Typer

DIT012/Joachim von Hacht

Varför Typer?

2

Typer används för att förhindra oss att använda data på ett felaktigt sätt, att blanda ihop saker och ting, att göra felaktiga operationer.

- Typer är till för att hjälpa oss!
- … men nybörjare tycker ofta de är i vägen …

Analogi: Det skall vara salt i saltkaret och peppar i pepparkaret! Inte salt i pepparkaret osv!!!

Statiskt Typsystem

```
// Compile (type) error
int r = 0.567;

// Compile (type) error
out.println(123 + true);
```

3

Java är ett statiskt typat språk

- Java har ett typsystem som skall eliminera typfel redan vid kompileringen (d.v.s. **statiskt**, innan vi kört programmet)
- I kompilatorn (javac) finns en "type checker" som sköter typkontrollen i samband med att koden kompileras.
  - Bryter vi mot reglerna för typer får vi ett kompileringsfel
  - IntelliJ kommer att visa ett felmeddelande, ett **typfel** (typsäkerhet)
- För att typsystemet skall fungera måste alla värden (uttryck) i Java ha en känd typ vid kompileringen d.v.s.
  - Alla värden måste tillhör en viss typ (int , double, o.s.v.)
  - Java vet t.ex. vilken typ av operander olika operatorer accepterar,... inbyggt i språket
  - Man kan se typer som olika mängder med tillhörande (tillåtna) operationer.

I bilden:

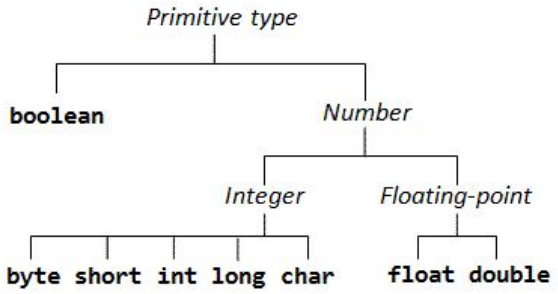
- Variabeln r kan bara lagra heltal. Vi försöker initiera med ett reellt tal (flyttal). Kompilatorn ger ett typfel eftersom vi tappar information (decimalerna)
- Värdet 123 kommer att ges typen int av typsystemet
- Värdet true kommer att ges typen boolean

- Operationen 123 + true är inte meningsfull!
    - Typsystemet vet att addition inte kan utföras med typen boolean.
- I detta exempel är det lätt att se felet, … men så är det inte alltid!

Ibland ges värden automatiskt en typ (t.ex. literaler), ibland måste vi ange typen

- När vi anger typ skriver vi helt enkelt int, boolean, o.s.v. i koden
    - …, t.ex. vid deklarationer.
- Mer om typer här.

Primitiva Typer



5

Primitiva typer bestämmer, förutom tillåtna operationer, dessutom:

- Hur mycket minne som behövs för värdet (variabeln) samt min och maxvärden
    - Om min/max-värdet under/överskrids, kan det leda till felaktiga resultat. Kallas underflow respektive overflow.
- Hur bitarna i minnet skall tolkas (t.ex. heltal eller potensform, förenklat)

Bilden visar de 8 primitiva (inbyggda) typerna i Java. Dessa finns färdiga i språket och kan inte ändras (eller lägga till nya).

- **boolean**, är typen för sanningsvärden
    - Finns bara två värden i typen: true och false (båda värdena är reserverade ord)
- **int**, är typen för heltal (integer)
    - Heltalsliteraler ges automatisk denna typ
    - -2147483648 till 2147483647 (32 bitar)
- **char**, är typen för enstaka tecken (character). Egentligen en teckenkod, därför räknas den som ett heltal.
    - Alla teckenliteraler ges denna typ
    - 0 till 65,535 (16 bitar).
- **double**, är typen för reella närmevärden (double precision, ca 15 decimaler)

        - Flyttalsliteraler får denna typ
        - 4.94065645841246544e-324d till 1.79769313486231570e+308d (64 bitar)
- byte, short, long och float hoppar vi över så länge (kommer troligen inte att användas i kursen)

Namn på primitiva typer är reserverade ord

Typkompabilitet

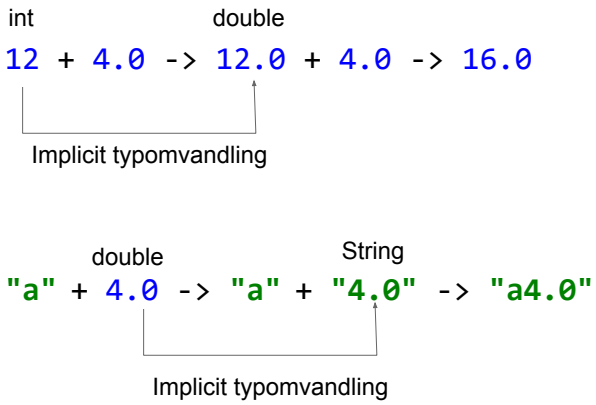
```
double d = 123;    // OK

out.println(d + 45); //Ok
```

5

Typkompabilitet bestämmer när det går bra att använda en typ istället för en annan

- Finns många inbyggda regler för detta i Java
- För primitiva typer gäller att det går bra att använd en “större” typ, för ett värde av “mindre” typ, t.ex. kan man tilldela en int till en double (den får plats, m.m.)
- För vissa operatorer innebär det ingen risk för typfel att tillåta olika typer för operanderna.
    - T.ex. addition med int och double.

Implicita typomvandlingar



6

Implicita typomvandlingar innebär att:

- Ett värde automatiskt omvandlas till en kompatibel typ för att man skall kunna utföra en viss operation.
- Värden byter alltså automatiskt typ (variabler byter aldrig typ)!
    - För primitiva typer betyder det att bitarna i värdet ändras.

Omvandling till strängar

- Då + operatorn har minst en operand av typen String omvandlas den andra operanden till String, därefter sker konkatenering (sammanslagning).

Explicit Typomvandling för Primitiva Typer

```
int width = 200;
double scale = width / 2 / PI;      //Built-in PI (double)
int x = width / 2 + (int) (scale * i); // Explicit cast

int a = …;
int b = …;
double d = (double) a / b;   //Explicit cast, fix real
                             //division
```

7

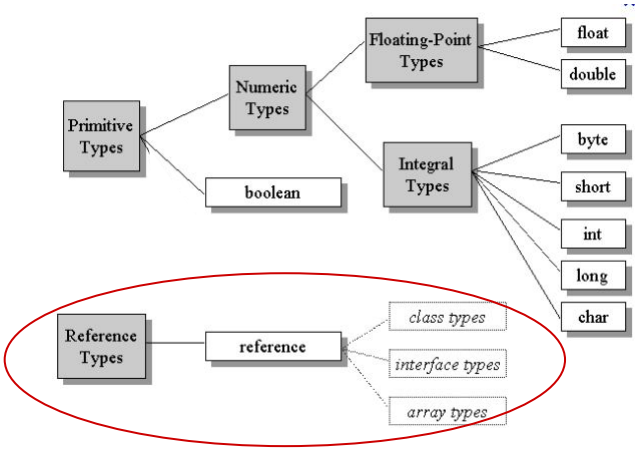
Ibland måste vi uttryckligen säga åt typsystemet att vi vill omvandla ett värde av en typ tillen annan.

- Kallas **explicit typomvandling (typecasting, cast)**
- Parenteser kan behövas för att visa vad (vilket uttryck) som berörs
- Explicit typomvandling har högre prioritet än samtliga aritmetiska operatorer

Omvandling primitiva typer

- Om vi vill omvandla ett double-värde till ett int-värde skriver vi (int) framför
  - Innebär att vi kapar alla decimaler (ingen avrundning)
  - Vi tappar information
  - Typsystemet satt ur spel vi får själva ansvara för följderna
    - Undvik om möjligt!
- Om vi vill omvandla en int till en double skriver vi (double) framför
  - Ingen information försvinner
  - I bilden gör vi det för att undvika heltalsdivision.

Referenstyper



8

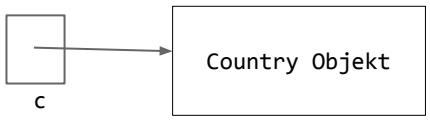
Referenstyper kan vara array-typer (t.ex. int[]), klasstyper (t.ex. String) eller gränssnittstyper (interface)

Man kan skapa nya referenstyper!

- Typsystemet är utbyggbart för array-, klass- och gränssnittstyper
  - Kallas också **egendefinierade typer** (vi definierar dem)
  - Genom att deklarera arrayer skapas nya typer utifrån en grundtyp.
  - Genom att deklarera klasser och gränssnitt skapas nya klass- respektive gränssnittstyper.

Klasstyper

```
// Variable with class type
Country c = new Country();
```



9

<u>En klass introducerar en ny referenstyp</u>

- Om vi skapar en ny klass kan vi deklarera variabler av denna typ!
- Vi kan deklarera en referensvariabel, c, av typen Country!
  - c kan bara referera Country-objekt (eller kompatibla objekt)!

Instansieringsuttrycket till höger returnerar en referens till ett Country-objekt …

  - Värdet är alltså en referens …
  - .. sidoeffekten är att ett (namnlöst) objekt skapas.
- Eftersom typ på värde (= referens till Country) och variabel (en referensvariabel för Country) är kompatibla fungerar tilldelning.
- <u>Ofta använder vi klass och typ som synonymer!</u>
  - Eftersom en klass introducerar en typ

Uppräkningstyper

```
// Enumeration type
enum WeekDay {
   MON, TUE, WED, THU, FRI, SAT, SUN
}

WeekDay d1 = WeekDay.FRI;
WeekDay d2 = WeekDay.THU;
//WeekDay d3 = "SUN";  // No “SUN” is a string, wrong type
WeekDay d4 = WeekDay.FRI;

out.println(d1 != d2); // == Work!
out.println(d1 == d4);
```

12

Ibland behövs ett begränsat antal värden av en viss typ (typen innehåller ett litet antal värden)

- T.ex. veckodagar, färger , etc
- VI skulle kunna använda String för dessa t.ex. “Mon”, “Tue” osv. men … (eller integer med Månd = 1, o.s.v.)
  - … detta blir inte typsäkert …
  - t.ex. om vi stavar fel, t.ex. “Tui”, så släpper kompilatorn igenom felet (felet kommer senare under körning)
  - Om vi använder int för dagar så kan någon av misstag tilldela en dag värdet -12, o.s.v. …
- Bättre att låta typsystemet se till att allt stämmer

Genom att deklarera en **uppräkningstyp** (enumeration**)** skapar vi en ny (egen) klassreferenstyp med ett antal (uppräknade) värden (en enum är en slags klass).

- Vi kan därmed deklarera variabler av denna typ.
- Kompilatorn kan kontrollera att vi bara använder korrekta värden!

Deklarationen av uppräkningstyp görs men det reserverade order **enum**.

- De värden som tillhör typen räknas upp (vi skriver namnen på värdena)
  - I bilden: MON, TUE, … o. s.v. (stora bokstäver skall användas)

    - Värdena är av typen WeekDay (INTE String, inga citattecken runt)
    - Det finns exakt ett (icke-ändringsbart) objekt för varje namn vi räknar upp.
        - Likhet (==) fungerar därför mellan värden (eftersom det är identitet i detta fall)
- För att komma åt värdena måste vi använda **punktnotation** d.v.s. typnamn.värde
    - Går att förenkla genom att använda import static …. Weekday.* (som med System)

Enum och Typomvandlingar

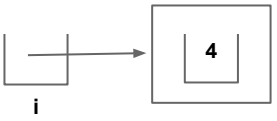
```
// Enumeration type
public enum WeekDay {
   MON, TUE, WED, THU, FRI, SAT, SUN
}

// Loop through all days
WeekDay[] days = WeekDay.values();
for (int i = 0; i < days.length ; i++) {
   out.println(w);    // Will print “MON”, “TUE”, ...
}

// Convert String to WeekDay
WeekDay d = WeekDay.valueOf("SUN");
```


Finns fördefinierat hur man omvandlar mellan enum-typer och String.

- Vid konvertering från sträng måste strängen vara exakt samma som namnet på värdet.

Omslagstyper

```
// Wrapper type Integer
Integer i = 4;   // Boxing


int j = i;       // Unboxing
if( i == j ){  // Ok, unboxing
 …
}

Double d  = 5.32;   // Boxing
Character ch  = 'X';
```



12

Omslagstyper är (färdiga) referenstypsversioner av primitiva typer

- De paketerar in ett primitivt värde i ett objekt, t.ex. int packas in i en Integer-objekt
  - Vi behöver omslagstyper t.ex. då vi använder samlingar (de kan bara lagra referenstyper), se Samlingar.
- Omvandling mellan omslagstyp och primitiv typ sköts automatiskt **(boxing/unboxing)**
  - Likhet kan använda ==
- Objekt av omslagtyper kan inte ändras (icke muterbara)
- Omslagstyper är klasstyper

Det finns omslagstyper för alla primitiva typer, några är …

- Double och Character.

Typomvandling med String

```
// From primitive to String (objects have toString())
String s = String.valueOf(45);   // Class methods
s = String.valueOf(true);
s = String.valueOf(1.45);
s = String.valueOf('X');

// From String to primitive (using wrapper types)
int i = Integer.valueOf("678");
double d = Double.valueOf("4.57");
boolean b = Boolean.valueOf("false");
```

Typomvandling till/från referenstypen String är lite speciell

- Vi omvandlar normalt från/till primitiva typer.
  - Kan omvandla till omslagstyper också (referenstyper).
- Omvandlingarna görs m.h.a. String-klassen och klassmetoder i omslagstyperna

Arrayer och Typomvandling

```
Integer i = 4; // Ok! Boxing

int[] iArr = { 1, 2, 3 };

// Error, no implicit casting (of elements)
double[] dArr = iArr;

// Error, no explicit casting
dArr = (double[]) iArr;

// Error! no implicit casting to reference
// elements
Integer[] iiArr = iArr;
```


Finns vissa komplicerade regler för typomvandling av arrayer, går inte in på detta

- .. .vi konstaterar att det i bilden inte fungerar.
- Vill vi typomvandla får vi gå igenom element för element och omvandla.

Mer om Typkompabilitet (1)

```
// Implicit type conversion to compatible type
// Possible to convert the bits in 4 (32 bits) to double
// (64 bits)
double d = 4;

// Now, any operation for type double allowed
d + 1;     // Ok
d % 3;     // Ok
...

d & 0x000F;  // NO! (bitmask only allowed for int's)
              // d declared as double!
```

Typkompabilitet för primitiva typer betyder:

- att bitarna i värdet kan omvandlas till den kompatibla typen.
- därefter utgår kompilatorn från den statiska (deklarerade) typen för att bestämma vilka operationer man i fortsättningen kan göra på (det omvandlade) värdet.

Mer om Typkompabilitet (2)

```
// Conversion reference (class) types
Car c = new Dog()       // Probably not compatible
Animal a = new Dog();   // Possibly ok?
Animal a = getAnimal();
Dog d = (Dog) a;        // Is "a" really a dog???


String s = new int[3];  // NO!
s = null;               // Hmm, yes, valid but
                        //operations fail (exception)
```

Typkompabilitet för referenstyper innebär:

- Inte att bitarna i värdet förändras, alla referenser har 64 (32) bitar. De kopieras rakt av vid tilldelning,

Normal är olika referenstyper inte kompatibla, de ju referera olika sorters objekt med olika operationer (metoder). Det man kan göra med en Car är troligtvis inte samma som med en Dog.

- Man kan göra explicita typomvandlingar, men mycket farligt … får ev. fel senare under körning!

Specialfall (undantag)

- Alla referenstyper är kompatibla med typen Null (med enda värdet null)
  - Kan inte deklarera variabler med typ Null …
- Om vi utför tillåtna operationer (för den deklarerade) typen, så undantag.
  - Formellt får vi inget typfel, vi får ett undantag.

Statisk och Dynamisk Typ

```
// Yes, Object type compatible with *any* reference type
// (no change of "fia", it's still a string object)
Object o = "fia";

o.lenght();          // Bad! No length method in object

// Declared type object, there is a method equals in
// Object so ok, … **BUT** method in String executed!!!
// (String overrides Object's equals)
o.equals("pelle");
```

Variable/static type (Object) = there is a method!
Object/dynamic type (String) = which method to run!

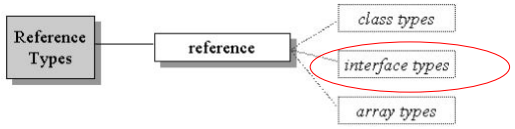
Huruvida typen för referensen till objektet (typen för objektet säger vi) och typen för variabeln är kompatibla bestämmas av ett stor antal komplicerade regler.

- Typen på variabeln och typen på objektet kan alltså var olika!
- Typen för variabeln (statiska, deklarerade) typen bestämmer vilka operationer över huvud taget kan göra (samma som för primitiva typer). Kontrolleras vid kompileringen.
- Typen på objektet (objekttypen, dynamiska typen) bestämmer vilken "version" av operationen som skall användas då programmet körs (ev. är metoden överskuggad)
    - Kallas också **sen bindning (lat/dynamic binding)**

Några grundläggande regler för referenstypskompabilitet:

- Alla referenstyper är kompatibla med typen Object
    - Därför att alla klasser och arrayer ärver metoderna från Object.
- Metoderna som typen för variabeln (Object) tillåter finns i objektet som referensen pekar på.

Gränssnittstyper




```
public interface List<E> ... {
    boolean isEmpty();
    boolean add(E e);
    E get(int index);
    ...
}

// Variable with interface type
private List<Integer> list = ... ;
```

Ett **gränssnitt** (interface) är en samling av metodhuvuden

- Kallas att metoderna är **abstrakta (abstract method)**
  - Eftersom det inte finns någon körbar kod
- Kan inte instansiera ett gränssnitt (med new)
- Vi säger att: Ett gränssnitt är ett kontrakt (eller en garanti)
  - Kontraktet anger vad man garanterat kan göra med ett objekt som uppfyller det.
    - Dvs vilka metoder man kan anropa på objektet.

I bilden deklareras ett gränssnitt List med ett antal metoder.

- Kontraktet anger vad som skall uppfyllas av en "lista", vad man kan göra med den.
- Vad listan innehåller är irrelevant. Gränssnittet List beskriver bara listan, Inte objekten i listan… objekten kan vara vilken referenstyp som helst.
  - Detta anges med en typparametern <E> på samma sätt som för en generisk metod.
    - Interface måste inte ha typparametrar, i detta fall har vi en typparameter

Ett gränssnitt introducerar en referenstyp

- Innebär att vi kan deklarera en gränssnittsstyp för en

- referensvariabel
- I samband med deklarationen anges vilken typ av objekt listan skall innehålla.
- List<Integer>, en lista med heltal eller List<String>.
    - Variabeln list i bilden refererar något objekt som garanterat har metoderna isEmpty(), add(), get(), m.fl.

Implementera ett Gränssnitt

```
public class MyList<E> ... implements List<E> {
    @Override
    boolean isEmpty(){ ... }     // Method bodies here
    @Override
    boolean add(E e) { ... }
    @Override
    E get(int index){ ... }
    ...

}
```

Hur skapar man ett objekt som uppfyller kontraktet för ett visst interface?

- Man låter objektets klass **implementera (implements)** gränssnittet.
  - Görs genom att skriva implements + gränssnittets namn i klassdeklarationen och …
  - …implementera metoder med med exakt samma metodhuvuden som angivits i gränssnittet
  - För säkerhets skull anger vi @Override i klassen, då kontrollerar kompilatorn att metodhuvuden matchar..
- Vi använder inga egna gränssnitt i denna kurs
  - Däremot förekommer de i samband med Samlingar

Gränssnitt och Kompabilitet

```
// Compatible because MyList implements List
// Declaration and instantiation
private List<Integer> list = new MyList<>();

if (list.isEmpty()) {   // Will work

}
```

En variabel av gränssnittstyp är alltid kompatibel med ett objekt av typen (klassen) som implementerar gränssnittet.

- I och med att typen för objektet implementerar gränssnittet så uppfyller objektet kontraktet
    - Detta kontrolleras av kompilatorn, anger vi implements … för klassen, så måste alla metoder finnas.