Föreläsning 3:
Underprogram och Javas standardtyper

Christian.Soderberg@cs.lth.se

29 januari 2013

Allmänt om objekt

- Varje objekt har ett *tillstånd* (exempelvis position på skärmen, riktning, pennans läge, …).
- Vi kan ändra tillståndet hos ett objekt genom att *anropa operationer* på objektet (exempelvis att få en sköldpadda att vrida åt höger eller vänster, eller gå framåt).
- *Parametrarna* till operationerna är deklarerade så att vi skall veta vilkets slags värde vi skall skicka in.
- Vi kan hämta information om objektens tillstånd genom att anropa operationer som returnerar värden – *returtypen* anger vilket slags värde vi får tillbaka, `void`-operationer returnerar inget värde alls.

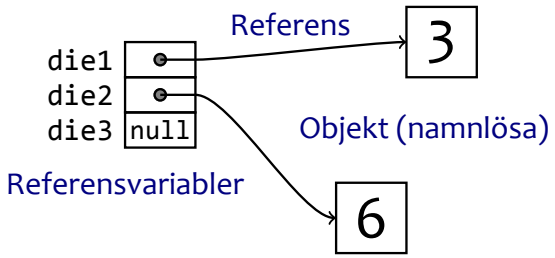
Begreppet klass

- En *klass* beskriver objekt av ett givet slag, vi kan ha många objekt av varje klass (exempelvis många tärningar och sköldpaddor).
- Klassen (typen) för ett objekt avgör vad vi kan göra med objektet – de operationer som vi kan anropa på ett objekt beskrivs i *klasspecifikationen* för dess klass.

Klasser i kapitel 3

- `Die`: beskriver speltärningar.
- `GraphicsWindow`: beskriver ett slags grafiskt fönster.
- `Turtle`: beskriver små 'sköldpaddor' som kan krypa omkring i `GraphicsWindow`-fönster.
- Olika slags geometriska figurer (rektangel, ellips, etc).
- `DotWindow`: ett annat slags grafiskt fönster som vi kan sätta ut prickar i.
- `Random`: låter oss dra olika slags slumptal.
- `Scanner`: låter oss läsa in från tangentbordet.

Objekt och referenser



- Objekten själva är namnlösa.
- Vi kommer åt våra objekt med hjälp av *referenser*.
- Referenser lagras i referensvariabler.
- En referensvariabel kan referera till ett objekt, eller ha värdet `null`.
- För att anropa en operation på ett objekt använder vi en referens till objektet:

```
die2.roll();
```

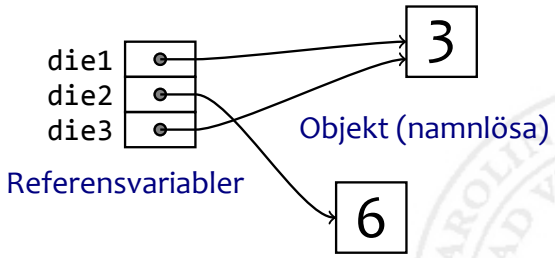
Att skapa objekt

- Objekt skapas med new-anrop (som i sin tur anropar *konstruktorn*).
- Ett objekt får ett väldefinierat starttillstånd när vi anropar konstruktorn.
- När vi skapar ett objekt får vi tillbaka en referens till det nya objektet.

Mer om referenser

Flera referensvariabler kan referera till samma objekt:

```
Die die1 = new Die();
Die die2 = new Die();
Die die3 = die1;
```



Bortglömda objekt och skräpsamling

När inga referenser finns kvar till ett objekt tas det bort ur minnet av *skräpsamlaren* (*garbage collector*).

Kommentarer i källkoden

- Två sorters kommentarer:
  - Enradskommentarer, inleds med //
  - Flerradskommentarer, omges av /* och */
- Kommentera gärna *varför* du gör något i ditt program, men kommentera inte *vad* du gör (om det ändå är uppenbart)

Om block

- Det som skrivs innanför ett par klamrar utgör ett *block*.
- Vi skjuter *alltid* in satserna inuti ett block fyra steg längre än omgivande block.
- Alla variabler som deklareras innanför ett block finns bara inuti blocket
- Variabler som deklareras i rubriken i en `for`-sats räknas som deklarerade inuti `for`-satsens block

Om block

- Det som skrivs innanför ett par klamrar utgör ett *block*.
- Vi skjuter *alltid* in satserna inuti ett block fyra steg längre än omgivande block.
- Alla variabler som deklareras innanför ett block finns bara inuti blocket
- Variabler som deklareras i rubriken i en `for`-sats räknas som deklarerade inuti `for`-satsens block

Underprogram

Våra program blir mycket enklare att skriva och läsa om vi kan anropa *underprogram* som löser delar av våra problem:

```
System.out.print(”Summan är ”);
System.out.println(sum(first, last));
```

istället för:

```
int sum = 0;
for (int term = first;
         term <= last; term = term + 1) {
    sum = sum + term;
}
System.out.print(”Summan är ”);
System.out.println(sum);
```

Underprogram

- Det finns i Java mängder av färdigskrivna underprogram.
- Vi kan enkelt skriva egna underprogram.
- Vi skiljer mellan två typer av underprogram:
  - Underprogram som beräknar värden – vi kallar dem *funktioner*.
  - Underprogram som gör något utan att skicka tillbaka något värde – vi kallar dem *procedurer*.

Funktioner

- En funktion innehåller ett antal programrader som vi kan anropa för att beräkna ett värde.
- Funktionen är ett eget block – vi kan ha nya variabler inne i funktionen, och de är helt frikopplade från det anropande programmet.
- Vid anropet skickar vi in de värden som funktionen måste känna till för att utföra beräkningen – dessa värden kallar vi *parametrar*, och de hör till funktionens block (ungefär som räknare i `for`-loopar).
- Funktionen returnerar det beräknade värdet med en `return`-sats – lokala variabler och parametrar försvinner när funktionsberäkningen är klar.

Underprogram och önsketänkande

- Ofta är det en god idé att önsketänka när vi programmerar – vi kan anropa underprogram som vi önskar att vi hade.
- Vi måste sedan skriva de underprogram som vi anropat, och kan då återigen önsketänka.
- På detta vis får vi mindre och mindre problem, och inte förrän problemen är helt triviala slutar vi önsketänka.

Olika slags tal

- Vi skiljer i Java mellan heltal och reella tal.
- Heltal är exakta, och vi försöker använda dem när vi är helt säkra på att de värden vi hanterar kommer att vara heltal.
- De flesta reella tal är avrundade, även många tal som ser ’enkla’ ut.

Olika slags heltal

- Vi har fyra olika slags heltal, som kräver olika mycket minne:
  - `byte`: −128 . . . 127
  - `short`: −32 768 . . . 32 767
  - `int`: −2 147 483 648 . . . 2 147 483 647
  - `long`: −9 223 372 036 854 775 808 . . . 9 223 372 036 854 775 807
- I kursen kommer vi nästan uteslutande att använda `int`.

Större och mindre typer

- Alla `byte`-värden är också `short`-värden, och alla `short`-värden är också `int`-värden, etc.
- Vi säger att typen `long` är *större än* typen `int`, som i sin typ är större än typen `short`, etc.
- Java är vad som kallas ett *statiskt typat* språk, dvs kompilatorn försöker kontrollera att vi bara gör saker som säkert kommer att fungera när vi kör programmet.
- Ett exempel på möjligt misstag är att lägga ett värde som kan vara för stort i en variabel:

```
byte smallValue = 25;
int value = smallValue;      // OK
smallValue = value;      // kompileringsfel!
```

Heltalsaritmetik

- `+`, `-`, `*`, `/`, `%`
- Heltalsdivision ger heltalsresultat:
  - kvoten får vi med `/` (värdet är trunkerat)
  - resten får vi med `%`
- `+=`, `-=`, `*=`, `/=`
- `++`, `--`

Reella tal

- De flesta reella tal kan inte representeras exakt i en dator med ändligt mycket minne.
- Räkna därför aldrig med att reella tal är exakta, inte ens så ’enkla’ tal som 0.1.
- Det finns två reella taltyper i Java:
  - `float`: 9 siffrors noggrannhet
  - `double`: 18 siffrors noggrannhet
- Det är i allmänhet meningslöst att undersöka om två reella tal är lika.

Exempel på reella konstanter

```
double x = 1.2;
x = .5;
x = -.42;
double nA = 6.022e23;
double h = 6.626e-34;
```


Problem vid division

Division mellan heltal ger heltal:

```
double value = 1/2;
value = 1.0/2;
value = (double)1/2;
int a = 1;
int b = 2;
double half = (double)a/b;
```


Viktiga reella operationer

- `Math.PI` ger π
- `Math.sqrt(x)` ger $\sqrt{x}$
- `Math.pow(x,y)` ger x^y
- `Math.exp(x)` ger e^x
- `Math.log(x)` ger $\ln x$
- `Math.sin(x)` ger $\sin x$
- `Math.cos(x)` ger $\cos x$
- `Math.abs(x)` ger $|x|$

Omvandling mellan reella tal och heltal

De reella talen en ’större typ’ än heltalen, vi kan därför lagra ett heltal i en reell variabel, men inte tvärtom:

- `double x = 1; // OK`
- `int n = Math.PI; // Fel!`

Omvandling till heltal

Vi kan omvandla ett reellt tal till heltal på olika sätt:

- `(int) 3.8` – trunkerar till närmaste mindre heltal.
- `Math.floor(x)`: som ger närmaste heltal som är mindre än eller lika med `x` – värdet som vi får tillbaka är av typen `double`.
- `Math.round(x)`: som ger närmaste heltal – värdet som vi får tillbaka är av typen `long`.
- Både med `Math.floor` och med `Math.round` måste vi 'typa om' för att lagra värdet i en `int`-variabel:

```
int n = (int) Math.floor(x);
int n = (int) Math.round(x);
```

Logiska värden

- Ett logiskt villkor kan vara antingen sant (true) eller falskt (false).
- Typen `boolean` innehåller precis dessa två värden: `true` och `false`.
- Även den enklaste heltalstypen (`byte`) innehåller 256 olika värden – så `boolean` är minst 128 gånger enklare än någon taltyp!
- Vi kan använda typen `boolean` för att skriva funktioner som avgör om olika saker är sanna eller ej.

Att bryta ner program i mindre delar

- Önsketänk när du programmerar – anropa funktioner och procedurer som du önskar fanns, och skriv dem sedan själv.
- När du skriver dessa underprogram, gör likadant: anropa funktioner och procedurer som du önskar fanns, och skriv dem sedan själv.
- Till slut är underprogrammen så enkla att du kan skriva dem med en gång, de är då ofta bara ca 5-10 rader långa.
- Att skriva program genom att anropa funktioner och procdurer som vi sedan skriver kallas *top-down-design*.

Fördelar med underprogram

- Vi slipper upprepa kod på flera ställen – detta är bra av flera skäl:
  - vi får mindre att skriva,
  - om vi skall ändra något räcker det att vi gör det på ett ställe, istället för på flera,
  - vi slipper risken för slarvfel när vi kopierar kod från ett ställe till ett annat.
- Vi får ett namn på det som skall göras, vilket gör programmet lättare att förstå. Detta gäller även om vi bara vill anropa underprogrammet en gång.
- Vi kan återanvända underprogrammen när vi skriver program i framtiden.

Sammansatta logiska villkor

- Det finns tre vanliga logiska operationer:
  - *och*: $a \wedge b$ skrivs `a && b`
  - *eller*: $a \vee b$ skrivs `a || b`
  - *inte*: $\neg a$ skrivs `!a`