Tema 4: Hashtabeller

Wilhelm Durelius `widu7139`

9 februari 2026

När man implementerar egna klasser som ska användas tillsammans med Javas hash-baserade klasser såsom HashMap och HashSet är hur man implementerar hashCode-metoden avgörande. En bra implementation av hashCode ser till att det går snabbt att hasha, gärna i O(1), medan en dålig implementation kan försämra prestandan rejält.

Det som kan få en implementation av hashCode att bli inkorrekt, är bland annat att hashCode inte genererar samma data för objekt smo bör vara lika varje gång. Är slump inblandat någonstans i kedjan, så kraschar funktionaliteten. En fortsättning på detta är att objekt som är lika, och därför genererar samma hashcode, alltid också måste vara lika enligt equals metoden. Man behöver alltså också implementera equals utöver hashcode metoden för egenskapade klasser som ska använda en hashcode, och utöver det se till att objekt som är lika alltid också är lika enligt både hashCode-metoden och equals-metoden.

Det är därför ett logiskt krav att både equals och hashcode använder samma egenskaper i sin logik, rent praktiskt är

det möjligt att skippa detta krav men ändå få en till synes fungerande hash, men det är skakigt och blir lätt fel. Svårigheten att testa alla edge cases för hashkoderna, speciellt när man jobbar med användarskapad data, är hög. Enhetstesterna kanske ser ut att fungera, men i produktion får du fram fel data eller långsam processering.

Undvika fält som kan förändras vid hashning, låt oss säga till exempel ett objekt som har en 'last_changed'-fält, använder du denna som en del av hashet kommer du inte hitta objektet igen om egenskapen förändras efter insättning.

Vid själva implementeringen av hashCode-metoden, utöver att använda samma egenskaper som equals-metoden, är det vanligaste sättet att börja med ett startvärde och därefter för varje fält som ska ingå i hashningen multiplicera det nuvarande värdet med ett primtal, oftast 17 eller 31, och addera fältets hashvärde. Anledningen till att primtal används är att de inte går att dela med något utom sig själv och 1, vilket förhindrar att olika tal gångrat med primtalet inte får samma resultat. Detta resulterar i bättre spridning av hashvärden, vilket minskar risken för kollisioner i hashtabellen.

För fältet inne i en klass, så behöver man också ofta generera hashkod för själva fälten för att användas i beräkningen i klassen som fälten tillhör. Primitiva datatyper har inbyggda metoder för detta i Java, till exempel sifferklasserna (Integer och Long).

Arrayer är ett annat specialfall som kräver extra uppmärksamhet. Standardimplementation av hashCode för arrayer baseras på minnesadressen snarare än dess innehåll, vilket gör den oanvändbar i de flesta sammanhang. Om ett objekt innehåller arrayer som ska påverka likhet måste man därför använda metoder

som beräknar hashvärdet baserat på innehållet istället.

Ett alternativ till att implementera hashCode manuellt är att använda hjälpmetoder som till exempel Objects.hash. Detta gör koden kortare och mer lättläst, men innebär samtidigt viss prestandaöverhead eftersom metoden använder varargs. I de flesta fall är denna kostnad försumbar, men i prestandakritiska delar av ett system kan en egen implementation vara att föredra.

Slutligen är det viktigt att komma ihåg att en hashfunktion inte behöver vara perfekt, utan tillräckligt bra för sitt användningsområde. För hashtabeller är målet inte att undvika alla kollisioner, utan att fördela objekt jämnt så att kollisioner sker sällan. Genom att följa kontraktet mellan equals och hashCode, undvika föränderliga fält och använda etablerade mönster kan man skapa hashfunktioner som fungerar korrekt och ger god prestanda i praktiken.