Projektbeskrivning

Virtual Machine

2018-03-24

Projektmedlemmar:
Jesper Sporron <jessp088@student.liu.se>

Handledare:
Teodor Riddarhaage <teori199@ida.liu.se>

Table of Contents

Planering

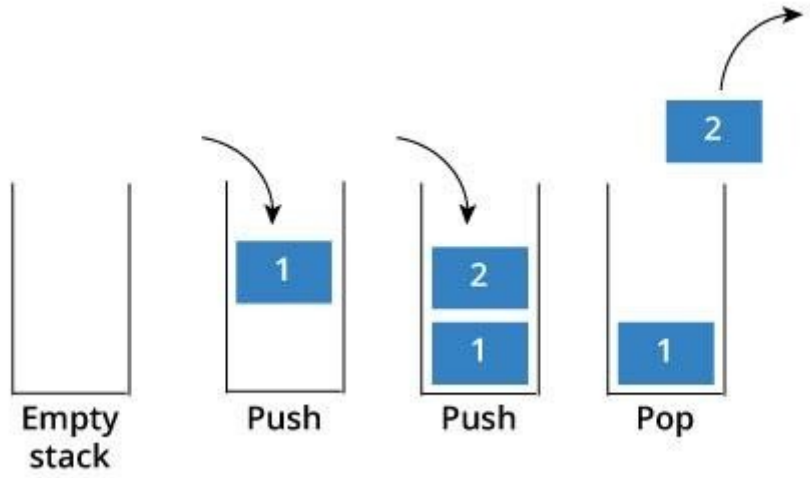
Introduktion till projektet

Jag ska utveckla ett simpelt Assembly-liknande programmeringsspråk med tillhörande ”debugger” som skall kunna användas för att stega igenom programmet rad-för-rad och se hur värden tolkas och sparas m.m.

Ytterligare bakgrundsinformation

Stack

Programmeringsspråket är stack-baserat. Detta betyder att alla operationer (addera, multiplicera, m.m.) utförs på den s.k. *data-stacken*. Data stacken fungerar enligt LIFOprincipen, d.v.s. Last In First Out vilket betyder att man endast har åtkomst till det senaste objektet man lade in i stacken. Här är en användbar bild för att förstå stacks:



Figur 1: Stack principen. Taget från https://www.programiz.com/dsa/stack

Tokenizer / Lexer

För att kunna tolka texten behöver man på något sätt dela upp den i olika ”tokens” som betyder olika saker. Detta görs ofta under det s.k. ”tokenizer / lexer steget”. En väldigt simpel lexer skulle kunna ta en bit text (*code*) och dela upp det i tecken separerat med mellanslag (*code.split(” ”)*) där varje element i den resulterande listan är ett token. Vill man däremot ha lite mer avancerad grammatik måste man använda sig av andra metoder – till exempel Regex (regular expressions). Wikipedia (https://en.wikipedia.org/wiki/Lexical_analysis#Token) har en mycket bra visualisering på hur detta kan se ut.

Regex

Regex står för ”Regular Expression” och är ett mönster som används för att jämföra text. Detta är användbart i många operationer så som hitta operationer (CTRL-F i din webbläsate till exempel) men i detta projekt används det huvudsakligen för att känna igen om en viss bit kod representerar en instruktion, variabel, siffra osv.

Milstolpar

#	Beskrivning
1	Implementera ett Stack-objekt som fungerar enligt bild ovan. Stack objektet skall minst ha funktioner för att kunna lägga på ett objekt och ta bort det översta objektet.
2	Unit test för Stack-objektet
3	Det skall finnas ett ”Lexer” objekt som kan läsa in text och dela upp texten radvis, och kunna dela upp raderna i olika tecken separerade av mellanslag (ex. ”add 1 2” skall delas in i tecknen ”add”, ”1”, ”2”).
4	Programmet skall kunna gå igenom alla rader och tecken ovan och kunna bygga upp ett ”Instruction” objekt som innehåller instruktion (”add”) och operander (”1” och ”2”).
5	Unit test för ”Lexer” och ”Instruction”
6	Ett ”VirtualMachine” objekt som tar in en lista instruktioner och utför dem i ordning.
7	Skapa den första riktiga instruktionen ”LITERAL” som lägger på ett konstant värde till stacken (LITERAL 5 -> push(5)).
8	Unit test för ”VirtualMachine” som ser till att ”LITERAL” fungerar.
9	Skapa instruktioner för de fyra räknesätten (ADD, SUB, DIV, MUL) och modulus (MOD) som tar bort de 2 översta värdena på stacken och pushar resultatet
10	Unit test för de nya instruktionerna
11	Skapa en logisk instruktion som används för att jämföra två tal (CMP) och diverse jump-instruktioner som används för att hoppa mellan olika bitar i koden.
12	Unit test för jump-instruktionerna.
13	Skapa instruktioner som används för att spara (STORE) och ladda (LOAD) variabler med värden.
14	Unit test för variabel instruktionerna
15	Börja implementera debugger applikationen. Det skall finnas en textruta som kommer innehålla koden och en lista som innehåller alla variabler. Det skall också finnas en menyrad som används för att öppna filer som innehåller kod och för att stega igenom koden rad för rad. **Endast utseende i detta steg.**
16	Implementera funktionalitet för att kunna ladda in ett program från en fil.
17	Implementera öppna-fil funktionaliteten i menyraden samt visa programmet i textrutan.
18	Implementera körfunktionen i debuggern så man kan köra koden man laddat in (**ej** stega igenom ännu).
19	Implementera stegvis exekvering av koden

20 Implementera variabelvisaren så man ser vad alla variabler har för värden och vilka värden som ligger på stacken.

21 Implementera ”breakpoints” så man kan köra koden upp till en viss position.

22 Utöka ”Lexer” objektet så den känner igen ”Labels” som kan användas för att namnge rader för att lättare hoppa till dem.

23 Utöka ”Lexer” objektet så den verifierar att koden du skrev är grammatiskt och logiskt korrekt (t.ex. att du har rätt antal parametrar).

24

Övriga implementationsförberedelser

VirtualMachine

Har hand om själva exekveringen av instruktionerna. Innehåller en referens till stacken samt den nuvarande kodraden. Måste tänka på att ej hårdkoda instruktionerna i denna klass utan att ha instruktionerna som sina egna klasser så det är lätt utökbart.

Lexer

Har hand om att göra om text till instruktioner samt verifiera att instruktionerna ej givits fel antal parametrar och liknande fel. Börja simpelt med whitespace-separerade kommandon och gå över till ett Regex-baserat system efter jag läst på lite om regex.

Saker att tänka på

Hur kan jag på snyggaste sätt få Lexer att förstå att *LITERAL 5* skall översättas till *new Literal(arguments[0])* utan att ha massor med *if (instruction.equals(”LITERAL”))* satser o.s.v.?

Utveckling och samarbete

Jag är ensam i projektet så jag tänker jobba när jag har tid över på kvällen samt på labbtillfällen. Jag har höga ambitioner med projektet då jag har programmerat mycket sedan innan och vill använda detta för att bättra på min tidigare kunskap. Jag satsar därför på en 5:a.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

Alla milstolpar är helt implementerade.

6.2. Dokumentation för programkod, inklusive UML-diagram

Programmet är huvudsakligen indelat I tre delar; Lexing, Parsing, och Execution. Varje del ansvarar över en liten del av programmet och kopplas sedan ihop för att uppnå det sista resultatet.

Lexing

Lexing också kallat *Tokenizing* eller på svenska *tokenisering* ansvarar för att ta en bit text som indata och dela upp texten i mindre självständiga delar kallade ”tokens”. Som ett exempel på tokenisering kan man dela upp denna mening ”`I would like 3 burgers, please`” till dessa tokens (representat som XML):

```
<sentence>
      <word>I</word>
      <word>would</word>
      <word>like</word>
      <number>3</number>
      <word>burgers</word>
      <delimiter>,</delimiter>
      <word>please</word>
</sentence>
```

eller denna C-kod ”`x = a + b * 2;`” till dessa tokens (exempel från Wikipedia):

```
[(identifier, x),(operator, =),(identifier, a), (operator, +),
(identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

Parsing

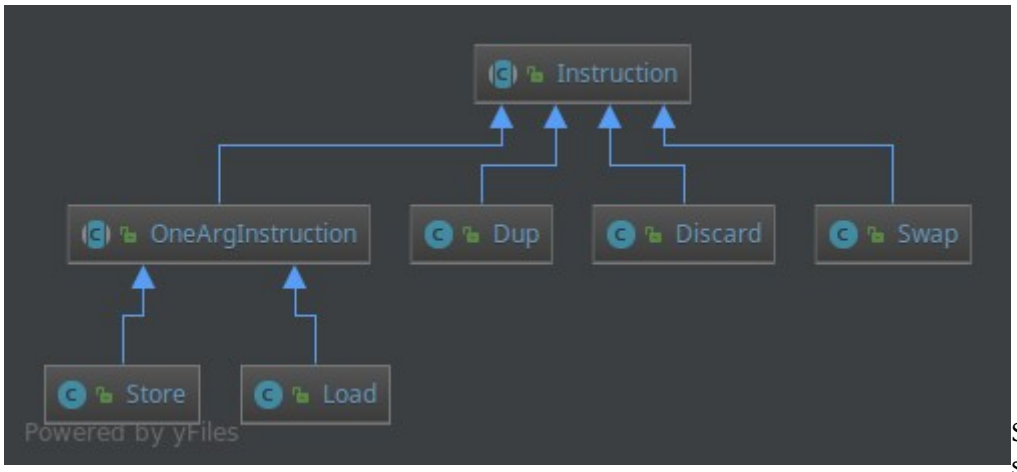
Parsing också kallat *syntaxanalys* är nästa steg efter tokenisering. I detta steg tar man in de tokens som tokeniseraren har producerat och matchar dessa tokens mot en grammatik som definierar hur språket ska vara strukturerat och skapar instruktioner utifrån detta. Det finns ett flertal olika typer av grammatiker och typer av analyserare. I mitt fall har jag valt en simpel LL(1) grammatik, vilket (simpelt beskrivet) betyder att analyseraren endast behöver veta nästa token för att kunna tolka grammatiken.

Execution

Executing eller *exekvering* steget är det sista steget. I detta steg tar ett VirtualMachine objekt in de instruktioner som parser steget skapade. Detta är det enda steget som användaren kan påverka och detta görs genom användargränssnittet.

Instruction

Instruction objekten är en viktig del utav programstrukturen. De ärver alla från en abstrakt klass kallad *Instruction*, eller från en annan abstrakt klass kallad *OneArgInstruction* (som själv ärver från Instruction). Här är ett exempel på relationen kan se ut:



Som kan ses på bilden så ärver *Discard* direkt från *Instruction*, för Discard behöver ej en inparameter medans *Store* ärver från *OneArgInstruction* då den behöver en inparameter. Det finns också en separat klass för alla ”jump” instruktioner. Alla instruktioner och exakt hur de fungerar kan avläsas från denna tabell:

Instruction	Operands	Description	Operation (Stack and PC)
Arithmetic Instructions			
LITERAL	c	Push Constant to Stack.	…, → …, c
ADD		Push Sum of Values.	…, L, R → …, L+R
SUB		Push Difference of Values.	…, L, R → …, L–R
DIV		Push Quotient of Values.	…, L, R → …, L/R
MUL		Push Product of Values.	…, L, R → …, L*R
MOD		Push Remainder of Values.	…, L, R → …, L%R
Control Flow Instructions			
CMP		Compare Top Values in Stack.	…, L, R → …, L-R
JMP	c	Unconditional Jump.	PC ← c
JMPGT	c	Jump if Greater Than Zero.	…, val → …, if(val > 0) PC ← c
JMPLT	c	Jump if Less Than Zero.	…, val → …, if(val < 0) PC ← c
JMPEQ	c	Jump if Zero.	…, val → …, if(val = 0) PC ← c
JMPNE	c	Jump if Not Zero.	…, val → …, if(val != 0) PC ← c
CALL	c	Subroutine Call.	…, → …, PC PC ← c
BREAK		Return from Subroutine.	…, ptr → …, PC ← ptr
RETURN		Return from Subroutine with Value.	…, ptr, val → …, val PC ← ptr
Data Transfer Instructions			
DUP		Duplicates Top of Stack.	…, val → …, val, val
SWAP		Swaps Top Values of Stack.	…, L, R → …, R, L

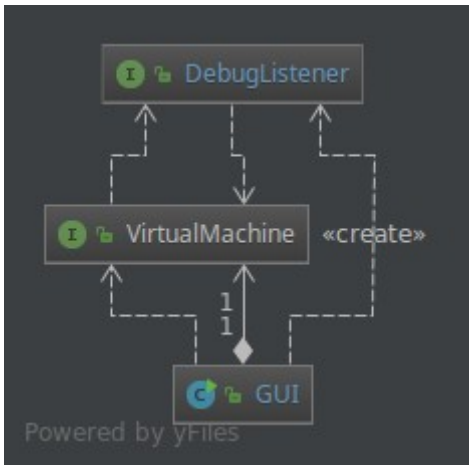
LOAD	c	Loads Variable to Top of Stack. The operand ***c*** must be an unsigned integer.	..., → ..., val
STORE	c	Stores Top of Stack to Variable. The operand ***c*** must be an unsigned integer.	..., val → ...,

Meta Control Instructions

MALLOC	c	Increases the Stack Size by ***c***.	
FREE	c	Reduces the Stack Size by ***c***.	
SIZETO	c	Sets the Stack Size to ***c***. All values outside of the new Stack size is **lost forever**.	

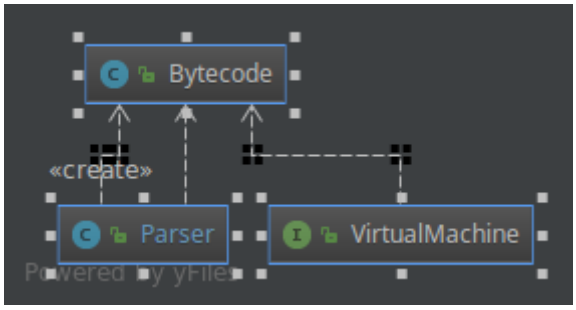
Listener

För att användaren ska kunna kolla vad som händer I *VirtualMachine* finns det en händig *DebugListener* klass som används för att lyssna på händelser I *VirtualMachine* – till exempel att en instruktion har börjat. Denna används av klasser så som *GUI* för att uppdatera tabeller eller pausa exekveringen:



Bytecode

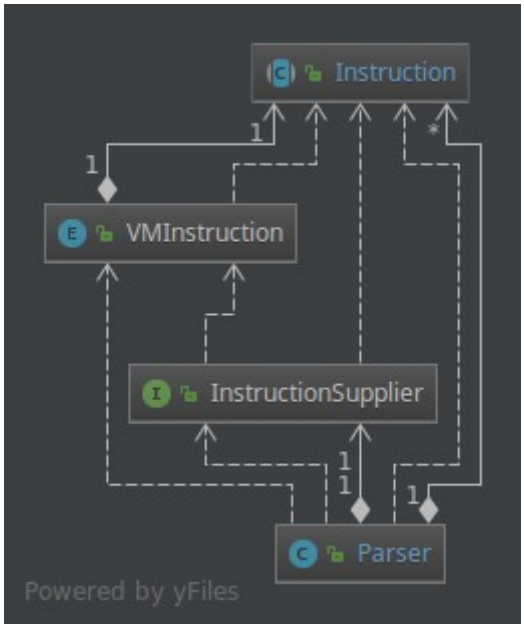
Parser har inte bara hand om att skapa instruktioner. Den har också hand om att hålla reda på vilken rad kod som tillhör vilken instruktion. Detta betyder att relationen mellan Parser, Bytecode och VirtualMachine egentligen ser ut såhär:



Där alltså Parser skapar ett Bytecode objekt och VirtualMachine sedan exekverar instruktionerna I detta objekt samt använder sig utav rad-till-instruktions datan för att hantera breakpoints.

Instruction Supplier

För att Parser ska kunna ta in en rad text och kunna omvandla det till en instruktion använder den sig utav en s.k. *InstructionSupplier*. Denna tar in en isntruktionstyp *VMInstruction* samt argument och skapar ett passande *Instruction* objekt.



6.3. Användning av fritt material

Jag har använt mig utav ***JUnit*** för unit-tests och jag har använt mig utav ***Logback*** för att logga. **På grund av att mitt projekt är Gradle-baserat bifogas ej dessa som jar filer** utan importeras automatiskt av Gradle när man importerar projektet som ett Gradle-projekt I IntelliJ eller Eclipse.

6.4. Användning av objektorientering

1. Inkapsling. Inkapsling är viktigt I *Instruction* då den innehåller metoder som endast dess subklasser borde kunna använda (t.ex. *push* som lägger på värden på stacken). Därför är dessa *protected* så subklasserna kan komma åt dem men inte utomstående klasser, för om utomstående klasser kallade på dessa metoder kan de lägga till eller ta bort värden mitt under exekveringen vilket skulle förstöra programmet. I ett icke-oo språk skulle man helt enkelt behövt ”markera” att dessa metoder ej borde användas (detta är vanligt I t.ex. Python där man använder ”_” för att markera att detta är en ”privat” metod) men hade inte haft någon garanti på att det respekteras. Detta är betydligt sämre än oo-sättet då det inte är lika säkert, men det har också sina fördelar för om en avancerad användare som vet vad denna gör skulle vilja använda dessa metoder av en riktig anledning så går inte detta med min metod.
2. Konstruktorer. Jag har olika konstruktorer I *Instruction* och *OneArgInstruction* då *OneArgInstruction* är menat att ta ett argument och *Instruction* är menat att inte ta några alls. På så sätt ser jag till att rätt instruktion får rätt antal parametrar. Utan objektorientering skulle jag istället ha tagit en array med värden och behöva kolla manuellt att den innehåller rätt antal parametrar. Den största fördelen med det objektorienterade tillvägagångssättet är att det automatiskt blir fel direkt I kompilatorn om jag t.ex. försöker skicka in 2st argument till en instruktion som endast kräver ett argument. Det icke-oo sättet skulle uppnå samma resultat men det skulle hända när programmet körs, inte statiskt innan.
3. Typheiarkier. *VirtualMachine* klassen tar in en lista av *Instruction* objekt som den kan exekvera. Själva *VirtualMachine* objektet behöver inte veta om vilken instruktion den kallar på, den kallar bara på *Instruction#process* och sedan tar *Instruction*s subklasser hand om den faktiska implementeringen. Utan OO-språk skulle man kunnat ha alla instruktioners kod direkt kopplat till *VirtualMachine* objektet som då skulle kunna kolla typen av instruktion och utföra den korrekta handlingen. T.ex. om den såg att

instruktionen var *Add* så skulle den veta att den ska addera två tal. OO-sättet är starkt att föredra för om det skulle visa sig att jag vill lägga till ännu en instruktion så är det bara att lägga till en ny klass. Jag behöver inte ändra I *VirtualMachine* eller I resten av logiken alls.

4. Overriding. Alla instruktioner är subklasser till *Instruction* och måste overrida metoden *process*. I denna metod utför subklasserna vad de ska göra när *VirtualMachine* kallar på denna *process* metod. T.ex. så adderar *Add* klassen två tal I sin *process* metod. Utan objektorientering skulle, precis som ovan, *VirtualMachine* behövt tagit hand om och ”vetat” hur den skulle hanterat de olika instruktionerna. Att ha metoder som subklasser skriver över är att föredra I detta fall för då inkapslar man beteendet hos en instruktion till själva instruktionen och då slipper man knyta en instruktion så starkt till *VirtualMachine* klassen.

6.5. Motiverade designbeslut med alternativ

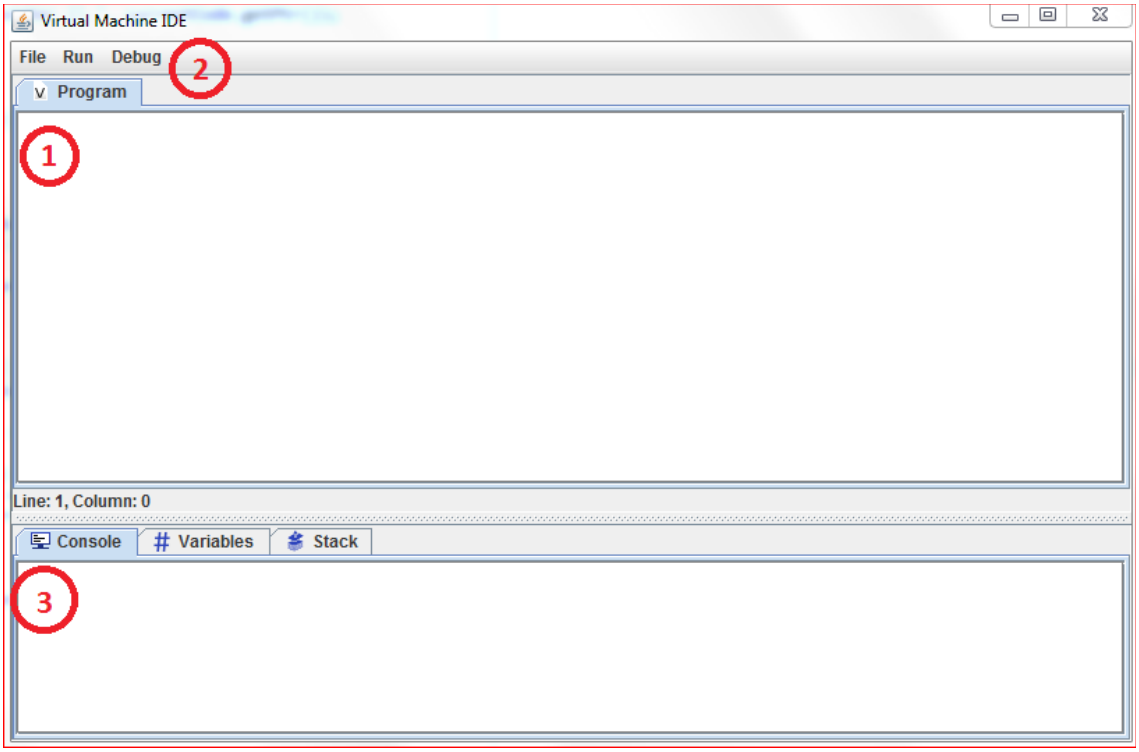
1. **Recursive-descent eller LL(k)?** När jag läste på om hur gramatiker och syntaxanalys fungerade och hur man skulle implementera dessa. Jag ville ha en lätt grammatik och ett lätt sätt att analysera det på vilket ledde mig till Recursive-descent och LL(k) grammatik. I slutändan valde jag en LL(k) grammatik för när jag började skriva på en skiss lösning så var det detta som jag tyckte var lättast. Min impementation av LL(k) grammatik kan hittas I *Parser* klassen.
2. **Klasser för varje instruktion?** I början var jag osäker om det var bra att ha klasser för varje instruktion men när det var dags att ta ett beslut valde jag detta tillvägagångssätt. Nu I efterhand tycker jag att det var en bra idé att ha varje instruktion som en separat klass för det gjorde det lätt att expandera och lägga till ytterligare instruktioner genom att bara extenda från *Instruction* klassen. Den alternativa lösningen som jag funderade över var att spara instruktioner och argument I en enda lång lista av heltal där ett visst nummer betyder en viss instruktion (ex 3 kanske betyder *Add*) och på så sätt kunna mer kompakt representera instruktioner och argument. Detta skulle betytt att all logik skulle legat I *VirtualMachine* vilket var någonting jag strävade efter att undvika.
3. **Signalera vilket tillstånd VirtualMachine är I**. När jag föst började skriva på VirtualMachine klassen var det svårt att veta exakt vilket tillstånd den befann sig I. Det var svårt att tyda om den var pausad eller om den var stoppad, och om den var stoppad för att det blev fel eller för att den helt enkelt var klar. Jag löste detta genom att lägga till en enum *VMState* som innehåller värden för de olika tillstånd som *VirtualMachine* kan befinna sig I. Detta gjorde det lättare att direkt kunna se vilket tillstånd den befann sig I genom att t.ex. kolla *vm.getState() == VMState.END_USER* för att se om användaren avslutade exekveringen. En alternativ lösning skulle vara att ha flera *boolean*s som håller reda på de olika fallen (pausad, avslutad, användaren avslutade, pausad p.g.a. breakpoint osv). Jag känner att den lösning jag valde är betydligt mer elegant då den är tydligare I vad den representerar och det är simplare att använda istället för att behöva kolla massor med boolean värden.
4. **Listener klass.** Jag valde att ha en listener klass som kan notifiera utomstående om något viktigt har hänt I *VirtualMachine* objektet. Detta gjorde jag för att simpelt kunna påverka flödet av exekveringen utan att behöva meka med det interna I *VirtualMachine*. Med hjälp utav listener objektet kan man kolla om VirtualMachine exekverar en viss bit kod och I så fall stoppa exekveringen för att simulera breakpoints eller att stega igenom kod. Det andra alternativet jag övervägde var att exponera vissa interna saker hos *VirtualMachine* till andra klasser, som t.ex. att det var

första gången den körde en viss bit kod. Detta skulle gett samma resultat I slutändan men skulle blivit fulare för användaren och varit sämre för prestandan då man hela tiden behövt kolla om ett visst tillstånd blivit uppnått *if (virtual_machine_just_started)* istället för att bara vänta på att listener säger åt dig.

5. **Hantera breakpoints internt eller överlåta det till användaren.** Detta beslut går hand-i-hand med Listener beslutet ovan. Efter att ha implementerat listener ville jag först att breakpoints skulle hanteras av användaren (med hjälp av listener) men eftersom breakpoints är så viktigt I debugging så valde jag att istället hantera det internt för att 1) få mer prestanda och 2) ha en konkret implementation som användaren kan luta sig på istället för att behöva komma på sin egen.
6. **Instruction Supplier.** Jag hade ett problem med att Parser var överdrivet kopplat med vilka instruktioner den kunde tolka och inte. För att lösa detta implementerade jag en klass som kunde ta namnet på instruktionen samt ett visst antal argument och skapa ett *Instruction* objekt. Detta löste problemet att *Parser* var väldigt kopplat med *Instruction* klasserna och gjorde det möjligt att göra suppliers som inte hade alla instruktioner (för att testa vissa specifika instruktioner).
7. **Programmet stöder endast siffror.** Jag tog tidigt beslutet att mitt program endast skulle stödja siffror och ej strängar eller andra typer av objekt. Jag tog detta beslut för att hålla programmet simpelt och lätt att utveckla. Om jag skulle valt att stödja flera typer av objekt skulle jag behövt spara objekten på ett annat sätt än vad jag gör nu; möjligtvis I en *Value* klass som innehåller vilken typ av värde samt en array av bytes som representerar det värdet I byte-form. Jag kände att jag tog rätt beslut för det höll programmet simpelt, om än mindre flexibelt.
8. **Stack-baserat språk.** I början var jag tvungen att bestämma om det språk jag utvecklade skulle vara ett stack-baserat språk som JVM eller ett register-baserat språk som x86. Det viktigaste för mig var att det var vilket som gick snabbast att utveckla och tolka med en Parser. I slutändan valde jag ett stack-baserat språk för det var lättare att tolka med min f.d. parser och jag hittade mycket JVM kod som jag kunde använda för att lära mig hur man ”tänker” när man ska skriva kod för stack-baserade språk. Alternativet hade såklart varit att ha register istället för att göra alla uträkningar på stacken vilket hade ändrat hur alla instruktioner såg ut och hur jag tog in argument. Det fanns inget direkt bästa val utan jag gjorde det val jag gjorde helt utifrån vilka läroresurser som fanns tillgängliga vid det tillfället.

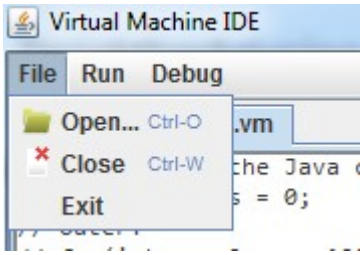
7. Användarmanual

Programmet startas genom att köra *main* metoden i *GUI.java* filen. När programmet startats kommer ni bemötas utav ett fönster med titeln *Virtual Machine IDE* som ser ut som nedan.



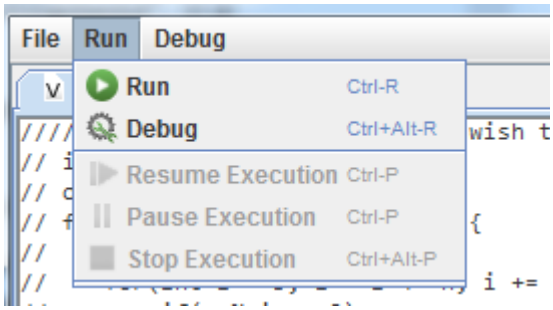
I detta fönster finns ett antal fokuspunkter som är viktiga att kunna:

1. **Programfönster**. I detta fönster kommer all kod att visas upp. Man kan använda detta fönster för att både visa och redigera kod direkt i programmet.
2. **Menyrad**. Menyraden används för kommandon som t.ex. att öppna en ny fil eller köra programmet man har öppnat.
    a. **File**.



- **Open...** öppnar en ny fil.
- **Close** stänger det nuvarande öppna fönstret.
- **Exit** stänger programmet.

a. **Run**.

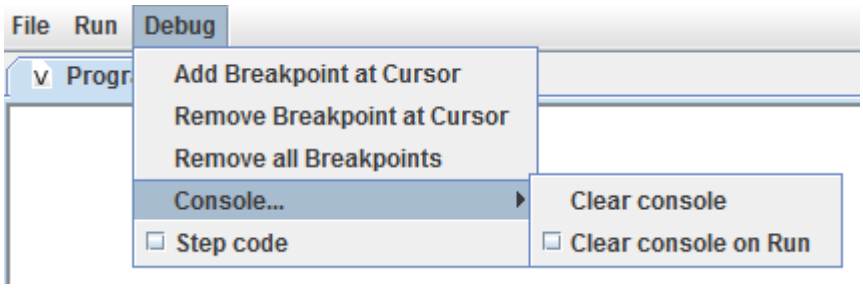


- **Run...** kör programmet i det nuvarande öppna fönstret.
- **Debug...** debuggar programmet i det nuvarande öppna fönstret vilket

tillåter breakpoints.

- **Resume Execution** återupptar exekvering av kod om den var pausad sedan innan. Knappen börjar avaktiverad och aktiveras när koden är pausad.
- **Pause Execution** pausar exekvering av kod om den körs just nu. Knappen börjar avaktiverad och aktiveras när koden körs.
- **Stop Execution** avslutar exekvering av kod om den är pausad eller körs just nu. Knappen börjar avaktiverad och aktiveras om koden är pausad eller om den körs just nu.

a. **Debug**.



- **Add Breakpoint** lägger till en breakpoint på samma rad som markören. En breakpoint pausar exekvering av kod när koden når breakpointen. Breakpoints visas ej grafiskt på skärmen utan användaren måste själv hålla reda på var denna har placerat breakpoints. **Notera att breakpoints delas av alla program!**
- **Remove Breakpoint** används för att ta bort en breakpoint på samma rad som markören.
- **Remove all** tar bort alla breakpoints.
- **Clear Console** tar bort alla meddelanden i konsolen (se punkt 3).
- **Clear Console on Run** tar bort alla meddelanden I konsolen när användaren kallar på **Run** eller **Debug**.
- **Step code** aktiverar stegvis exekvering av koden. Detta betyder att koden kommer pausas efter varje instruktion och vänta på att användaren klickar **Resume Execution** (se ovan). Tryck igen för att avaktivera.

1. **Hjälpfönster.** Det finns ett flertal hjälpfönster som ger användaren information om vad som händer i programmet.
    a. **Console**. Programmets konsol. Här skrivs alla debug- och errormeddelanden.
    b. **Variables**. En tabell med två kolumner. *Variable* visar variabelns index och *Value* visar variabelns värde.
    c. **Stack**. Visar stacken under programmets gång.

8. Slutgiltiga betygsambitioner

Jag siktar på betyg 5 och önskar komplettera denna rapport så snart som möjligt.

9. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
    - Själva programmeringen och beslutsfattandet gick mycket bra då jag var ensam och då kunde bestämma vad jag ville. Det som gick mindre bra var att hålla rapporten uppdaterad efter tidens gång.
- *Har ni lagt ned för mycket/lite tid?*
    - Jag känner att jag har lagt ner precis lagom med tid på själva programmeringen. Däremot underskattade jag tiden det skulle ta att skriva denna rapport...
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Projektbeskrivningen var en bra mall för att veta vad man ungefärligt skulle ha med I det slutgiltiga resultatet och det var bra att ha så man kunde kolla tillbaka på det då och då och se om man uppnått det man själv skrev ner att man skulle uppnå.