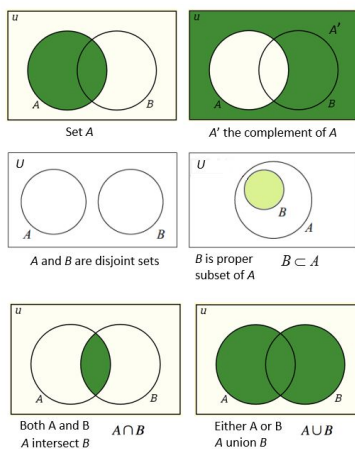


Typen

TDA548/Joachim von Hacht

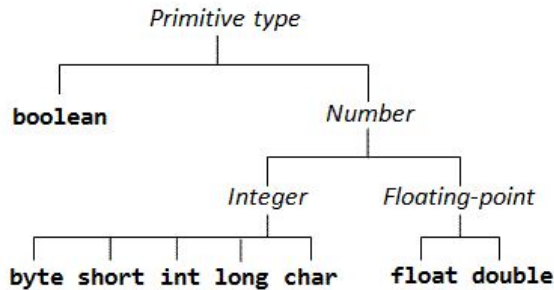
Vad är en Typ?



Finns inget enkelt svar.

- I denna kurs ser vi en typ som en mängd
- T.ex. mängden av reella tal, mängden av sanningsvärden o.s.v.

Primitiva Typer



3

Bilden visar de 8 [primitiva 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
- int, är typen för heltal (integer)
 - Heltalslitteraler ges automatisk denna typ
- char, är typen för enstaka tecken (character). Egentligen en teckenkod, därför räknas den som ett heltal.
 - Alla teckenlitteraler ges denna typ
- double, är typen för reella närmevärden, kallas också flyttal (double precision, ca 15 decimaler)
 - Flyttalslitteraler får denna typ
- long, används för stora heltal
- byte, short, och float hoppar vi över så länge (behövs inte i kursen)

Namn på primitiva typer är reserverade ord

Varför Typer?



4

Typer används för att förhindra oss att använda data på ett felaktigt sätt, att blanda ihop saker och ting, att göra felaktiga (meningslösa) 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 (och inget annat)!

Typsystem

$$\begin{array}{c}
 \frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_S x : \tau} \quad [\text{Var}] \\
 \\
 \frac{\Gamma \vdash_S e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash_S e_1 : \tau}{\Gamma \vdash_S e_0 e_1 : \tau'} \quad [\text{App}] \\
 \\
 \frac{\Gamma, x : \tau \vdash_S e : \tau'}{\Gamma \vdash_S \lambda x. e : \tau \rightarrow \tau'} \quad [\text{Abs}] \\
 \\
 \frac{\Gamma \vdash_S e_0 : \tau \quad \Gamma, x : \bar{\Gamma}(\tau) \vdash_S e_1 : \tau'}{\Gamma \vdash_S \text{let } x = e_0 \text{ in } e_1 : \tau'} \quad [\text{Let}]
 \end{array}$$

Exempel på "regler" för ett typsystem (inte Java's typsystem).

5

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 (det finns inbyggda regler).
 - Bryter vi mot reglerna för typer får vi ett kompileringsfel, ett typfel, från kompilatorn (typsäkerhet)
 - IntelliJ kommer att visa ett felmeddelande.
 - Igen: Det skall inte kunna bli några typfel då vi kör programmet!
- För att typsystemet skall fungera måste olika delar av programmet ha en känd typ vid kompileringen.
 - Literaler ges automatiskt en typ, 123 får t.ex. typen int och true får typen boolean.
 - Uttryck kommer automatisk att ges en "beräknad" typ utifrån de ingående delarnas typer (literaler, variabler, operatorer, ...)
 - För variabler, arrayer och metoder måste vi explicit ange typen vid deklarationen, vi skriver helt enkelt int, boolean, Dog eller Player o.s.v. i koden.

En fördel med ett statiskt typsystem är också att typerna fungerar som en

dokumentation, man ser tydligare vad man arbetar med och ev. hur det är tänkt att användas.

Typer och Operationer

```
out.println(false > 1); // Senseless
out.println(!45);       // Senseless

int i = 1;
double d = 1.0;

// Many operators applies for both int and double ...
out.println( i + 2 * i + d / 2.0 );

// .. but not all
out.println( ~i ); // Bitwise flip for int, ok!
out.println( ~d ); // Bitwise flip for double, NO!
```

6

Det går inte att utföra alla operationer med alla typer!

- De olika typerna har olika uppsättningar av tillåtna operationer.
 - Eller ekvivalent: Operationerna kräver att operanderna har vissa typer
- Vilka operationer som är tillåtna för respektive primitiv typ är inbyggt i Java
 - Kompilatorn vet!

Subtyp

```
// Operations allowed with double's
out.println(4.0 + (5.0 * 3.0) > 6.0);

// No type error if replacing *all* with int's
out.println(4 + (5 * 3) > 6);

// Operations allowed on int's
out.println(5 * ~3 / (4 - 1));

// Type error if replacing *all* with doubles.
out.println( 5.0 * ~3.0 / (4.0 - 1.0));
```

8

Om vi överallt i ett uttryck kan ersätta ett värde av typ T med en annat värde av typ S utan att något typfel uppstår säger vi att S är en **subtyp** till T, skrivs $S <: T$

- I bilden: Vi kan var som helst i ett uttryck där ett double-värde används ersätta detta med ett int-värde utan att typfel uppstår, d.v.s. $\text{int} <: \text{double}$ (int är en subtyp till double)
- Däremot kan vi inte ersätta int-värden med double-värde eftersom vissa operationer inte är tillåtna, vi får ett typfel d.v.s. double är inte en subtyp till int, double är en **supertyp** till int.
 - Man kan alltså göra fler operationer med en subtyp än en supertyp.
- Super/subtyp relationen ($<:$) är en binär relation mellan typer som säger att en typ (subtypen), i alla sammanhang, kan ersätta en annan typ (supertypen).
- För värden av primitiva typer gäller bl.a.
 - $\text{char} <: \text{int}$,
 - $\text{int} <: \text{float}$
 - $\text{float} <: \text{double}$
- Super/subtyp relationen är transitiv
 - D.v.s. $\text{char} <: \text{int} <: \text{double}$, ger att $\text{char} <: \text{double}$

Genom att använda subtyper kan vi skriva koden på ett mer abstrakt sätt.

- Vi kommer att se många exempel senare

Subtyper vid Tilldelning

```
double d; int i;

// Right side is an expression
d = d + (d * d) / d - d;
// Replace with subtype right ok, int <: double
d = i + (i * i) / i - i;

// Can't replace left side, not an expression!
i = d + (d * d) / d - d; // Assignment not accepted
out.println(~i); // If allowed, use of ~ for double value

// Left side a location
i = i + (i * i) / i - i;
// Replace with supertype left side OK!
d = i + (i * i) / i - i;
out.println(~d); // Operation not accepted by compiler
```

8

För tilldelning gäller

- Högersidan är ett uttryck (värdet), ok att ersätta supertyp med subtyp enligt tidigare resonemang.
- Vänstersidan representerar inte ett värde, den representerar en plats.
 - Vi kan inte ersätta variabel till vänster med en variabel av någon subtyp!
 - Eftersom vi då skulle kunna göra otillåtna operationer med värdet (det finns ju fler operationer för subtypen)
 - Däremot kan den ersättas med en variabel av supertypen!
 - Eftersom det finns färre operationer för supertypen (en begränsning av vad vi kan göra med värdet).


Man kan se det som: Mängden av operationer för supertypen är en delmängd av operationerna för subtypen

- D.v.s. på högersidan ersätter vi med något som kan mer.
- På vänstersidan ersätter vi med något som kan mindre.

Implicita typomvandlingar för Primitiva Typer

int double

12 + 4.0 -> 12.0 + 4.0 -> 16.0



Implicit typomvandling (int <: double)

```
// 1 converted to 1.0  
double d = 1;
```

9

Implicita typomvandlingar innebär en automatisk typomvandling

- Sker alltid från från subtyp till supertyp
- Används för att göra det möjligt att blanda olika typer i t.ex. aritmetiska uttryck.
- För primitiva typer betyder det dessutom att bitarna i den binära representationen av värdet ändras.
- Värden byter alltså typ (variabler byter aldrig typ)!
- Om det kan ske en implicit omvandling säger vi att typerna är kompatibla

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

double n = (double) true;    // No super/sub relation!
```

10

Vi kan uttryckligen säga åt typsystemet att vi vill omvandla ett värde av en typ till en annan.

- Kallas **explicit typomvandling (typecasting, cast)**
- Kan bara ske om det finns en super/subtyp relation.
- Parenteser kan behövas för att visa vad (vilket uttryck) som berörs
- Explicit typomvandling har högre prioritet än samtliga aritmetiska operatörer.
- Om vi gör en explicit omvandling får vi själva ta ansvaret för ev. typfel, undvik!

Omvandling primitiva typer

- Om vi vill omvandla ett double-värde till ett int-värde skriver vi (int) framför
 - Innebär dessutom att vi kapar alla decimaler (ingen avrundning)
- 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.

Primitiva typer: Min och maxvärden

```
// Overflow
int i = Integer.MAX_VALUE;
out.println(i + 1);

// Underflow
int j = Integer.MIN_VALUE;
out.println(j - 1);

// True
out.println(Integer.MAX_VALUE + 1 == Integer.MIN_VALUE);
```

11

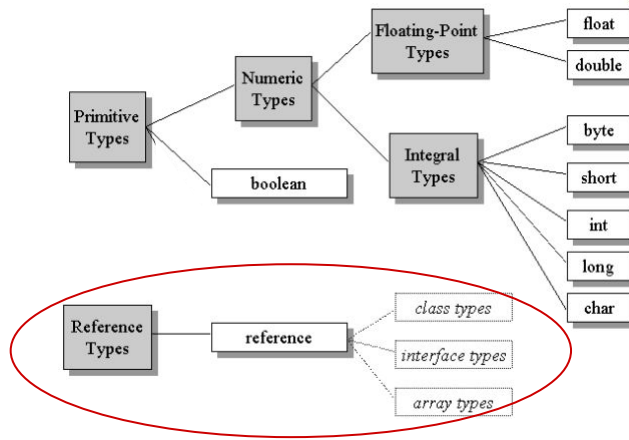
En mer implementationsmässig (teknisk) aspekt på primitiva typer är att de har olika storlekar i minnet och därmed min och maxvärden. För Java gäller:

- int (32 bitar): -2147483648 - 2147483647
- char (16 bitar): 0 - 65,535
- double (64 bitar): 4.94065645841246544e-324d - 1.79769313486231570e+308d

Över eller underskrider vi max eller min värdena för en primitiv typ får vi overflow eller underflow.

- Innebär att värdet "slår över" från max till min och tvärtom.

Referenstyper



12

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.

Referenstyper har alla samma min och maxvärden, de är alla lika stora

- 32 bitar för 32-bitars maskiner
- 64 bitar för 64-bitars maskiner
- Värdet (bitarna) i ett referensvärde ändras aldrig.

Klasstyper

// Class introduces new type

```
class Country {  
    ...  
}
```

// Variable with class type

```
Country c = new Country();
```

13

En klass introducerar en ny referenstyp

- Om vi skapar en ny klass kan vi deklarerar variabler av denna typ!
- Om vi skapar klassen Country, kan vi deklarerar en referensvariabel, c, av typen Country!
- Typen på värdet från uttrycket till höger (new-operatörn) är en referens till Country.

Ofta använder vi klass och typ som synonymer!

Subtyper för Klasstyper

```
class Player { ... }  
class Dog { ... }  
  
Player p = new Player();  
  
// No super/sub relation,  
// i.e. no implicit or explicit type casting  
Dog d = p; // No  
p = new Dog(); // No  
Dog d = (Dog) p; // No
```

14

Olika klasstyper har inte någon super/subtype relation

- Det finns ingen implicit eller explicit typomvandling mellan klasstyper.
- Det går att skapa en super/subtype relation , se vidare arv nedan.

Omslagstyper

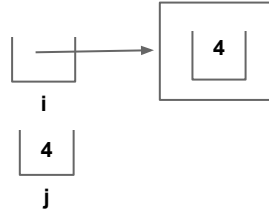
```
// Wrapper type Integer  
Integer i = 4; // Boxing
```

```
int j = i; // Unboxing
```

```
Double d1 = 5.32; // Boxing  
Character ch = 'X';
```

```
d1 = i; // No super/sub relation (class types)  
Double d2 = 4; // No, boxing goes to Integer
```

```
Number n = d1; // Ok! Number is super type to Double  
n = i; // Ok! Number is super type to Integer
```



15

Omslagstyper är referenstypsversioner av primitiva typer

- Finns färdiga i Java
- De paketerar in ett primitivt värde i ett objekt, t.ex. int packas in i en Integer-objekt
 - Omvandling mellan omslagstyp och primitiv typ sköts automatiskt (**boxing/unboxing**)
- Objekt av omslagstyper kan inte ändras (icke muterbara)
- Omslagstyper är klasstyper
 - D.v.s det finns ingen super/subtyp relation mellan omslagstyperna.
 - Alltså ingen typomvandling mellan omslagstyper
 - För Integer och Double och finns dock en gemensam supertyp Number.
- Vi behöver omslagstyper t.ex. då vi använder samlingar (de kan bara lagra referenstyper), se Samlingar.

Det finns omslagstyper för alla primitiva typer, ...

- T.ex. Boolean och Character.

Omslagstyper, Boxing och Likhet

```
Double d1 = 5.0;
Double d2 = 5.0;
out.println(d1 == 5.0);    // True, unboxing

out.println(d1 == d2);     // False! No boxing
out.println(d1.equals(d2)); // True, better use!

out.println(d1 == 0 + d2); // True, unboxing
out.println(d1 == 1 * d2); // True

Integer i1 = 99;
Integer i2 = 99;
out.println(i1 == i2);     // True! ... caching!
```

16

Omslagstyper håller referenser till objekt.

- == -operation ger referenslikhet för objekt av omslagstyper
 - Om jämförelsen görs mot en primitiv typ sker unboxing och värdena jämförs.
- För värdelikhet måste man annars använda equals().
 - Förutom för små heltal (< 128) finns det färdiga objekt, d.v.s. == fungerar
 - Kallas : [caching](#) (av effektivitetsskäl)

Array-typer

```
// Primitive arrays
int[] ia = {1, 2, 3};
double[] da = {1.0, 2.0, 3.0};
da = new double[]{1, 2, 3}; // Conversion

// Reference (wrapper) type arrays
Integer[] iia = {1, 2, 3}; // Boxing
Double[] dda = {1.0, 2.0, 3.0};
Double[] dda = {1, 2, 3}; // Bad boxing
```

17

Array-typer skapas utifrån någon existerande typ

- Primitiva eller ...
- Referenstyper
 - OBS! Att vid initiering av variabler kan implicit typomvandling eller boxing förekomma.

Subtyper för Primitiva Arrayer

```
// No subtype relations for primitive arrays
int[] ia = {1, 2, 3};
double[] da = {1.0, 2.0, 3.0};

//int <: double but *NOT* int[] <: double[]
ia = da;      // No
da = ia;      // No
da = (double[]) ia; // No!
```

18

Finns ingen super/subtyp relation mellan primitiva arrayer.

Förklaring:

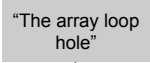
- Leder till hål i typsystemet, se Subtyper för Referens-Arrayer

Subtyper för Referens-Arrayer

```
Integer[] iia = ...;
Double[] dda = ...;
iia = dda; // No super/sub relation
dda = iia;

// If Integer <: Number then Integer[] <: Number[] !!!!!
// Decided by Java designers
Number[] ns = iia; // Ok!
ns[0] = 4; // Ok! 4 is boxed to subtype Integer
ns[1] = 4.5; // Will compile, runtime ArrayStoreException!!

iia = (Integer[]) ns; // If not exception ...
out.println( ~iia[1] ); // ...this would be allowed! Bad!
```



19

Mellan arrayer av primitiv typ finns inga super/sub relationer

- T.ex. `int <: double` innebär inte att `int[] <: double[]`

P.g.a. vissa omständigheter bestämda de som utvecklade Java att för referenstyper gäller ...

- .. om `S <: T` så är `S[] <: T[]` !!!
- T.ex. `Integer <: Number` innebär att `Integer[] <: Number[]`
- Leder till ett "hål" i typsystemet (the array loophole)
 - Kompilatorn släpper igenom något som kan ge typfel under körning
- För att åtgärda detta gör Java under körning alltid en test vad man försöker stoppa in i en array av referenstyp.
 - Försöker man stoppa in ett värde av felaktig typ får man en ArrayStoreException.
 - D.v.s. det finns en del typinformation under körning

Uppräkningstyper

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

WeekDay d1 = WeekDay.FRI;
WeekDay d2 = WeekDay.THU;
WeekDay d3 = WeekDay.FRI;

out.println(d1 != d2); // == Works!
out.println(d1 == d3); // Same object

// Loop through all days
for (int i = 0; i < WeekDay.values(); i++) {
    // Do something
}
```

22

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 deklarerar en **uppräkningstyp** ([enumeration](#)) skapar vi en ny (egen) klasstyp med ett antal (uppräknade) värden (en enum är en slags klass).

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

Deklarationen av uppräkningstyp görs men det reserverade ordet **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)

Subtyper för Uppräkningsstyper

```
// Enumeration types
public enum WeekDay {
    MON, TUE, WED, THU, FRI
}
enum WorkingDay {
    MON, TUE, WED, THU, FRI; // Same values ...
}

// No super/sub relation
WeekDay week = WeekDay.FRI; // No
week = WorkingDay.FRI; // No
week = (WeekDay) WorkingDay.FRI; // No
```

21

En uppräkningsstyp är en klasstyp d.v.s. ingen super/subtyp relation mellan uppräkningsstyper

- En uppräkningsstyp kan inte ha några subtyper, det finns inga implicita typomvandlingar från något typ till en uppräkningsstyp.
- Däremot finns en supertyp, se Object senare.
- Explicit typomvandling kan endast ske till/från String, se Strängar.

Typen Object

```
// Type object is supertype to any reference type  
Object o = null;  
o = "Any";  
o = new Dog();  
o = WeekDay.MON;  
o = new int[]{1, 2, 3};  
o = 5;           // Boxing  
o = false;      // Boxing  
o = new Object(); // Use the class
```


Det finns en "översta" referenstyp, typen Object. Object är supertyp till alla typer men har inte någon egen supertyp.

- D.v.s. det finns en klass Object (måste ha en klass för att introducera en typ).

null-värdet är enda värde i en namnlös typ som är subtyp till alla referenstyper.

Implicit Typomvandling med + - operatoren och String

```
"a" + 4.0 -> "a" + new Double(4.0).toString()  
                                -> "a" + "4.0" -> "a4.0"
```



Implicit typomvandling

```
out.println(dog) -> out.println(dog.toString());
```

Sträng inte supertyp till någon typ d.v.s. inget kan implicit omvandlas till String ...

- ...förutom vid två tillfällen
 - Då + operatoren har minst en operand av typen String.
 - out.println(), ... då egentligen metoden toString() anropas.

Explicit Typomvandling med String

```
// From primitive to String
String s = String.valueOf(45); // String class methods
s = String.valueOf(1.45);
s = Integer.toString(45); // Same using wrapper types
s = Double.toString(1.45);
// From String to primitive
int i = Integer.valueOf("678");
double d = Double.valueOf("4.57");


// From/to enum
String day = WeekDay.FRI.toString();
WeekDay w = WeekDay.valueOf("FRI");
```

Eftersom String inte har några subtyper måste all typomvandling ske explicit (förutom föregående bild).

Arv och Subtyp

```
public class Pet {  
    private String name;  
    public String getName() {  
        return name;  
    }  
}  
  
public class Dog extends Pet {  
    // Inherited  
    public String getName(){...}  
}
```

Dog <: Pet



25

Om en klass ärver (extends) en annan klass skapas en super/subtyp relation!

- Klassen som ärver blir subtyp
- Logiskt: Eftersom klassen som ärver kan minst lika mycket som klassen den ärver från!
 - Alla operationer som vi kan göra med superklassen kan vi göra med subklassen (men inte tvärtom).
 - Som tidigare: Subtypen kan mer!
- Se även Klasser.

Typomvandlingar vid Arv

```
public class Pet {  
    ...  
}  
  
public class Dog extends Pet {  
    ...  
}  
  
Pet p = new Dog(); // Implicit sub to super, ok  
  
Dog d = (Dog) p;   // Explicit super to sub, allowed  
                   // but possibly ClassCastException  
                   // at runtime
```

26

Typomvandling från sub till super sker automatiskt vid behov.

- Tvärtom får man (på egen risk) explicit typomvandla (undvik).

Statisk och Dynamisk Typ

```
// Yes, Object type is 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!

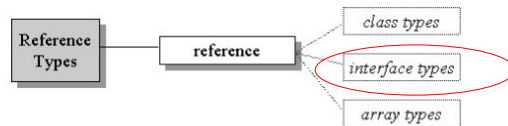
Huruvida typen för referensen till objektet (typen för objektet säger vi) och typen för variabeln är kompatibla bestämmas av ett stor antal komplicerade regler, varav den viktigaste är super/subtyp relationen

- 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 (vilka operationer kompilatorn godkänner, samma som för primitiva typer).
- Typen på objektet (objekttypen, dynamiska typen, runtime-typen) bestämmer vilken "version" av operationen (metoden) som skall köras
 - Ev. är metoden överskuggad (override). Se Klasser och Metoder.
 - Kallas också **sen bindning (lat/dynamic binding)**

Sammanfattning:

- En primitiv variabel innehåller ett värde av exakt den primitiva typen.
- En referensvariabel kan innehålla en referens till den deklarerade typen eller någon subtyp.

Gränssnittstyper



```
public interface MyList ... {  
    boolean isEmpty();  
    boolean add(int i);  
    int get(int index);  
}  
  
// Variable with interface type  
MyList list = ... ;  
list.add(4); // Ok, method in interface type
```

Ett **gränssnitt** ([interface](#)) är en samling av publika abstrakta metoder (public abstract behöver inte anges)

- Kan inte instansiera ett gränssnitt (med new) eftersom metoderna är abstrakta.
- Ett gränssnitt introducerar en referenstyp
 - De operationer som är tillåten för typen är de metoder vi angivit.
 - Innebär att vi kan deklarera en gränssnittstyp för en referensvariabel
 - Vi kan alltså skapa en ny typ utan att ange någon implementation av operationerna (finns inga metoder med körbar kod!)
 - Abstraktion! Vi kan ange att vi vill ha "något" som kan utföra vissa operationer (exakt vad detta är behöver vi inte ange).

I bilden deklareras ett gränssnitt MyList med ett antal metoder.

Vi säger också att ett gränssnitt definierar ett kontrakt.

Subtyper för Gränssnittstyper

```
public interface MyList ... {  
    ...  
}  
  
public class MyListImpl implements MyList {  
    ...  
}  
  
// Variable interface type, object as implementing type  
// MyListImpl <: MyList  
MyList list = new MyListImpl() ;
```

En klass som implementerar ett gränssnitt blir en subtyp till gränssnittstypen. Se Klasser

- Eftersom alla operationer som gränssnittstypen specificerar finns i objektet (kontrolleras av kompilatorn).

Typomvandlingar med Gränssnittstyper

```
public interface ISayable {
    String say();
}

public class Car {
    // Class has nothing
    // to do with ISayable!!!
}

public class TalkingCar extends Car implements ISayable {
    @Override
    public String say() { ... }
}

Car car = new Car();
ISayable s = (ISayable) car; // Compiles, why is this allowed ??? ...

car = new TalkingCar(); // Because could be like this! TalkingCar <: Car
ISayable s = (ISayable) car; // ... sub class may implement!
```

30

Vilket typ som helst kan explicit typomvandlas till en gränssnittstyp!!!

- Trots att objektet kanske inte implementerar gränssnittet d.v.s. inte är en subtyp.
 - Kompilering tillåten eftersom någon subklass kanske implementerar!
- Om objektet inte implementerar blir det körningsfel, undantag.

I Bilden: Ett TalkingCar-objekt kan sägas ha flera typer (TalkingCar, Car eller ISayable). Gäller generellt med OO-språk.

Interface och getClass()

```
public interface ISayable {
    String say();
}

public class Car {
    // Class has nothing
    // to do with ISayable!!!
}

public class TalkingCar extends Car implements ISayable {
    @Override
    public String say() { ... }
}

Car c = new Car();
TalkingCar tc = new TalkingCar();
ISayable s = tc;

out.println(c instanceof Car); // true
out.println(tc instanceof Car); // true
out.println(c instanceof TalkingCar); // false

out.println(c.getClass() == Car.class); // What's the type of the object?
out.println(s.getClass() == TalkingCar.class);
```

31

För att under körning undersöka ett objekts typ kan man använda

- instanceof: Sant om objektet är av typen eller någon subtyp
- getClass(): Ger objektets typ (klassobjektet för objektet)

Undvik, ger oflexibel kod, föredra override se Klasser och Metoder.

- Dock, för att säkert kunna typomvandla från super till subtyp kan man undersöka om detta går m.h.a. instanceof (slipper ClassCastException)

Generiska Typer

```
// A generic class/type
class Box<T> {
    T content;    // T is any reference type
}

// Supply existing type at declaration
Box<String> sm = new Box<>();
sm.content = "remember";

Box<Integer> im = new Box<>();
im.content = 1;
```

Klass och gränssnittstyper kan ta en annan typ som parameter kallas för en generisk typ

- En typparameter skrivs inom <...> och betecknas ofta med T
 - Vi har sett liknande vid generiska metoder.
- Då man använder den generiska typen vid en variabeldeklaration måste man ge typparametern ett värde
 - Man ger en existerande referenstyp som argument.

Tyvärr är generiska typer i Java svårhanterliga, många saker man inte kan göra t.ex. `new T()` eller dylikt.

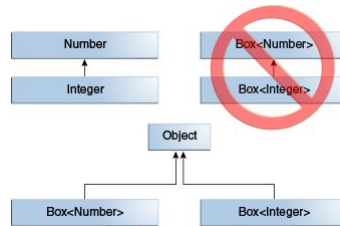
- Mer i senare kurser.

Alla samlingar i Java är generiska, typparametern anger typen för elementen i samlingen.

- Se Samlingar

Subtyper för Generiska Typer

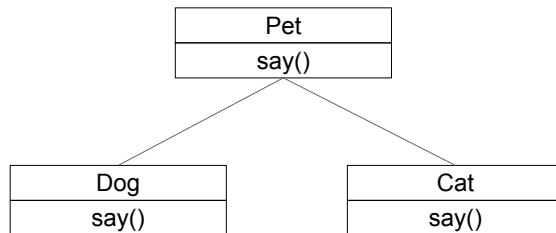
```
Box<Integer> m1 = new Box<>();  
Box<Double> m2 = new Box<>();  
  
Box<Number> m3 = m1;           // No  
Box<Number> m4 = (Box<Number>) m1; // No  
Box<Number> m5 = m2;           // No
```



Generiska klasser har inget super/subtype förhållande.

- Även om typargumentet har super/sub
 - `Integer <: Number` men: `Box<Integer>` är inte subtyp till `Box<Number>`
- Går att åstadkomma super/subtyp relation, mer i senare kurser.
- Alla generiska typer är kompatibla med `Object`.

Polymorfism



34

Ett sammanfattande begrepp för "att det som händer beror på de inblandade typerna" är

polymorfism. Vi har sett följande

- Vilken metod som körs beror på parametrarnas typer, överlagring/overloading
- Vilken metod som körs beror på objektets typ, överskuggning/override
- Generiska typer.

Anm: Det som vanligen avses när man talar om polymorfism i OO sammanhang är override.

- Metoden **say()** i bilden.