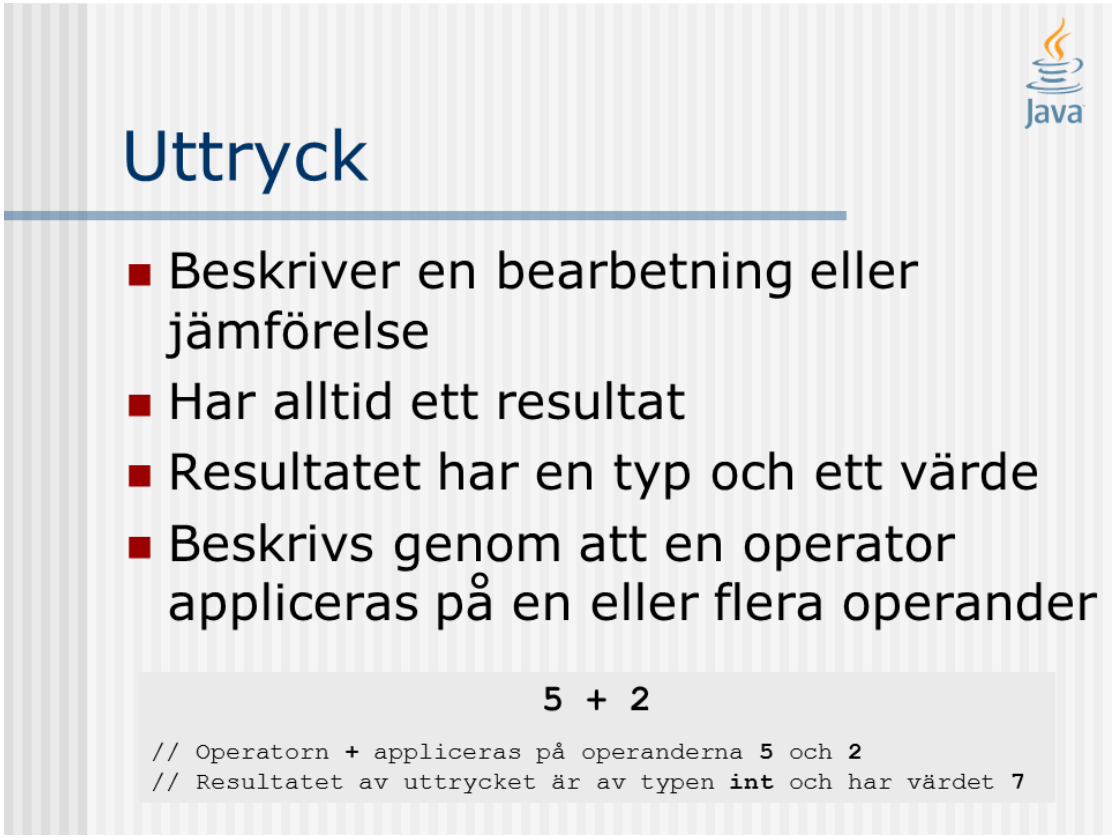
Mittuniversitetet
MID SWEDEN UNIVERSITY

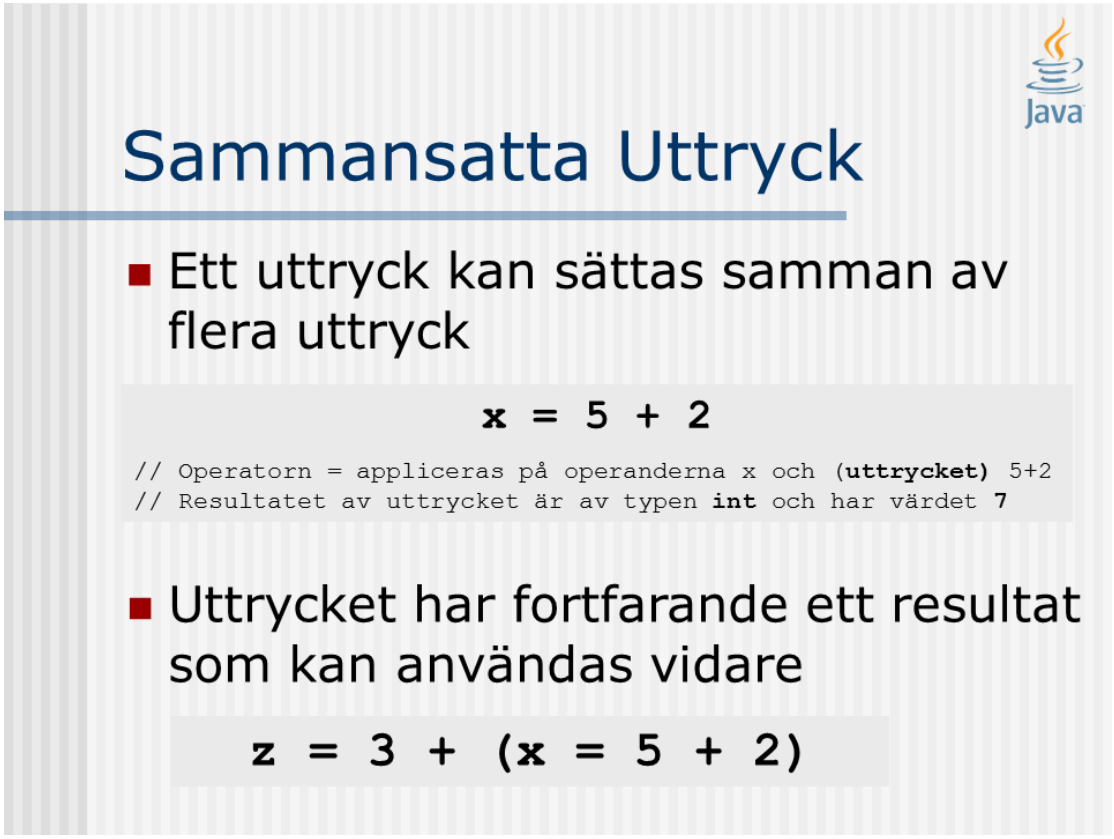
Lektion 1

Java vs. C++

Java för C++-programmerare, 7,5 h

Syfte:	Ge en snabb genomgång av Javas operatorer och datatyper. Det blir främst en repetition eftersom det mesta överensstämmer med C++. Sedan pekas ut andra viktiga skillnader mellan C++ och Java för att underlätta övergången mellan språken..
Att läsa:	Inget kapitel i kursboken Comparison of Java and C++ https://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B





Inga skillnader gentemot C++.

Prioritet Och Associativitet

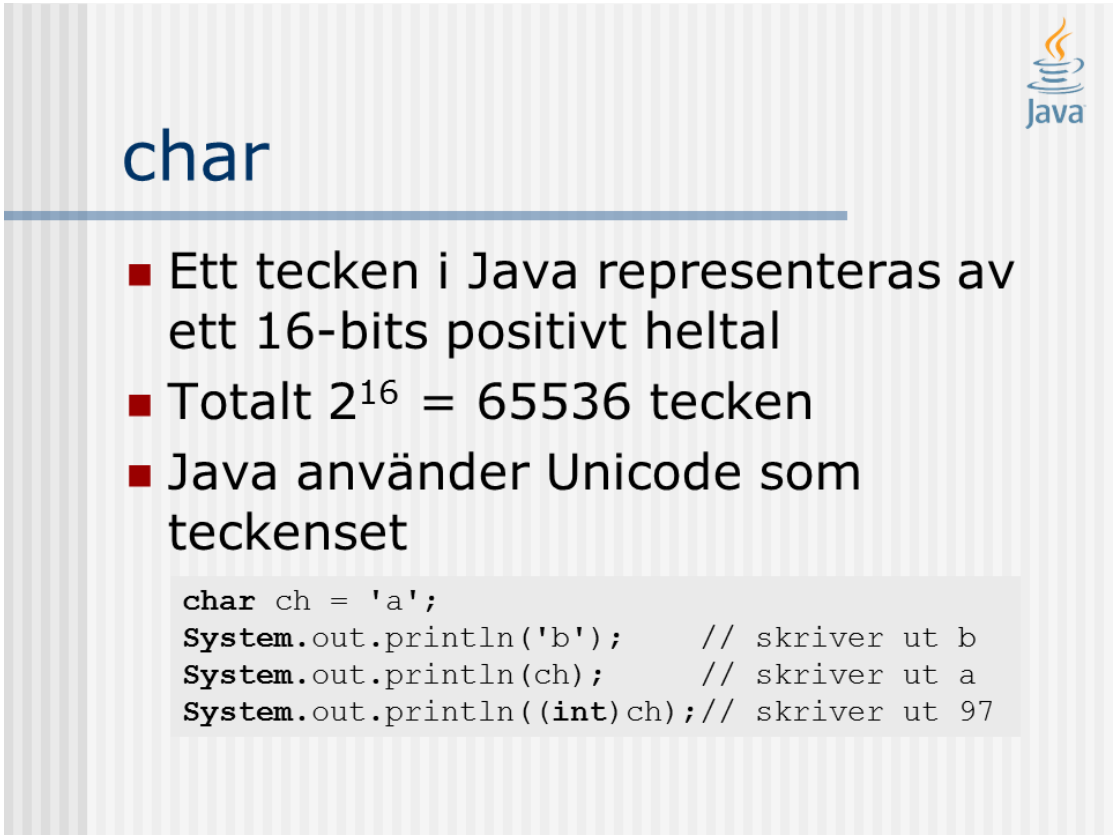
- För att ett givet uttryck ska alltid ge ett bestämt resultat finns det regler hur ett uttryck ska evalueras
- Prioritering av operatorerna
  - 5 * 6 – 5 // prioritet
- Tillämpning av associativitet
  - 60 / 6 * 5 // vänsterassociativitet
  - a = b = c = 42 // högerassociativitet
- Med parenteser kan evalueringsordningen bestämmas

```
x = ((a + b) * (c / d)) - 42;
```

Primitivatyper

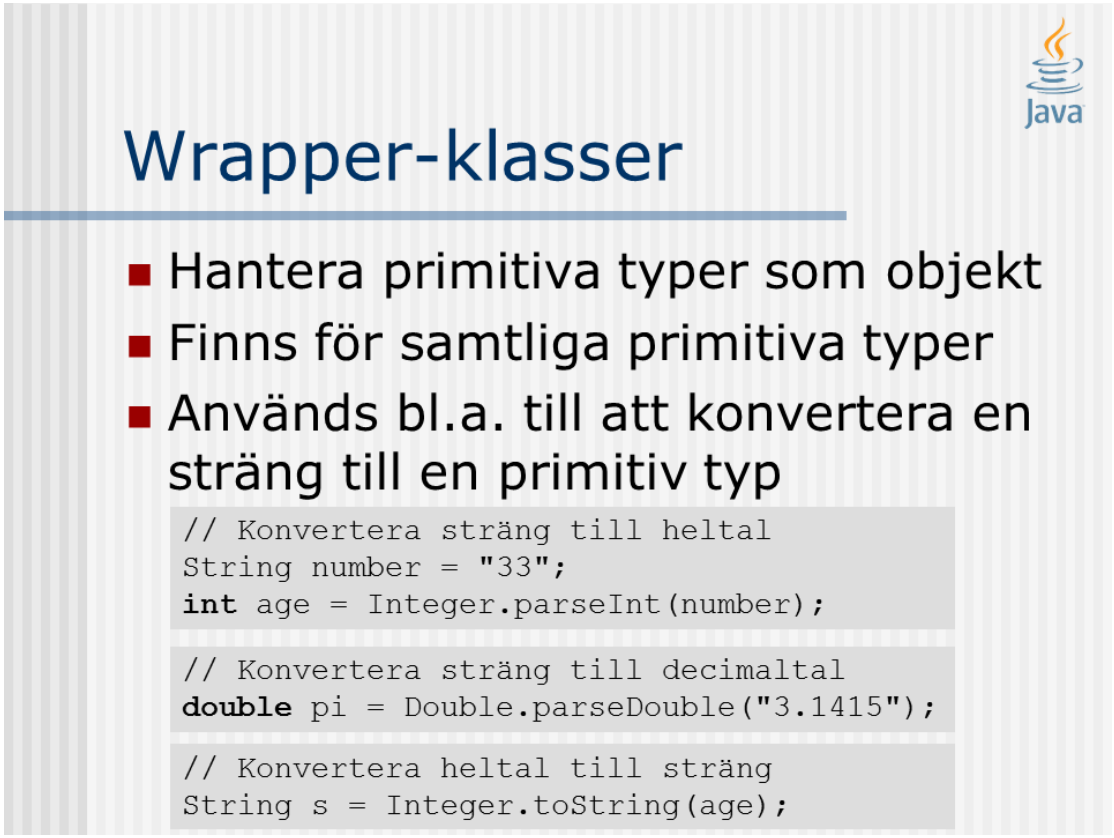
`boolean`	boolsk typ
`char`	16-bitar Unicode
`byte`	8-bitars heltal
`short`	16-bitars heltal
`int`	32-bitars heltal
`long`	64-bitars heltal
`float`	32-bitars flyttal
`double`	64-bitars flyttal

- Samtliga typer är definierade i detalj, vilket innebär att t.ex en `int` alltid är 32 bitar



Här ser vi en del skillnader:

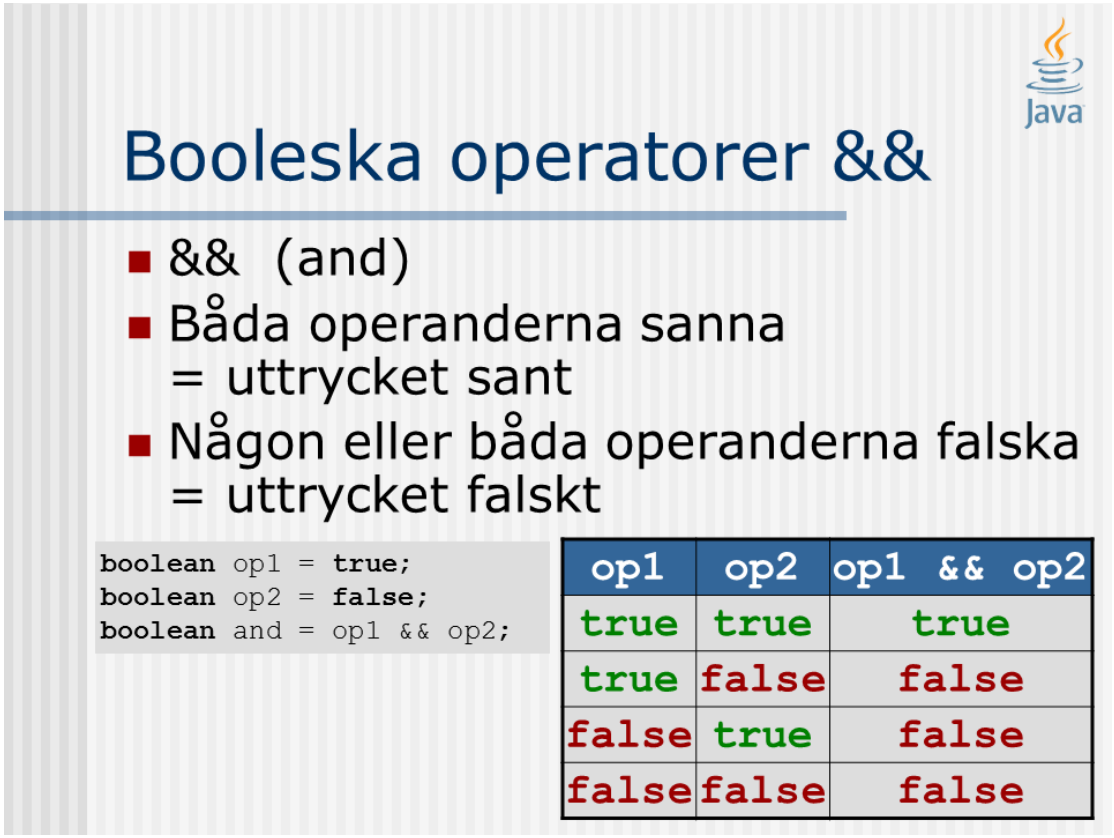
- där C++ har en 8-bitars char har Java en 16-bitars Unicode char
- Java har en mer exakt specifikation av antalet bitar för de olika typerna. T.ex. lagras en Java-float i 32-bitar medan en C++-float ska lagras i *minst* 32 bitar.

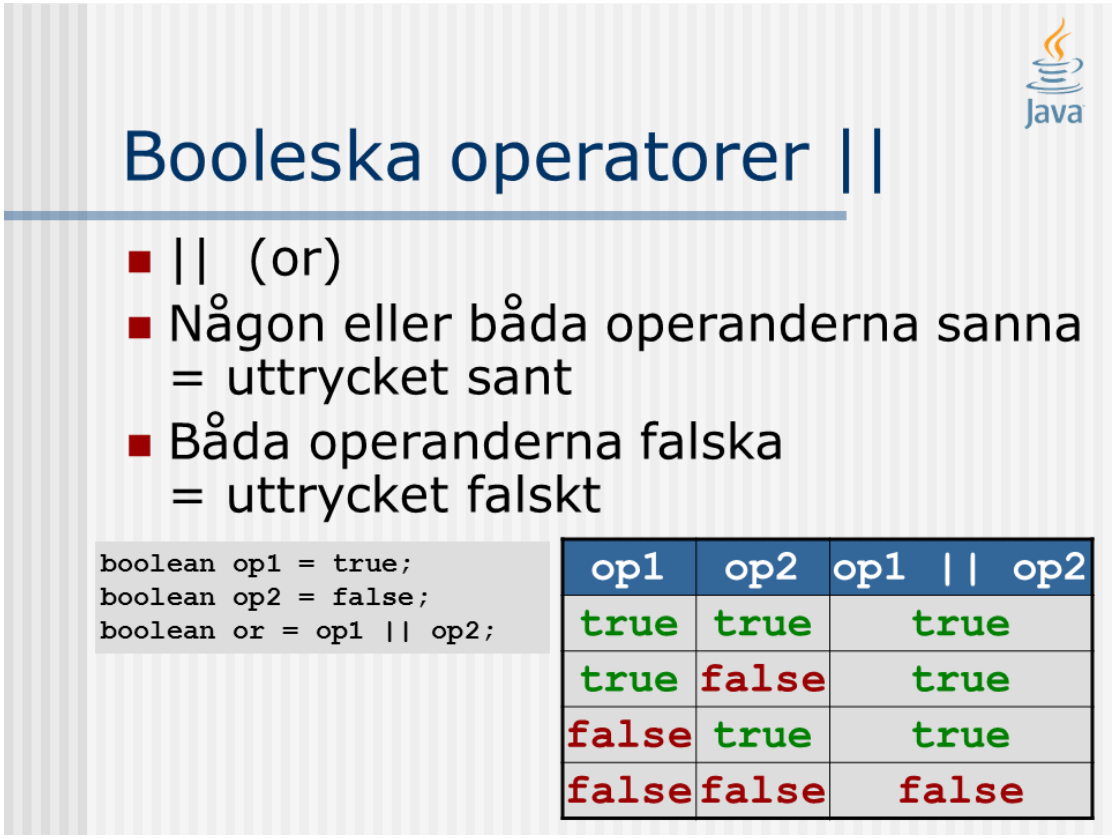


Java har wrapper-klasser för många primitiva typer:

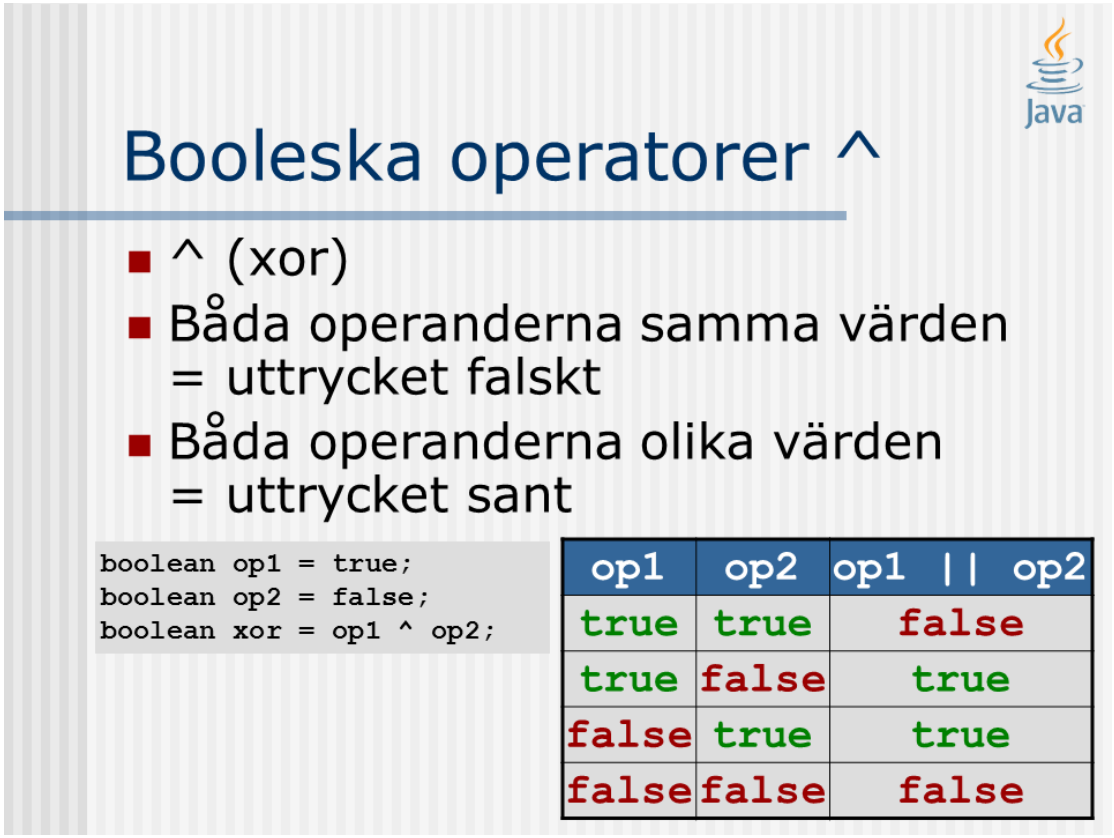
int	↔	Integer
float	↔	Float
double	↔	Double
boolean	↔	Boolean

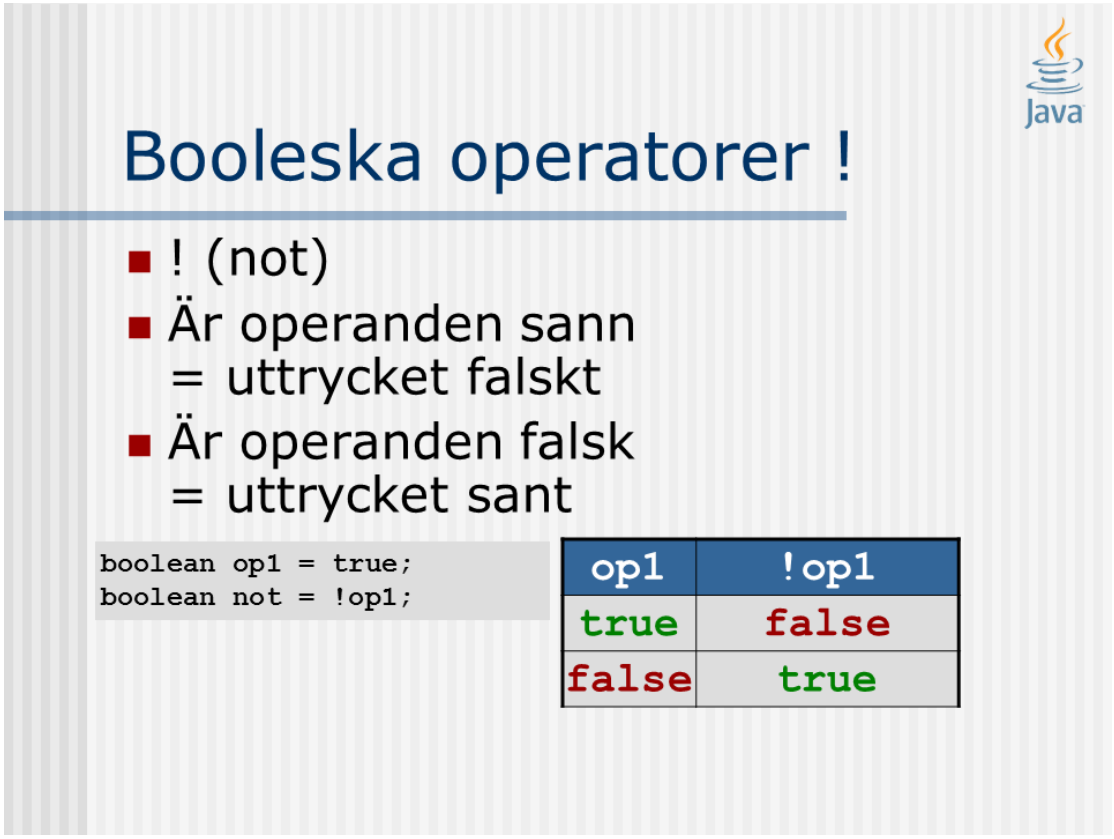
En av anledningarna är att Javas kontainerklasser (samlingsklasser) som t.ex. `Vector` och `ArrayList` endast kan ha klasstyper som elementtyp. `ArrayList<Integer>` är OK medan `ArrayList<int>` inte fungerar. Konvertering mellan t.ex. `int` och `Integer` sker i många fall implicit.





Inga skillnader gentemot C++.





Inga skillnader gentemot C++.

Aritmetiska Operatorer

- Operatorer

Operation	Operator	Java	Algebra
addition	+	m +2	m + 2
subtraktion	-	m – 2	m - 2
multiplikation	*	m * 2	2m, m · 2
division	/	m / 2	$\frac{m}{2}$
modulus	%	m % 2	m modulo 2

- Prioritet

Prioritet	Operator
1	( )
2	* / %
3	+ -

- Heltalsdivision resulterar i ett heltal

```
3 / 2       → 1       // Heltal
3.0 / 2.0   → 1.5     // Decimaltal
3 / 2.0     → 1.5     // Decimaltal
3.0 / 2     → 1.5     // Decimaltal
```

Inkrementeringsoperatorer

- Används för att öka eller minska ett värde med 1 (++, --)

Start	Uttryck	Operation	Tolkning	Resultat
int j = 0, k = 0;	j = ++k;	prefix	k = k + 1, j = k;	j är nu 1 och k är 1
int j = 0, k = 0;	j = k++;	postfix	j = k, k = k + 1	j är nu 0 och k är 1
int j = 1, k = 1;	j = --k;	prefix	k = k – 1, j = k	j är nu 0 och k är 0
int j = 1, k = 1;	j = k--;	postfix	j = k, k = k - 1	j är nu 1 och k är 0

- Använd först k, öka det sedan
- Öka först k, använd det sedan

Inga skillnader gentemot C++.

Operator	Typ	Exempel	Tolkning
=	Vanlig tilldelning	m = n;	m = n;
+=	Addition, sen tilldelning	m += 3;	m = m + 3;
-=	Subtraktion, sen tilldelning	m -= 3;	m = m – 3;
*=	Multiplikation, sen tilldelning	m *= 3;	m = m * 3;
/=	Division, sen tilldelning	m /= 3;	m = m / 3;
%=	Rest, sen tilldelning	m %= 3;	m = m % 3;


```
if (x >= 0)
  System.out.println("x är större än eller lika med  0");
if (x != -1)
  System.out.println("x har inte värdet -1");
```

Inga skillnader gentemot C++.

Jämförelseoperatorer

- Används för att jämföra variabler och uttryck med varandra
- Resultatet är av typen boolean

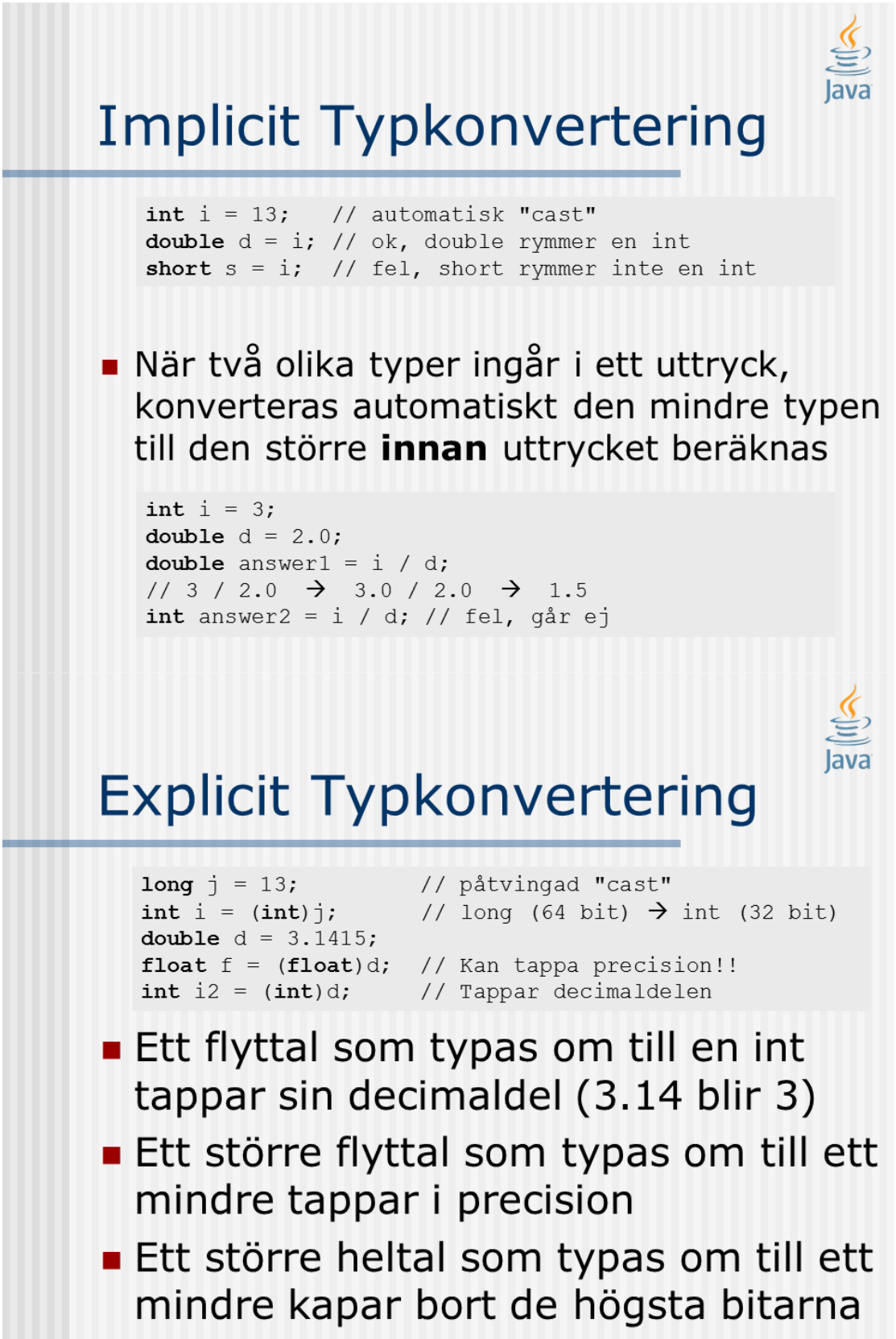
Operator	Typ	Exempel	Resultat
<	mindre än	5 < 10	true
>	större än	10 > 10	false
<=	mindre än eller lika med	11 <= 10	false
>=	större än eller lika med	10 >= 10	true
==	lika med	5 == 5	true
!=	ej lika med	5 != 5	false

Prioritet

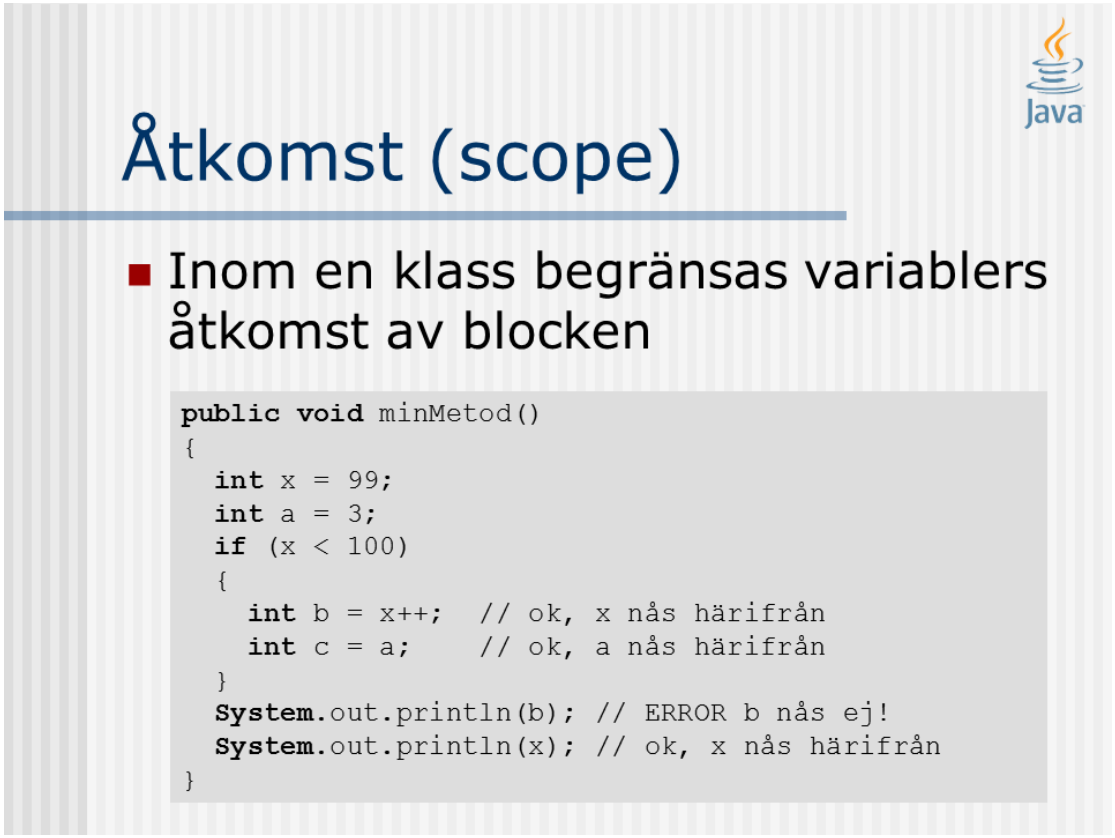
Prioritet	Operator	Typ
1	( )	paranteser
2	++ --	ökning, minskning
3	* / %	multiplikation, division, modulus
4	+ -	addition, subtraktion
5	< > <= >=	mindre-, större än (eller lika med)
6	== !=	är lika med, skiljt ifrån

- Använd parenteser för att tydliggöra

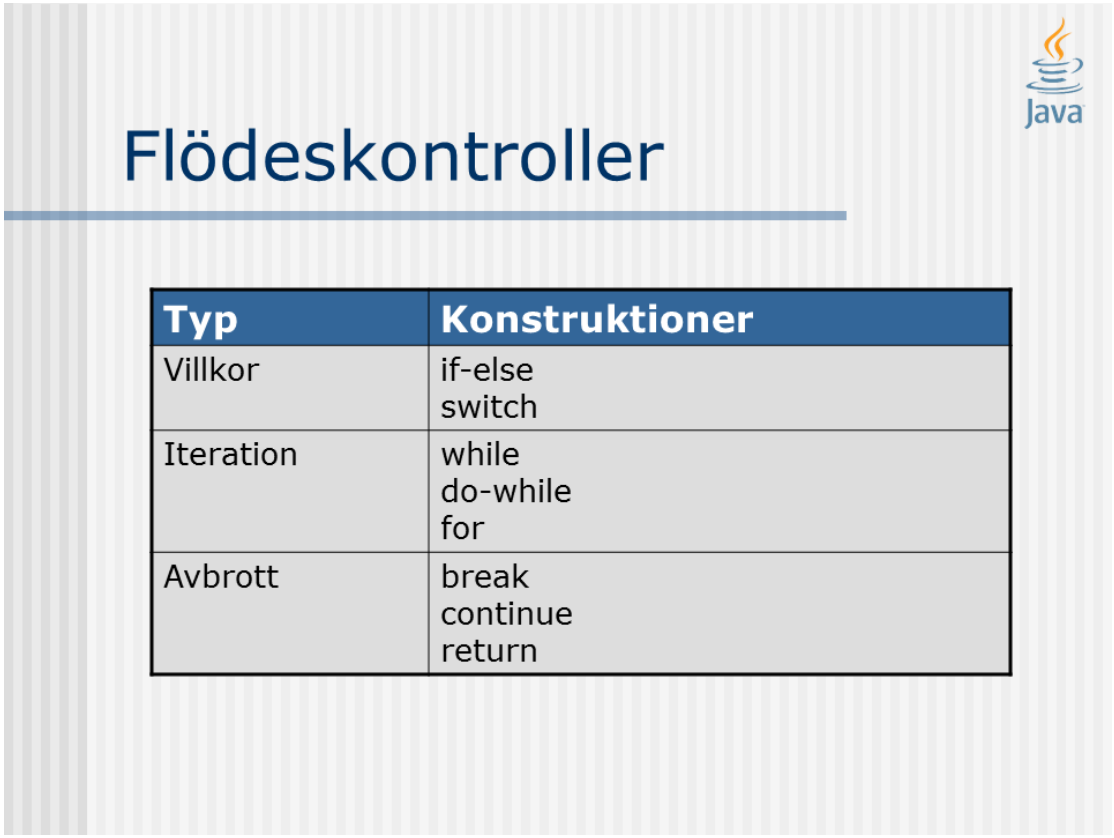
Inga skillnader gentemot C++.



Java använder den gamla C-syntaxen, `(new type)`, för att göra ett typecast men saknar motsvarigheter till operatorerna `static_cast`, `reinterpret_cast` och `const_cast` i C++.



Inga skillnader gentemot C++.



Typ	Konstruktioner
Villkor	if-else switch
Iteration	while do-while for
Avbrott	break continue return

Alla kontrollstrukturer från C++ fungerar med samma syntax i Java:

- alla konstruktioner med if och else
- switch
- for
- while
- do – while
- break
- continue
- return

I Java finns dessutom en förenklad typ av for-sats, mer om den i samband med Collections.

Andra viktiga skillnader mellan Java och C++.

I Skansholms ”Java direkt med Swing” eller Schildts ”Java: the Complete Reference” finns ingen bra beskrivning av skillnaderna mellan språken. Ett antal viktiga skillnader finns beskrivna här nedan. Om du vill fördjupa dig utöver detta så kan du söka på nätet. En sökning med ”Java C++ differences” eller ”Java C++ comparison” ger mängder med träffar.

Java är till skillnad från C++ ett *rent* objektorienterat språk. Det har alltså inga fristående funktioner. Mycket av inbyggda enkla typer och syntax har hämtats från C++ så därför känns Java-kod ändå väldigt bekant för en C++-programmerare.

Terminologi

- Det som kallas datamedlemmar i en C++-klass motsvaras i Java av **instansvariabler** (unika värden för varje instans).
- Statiska datamedlemmar i C++ motsvaras i Java av **klassvariabler** (gemensamma för alla instanser av klassen).
- Medlemsfunktioner i C++ motsvaras av **metoder** i Java.

Klasser, objekt och metoder

Java har liksom C++ inbyggda enkla typer som inte är klassbaserade, det är framförallt de numeriska typerna `int`, `float`, `double` m.fl. men även den booleska typen `boolean`. Variabler av dessa typer kan skapas lokalt inom ett block `{…}` precis som lokala variabler i C++ som har lagringsklass auto. Alla *klassinstanser* (objekt av klasstyp) i Java måste däremot allokeras med hjälp av operatorn **new**. I C++ kan även klassinstanser skapas som lokala objekt på stacken men det går alltså inte i Java. T.ex. finns i Java klassen `String`. Följande kod allokerar en `String` med namnet `myString`:

```
String myString;              // ”referens” till en String
myString = new String();      // Alltid anrops-parantser…
```

eller

```
// deklaration och tilldelning
String myString = new String();
```

Observera att parenteser för metodanrop måste finnas även om inga argument skickas med till konstruktorn. Ett objekt som skapats med `new` lever automatiskt så länge det används, dvs så länge det refereras från något annat objekt i programmet. När det inte längre behövs kommer det så småningom automatiskt att deallokeras genom den inbyggda ”garbage collection”-mekanismen. Java saknar motsvarighet till operatorn delete i C++.

I Java finns inga fristående funktioner, endast metoder inom klasser. Det närmaste man i Java kommer fristående funktioner är statiska metoder. I C++ startar exekveringen i en fristående main-funktion medan Java startar exekveringen i en

statisk main-metod i någon klass. En Java-version av ”Hello world” kan se ut på följande sätt:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
} // Inget semikolon!
```

Observera: inget semikolon efter den avslutande krullparantesen.

I C++ har vi operatorn **::** som ”scope resolution operator” medan Java använder en punkt, **.** .
En klassmetod (statisk funktion) kan ju anropas utan att någon instans av klassen finns och i Java görs det alltså med `Klassnamn.metodnamn();`

Referenser (och pekare…)
I all dokumentation till Java sägs att språket saknar pekare och att accesser till objekt görs via referenser. Man brukar även säga att parameteröverföring vid metodanrop sker genom ”pass-by-reference”. Tyvärr är detta en missledande beskrivning som inte stämmer överens med det allmänt vedertagna innebörden av begreppet referens, t.ex. som det används i C++, och måste betraktas som ett misstag av Sun Microsystems som skapat språket. **All** parameteröverföring i Java är i verkligheten av typen ”pass-by-value”. Det innebär att de aktuella parametrarna (argumenten) vid metodanrop *kopieras* till motsvarande formella parametrarna (dvs. de parametrar som man har definierat i metodhuvudet). Problemet för en C++-programmerare är att Javas referensbegrepp inte motsvarar det referensbegrepp som används i C++.

I C++ (och de flesta andra språk) är en referens ett *alias* för ett objekt, dvs. referensen motsvarar det ursprungliga objektet och förändringar av referensen förändrar det ursprungliga objektet. Det som kallas referens i Java är snarare vad vi i C++ kallar en pekare fast den är begränsad till att bara peka på ett objekt. Operatorerna för dereferens- och adressoperatorer (* respektive &) i C++ är avsiktligt borttagna från Java.

Om vi återgår till koden

```
String myString;
myString = new String();
```

är alltså `myString` egentligen en `String`-pekare, fast vi kallar det för en referens i Java.

För att mer konkret visa vad detta kan innebära ska vi studera C++-funktionen

```
void swap(MyClass &a, MyClass &b) {
  MyClass tmp = a;
  a = b;
  b = tmp;
}
```

Vi definierar två objekt,

```
MyClass x(10),  y(20);
```

Efter anropet `swap(x,y)` finner vi att `x` och `y` har bytt innehåll.

Om vi gör en motsvarande statisk Java-metod (i klassen `MyClass`),

```
static void swap(MyClass a, MyClass b) {
  MyClass tmp = a;
  a = b;
  b = tmp;
}
```

och sedan definierar objekten

```
MyClass x = new MyClass(10);
MyClass y = new MyClass(20);
```

och gör anropet `MyClass.swap(x,y);`

kommer vi att finna att `x` och `y` har samma innehåll efter anropet som före. Det som har hänt är att de formella parametrarna (pekarna) `a` och `b` har bytt värden med varandra men att det inte har påverkat de aktuella parametrarna (argumenten) `x` och `y`.

Alltså: när det i Java talas om ”reference” och ”pass-by-reference” underlättar det ibland att tänka ”pointer” och ”pass-by-value”.

Om du vill fördjupa dig i detta kan du studera ”Java is Pass-by-Value, Dammit!” (http://javadude.com/articles/passbyvalue.htm).

Dynamisk bindning
Eftersom både Java och C++ är fullfjädrade objektorienterade språk implementerar de dynamisk bindning. C++ aktiverar dynamisk bindning för en medlemsfunktion med nyckelordet **virtual** i klassdefinitionen. Anrop via referens eller pekare binder då anropet till den implementation av medlemsfunktionen som det aktuella objektet har. Detta sker under runtime och därför kallas det även ”late binding” till skillnad från statisk bindning som sker redan vid kompileringen.

Skillnaden mellan Java och C++ är att i Java är den dynamiska bindningen aktiverad som default utan att något speciellt nyckelord används. För att *förhindra* att en metod kan omdefinieras i deriverade klasser används nyckelordet **final**.

Arv
I Java uttrycks arv med nyckelordet **extends**.

Ex.

```
class JavaBase {  // Base class
…
}

class JavaDerived extends JavaBase { // Derived from JavaBase
…
}
```

Abstrakta klasser
För att göra en klass abstrakt (ej möjlig att instansiera) i C++ ska minst en medlemsfunktion vara ”pure virtual” vilket åstadkoms genom att den ”nollas” i klass-definitionen.

Ex. C++

```
class AbstractCppClass {
public:
          virtual void pureVirtualFunc( ) = 0; // pure virtual
          virtual void virtualFunc( ) {...} // virtual
};
```

I Java finns nyckelordet **abstract** som används för att kvalificera både klasser och enskilda metoder som abstrakta.

Ex. Java

```
public abstract class AbstractJavaClass {
  public abstract void abstractMethod( );  // abstract
  public void method( ) {...} // inte abstract
}
```

Interface och arv
Nyckelordet **interface** i Java motsvarar en C++ - klass som endast definierar ”pure virtual functions” utan några implementationer.

Ex. C++

```
class CppInterface {
public:
          virtual void func1( ) = 0;
          virtual void func2( ) = 0;
          . . .
};
```

Ex. Java

```
interface JavaInterface {
  public void method1( );
  public void method2( );
  . . .
}
```

Konstruktionerna med interface och tillhörande nyckelord **implements** i Java är nödvändiga **eftersom Java inte tillåter multipelt arv**. Om Java-klassen `MyJavaClass` ska ärva klassen `JavaBase` och samtidigt implementera interfacet `JavaInterface` blir koden

```
class MyJavaClass extends JavaBase implements JavaInterface {
...
}
```

Motsvarande kod i C++ blir

```
class MyCppClass : public CppBase, public CppInterface {
...
};
```

Defintion vs Implementation
I Java, till skillnad från C++, kan inte klassdefinition separeras från implementationen av klassens metoder, hela implementationen skrivs i klassdefinitionen i .java-filen.

include i C++ --> import i Java
I Java finns det inga header-filer att inkludera eftersom ingen uppdelning görs mellan definition och implementation. Istället använder man nyckelordet **import** i början av en fil för att få tillgång till existerande samlingar av klasser. En sådan samling kallas en **package** (reserverat ord), eller paket på svenska.

Strömmar
De fördefinierade stream-objekten `cout`, `cerr` och `cin` i C++ motsvaras i Java av objekten.
`System.out`, `System.err` respektive `System.in`.

Dynamisk typomvandling
I Java används operatorn `instanceof` för att under runtime testa om ett objekt är av en viss typ så att ett 'downcast' i en klasshierarki kan göras på ett säkert sätt.
Motsvarigheten i C++ är operatorn `dynamic_cast<new type>(argument)`.