
- begrænsede generiske typer kan selvfølgelig også bruges på klasser
  - `public class Box<T extends Car> {...}`

Generics

- `<U extends Something>`
  - `Something` kan være en klasse (fx `Car`)
  - `Something` kan være et interface
    - Betyder at `U` skal implementerer interfacet
    - Lille smule misbrug af nøgleordet `extends`
- begrænsede generiske typer kan selvfølgelig også bruges på klasser
  - `public class Box<T extends Car> {...}`
- Hvad modellere følgende klasse (`MyClass`)?

```
public class MyClass<E extends Comparable<E>> implements List<E> {...}
```