1 Comparator & Comparable

1.1 Implementation av Comparable

Se javakoden.

Kommentarer

- Att implementera **Comparable** innebär att man gör objekt av sin klass jämförbara med andra och att det därmed antas existera en naturlig ordning för klassen. (Informellt kan tänkas att objekten ska kunna läggas på rad i storleksordning genom att par av objekt jämförs med varandra.)[1] Denna ordning bör vara en total ordning[2].
- Enligt kontraktet för ett anrop `a.compareTo(b)` ska följande returneras
  1. ett positivt heltal, om a > b
  2. noll, om a == b
  3. ett negativt heltal, om a < b
- Observera hur subtraktion används i `modelYear - c.modelYear` för att skapa ett returvärde från `compareTo()` utan att använda "if-satser"eller liknande konstruktioner.
- Det rekommenderas starkt att metoden `compareTo()` är konsistent med metoden `equals()`, d.v.s. för två objekt *a* och *b* där `a.compareTo(b) == 0` gäller att `a.equals(b) == true` och omvänt. Om detta inte gäller kan det t.ex. inträffa underligheter när objekten lagras i samlingar.

1.2 Implementation av en flexibel(?) Comparator

Se javakoden.

Kommentarer

- För **Comparator** gäller det till stor del att tänka på motsvarande saker som när **Comparable** ska implementeras. (med metoden `compare()` som motsvarighet till `compareTo()`), se därför kommentarerna till föregående uppgift ("Car implements Comparable").
- En komparator tar typiskt in "inställningar" för hur jämförelsen ska gå till i konstruktorn och sparar dessa i sitt inre tillstånd (d.v.s. i instansvariabler). Inställningarna används sedan när jämförelser utförs genom metoden `compare()`.

[1] Se Javas API: `http://download.oracle.com/javase/6/docs/api/java/lang/Comparable.html`

[2] `http://en.wikipedia.org/wiki/Total_order`

2 Strategimönstret

2.1 Numerisk integrering

Kommentarer

Se javakoden.

3 Exceptions

3.1 Propagering av fel uppför anropsstacken

3.1.1 Om vi inte haft exceptions?

Lösning

```
void metod1() {
  do something A;
  success = metod2();
  if (!success) doErrorProcessing;
  else do something B;
  }
}

boolean method2() {
  do something C;
  success = metod3();
  if (!success) return false;
  else do something D;
  }
}

boolean method3() {
  do something E;
  success = readFile();
  if (!success) return false;
  else do something F;
  }
}
```

Kommentarer

- Poängen ligger i att observera hur felet som uppstår i `readFile()` och ska hanteras i `metod1()`, kräver att felhantering även införs i `metod2()` och `metod3()` (för att felet ska kunna propageras uppåt).

3.1.2 Exceptions räddar dagen?

Lösning

```
void metod1() {
  do something A;
  try {
    metod2();
  }
  catch(ReadFileFailedException) {
    doErrorProcessing;
  }
  do something B;
}

boolean method2() throws ReadFileFailedException {
  do something C;
  metod3();
  do something D;
}

boolean method3() throws ReadFileFailedException {
  do something E;
  readFile();
  do something F;
```

Kommentarer

- Felhanteringen som tidigare gjordes ”manuellt” med if-satser kan nu hanteras med exceptions. Det gör att felhanteringen dels blir tydligare (if-satser kan ju handla om annat än fel) och dels kan tas bort från kod som inte har med felet att göra. Innehållet i både `metod2()` och `metod3()` blir nu mycket ”renare”. Metoderna uppmärksammar att något kan gå fel under körningen genom sina `throws`-deklarationer, men behöver inte hantera felet själva.

- Använd endast exceptions för sådant som är just ”undantag” från kodens huvudsakliga uppgift, d.v.s. (i första hand) olika typer av fel. *Om det inte*

finns någon exekveringsväg i ett program som är helt fri från exceptions så är de inte just undantag.

- Låt aldrig ett `catch`-block stå tomt! Då kan ett fel inträffa utan att det upptäcks.
- Använd lämplig typ av exceptions beroende på kodens abstraktionsnivå. Låt t.ex. inte ett **IOException** kunna fångas i kod där programmeraren inte vet om att någon input/output inträffar. I ett sådant fall bör detta **IOException** fångas i den kod som handhar input/output och en mer lämplig typ av exception kastas vidare.
- Fånga aldrig ett **Exception** (alltså basklassen för alla exceptions). Det har en benägenhet att dölja fel. Fånga bara de exceptions du kan förvänta dig.

3.2 Arv av exceptions

3.2.1 Vad fångas I

Kommentarer

- Det fångas eftersom ett **NotAPositiveNumberException** också är ett **NotANumberException** (genom arvsrelationen).

3.2.2 Vad fångas II

Kommentarer

- Koden kompilerar inte eftersom `catch(NotAPositiveNumberException napne)` inte kan nås (”unreachable code”). Av samma anledning som i föregående uppgift fångas alla **NotAPositiveNumberException** i `catch(NotANumberException nane)`.

4 (Icke-)muterbarhet

4.1 Publika instansvariabler?

Kommentarer

- Vi har minst två skäl att undvika publika instansvariabler:
  - De kan tillåta andra objekt (av andra klasser) att manipulera vår interna representation direkt. Som en konsekvens blir det svårt att garantera att eventuella klassinvarianter håller.
  - Den faktiska implementationen exponeras. Detta medför att vi vid ett senare tillfälle inte har möjlighet att ändra vår implementation om vi skulle behöva; typen hos variabeln blir en del av klassens *externa* kontrakt.

I ett fåtal fall kan det ändå vara rimligt att ha publika instansvariabler. Dessa måste då vara deklarerade `final`.

I **Pair**-klassen kan man motivera det med att det som exponeras är just värdena; **Pair** har *ingen* implementationsdetalj eller klassinvariant som är beroende av vilka *värden* de olika delarna av paret har. Den enda invariant vi formulerat kan vi garantera genom `final`.

4.2 Vad gör final?

Kommentarer

- Att deklarera en variabel `final` innebär endast att själva referensen inte kan ändras (efter första tilldelningen). *Tillståndet för objektet referensen hänvisar till kan dock ändras.*

  Man kunde tänka sig att detta problem skulle kunna lösas (garantera att *Pair* var strikt icke-muterbar) genom kloning. Typparametrarna skulle då istället bli <**A extends Cloneable, B extends Cloneable**>.

  Då klassen främst är till för att "lösa" problemet att man inte kan returnera tupler[3] i Java vill man inte skapa för många nya objekt i onödan; att vi behöver skapa det omgivande **Pair**-objektet är illa nog.

  Ett betydligt allvarligare problem är att man kan tänka sig flera innebörder av att vara **Cloneable** innebär. Om kopian är av typen "djup" eller "grund" spelar stor roll om man vill uppnå icke-muterbarhet. Se `http://en.wikipedia.org/wiki/Object_copy#Deep_vs._Shallow_vs._Lazy_copy` för en förklaring av grund kontra djup kloning.

  Kontentan är att vi kan garantera icke-muterbarhet *om och endast om objekt av de klasser vi har som typparametrar också är strikt icke-muterbara.*

  Klassinvarianten blir "*first* and *second* will always have the same values as were passed to the constructor". Detta säger inte att klassen i strikt bemärkelse är icke-muterbar men att den parametriserad över "rätt" typer skulle kunna vara det.

[3] `http://en.wikipedia.org/wiki/Tuple`