Laborator Nr4

Tema : Debugging, profiling, logging.

Scopul: experimentarea unui mecanism **complementar** testarii, pentru detectarea bug-urilor si validarea unui program precum si familiarizarea cu instrumente de debugging, studiul performantei si analiza corectitudinii codului.

A se susține pe 09/11/2020 **, orice întârziere se taxează cu -2 de la nota primita.**

Subiecte atinse:

- *logging*
- pachetul `log4j`
- debugging (Eclipse)
- profiling (TPTP)
- thread dumps
- static checking (JLint & AntiC)

Logging

Logarea reprezinta inregistrarea de informatii din timpul rularii programelor: starea sistemului la un moment dat, aparitia unui eveniment etc. Cu totii am folosit o forma primitiva de logare: instructiunea `print` pentru a afisa valoarea unei variabile in diferite puncte din program.

Logarea este folosita cu dublu **scop**:

- ***debugging***: urmarirea fluxurilor de control si date ale programului
- ***auditing***: desprinderea unor informatii utile din evolutia sistemului, ca, de exemplu, utilizatorul care a executat o anumita actiune

Experienta demonstreaza ca, in numeroase cazuri, logarea se doveste mai utila decat utilizarea unor tool-uri de ***debugging***, deoarece se poate accesa informatia care intereseaza, in momentele relevante din executia programului, evitand activitatile mai complexe, specifice procesului de *debugging*, cum este avansul instructiune cu instructiune. Logarea se dovedeste si mai utila in situatia in care *debugging*-ul este foarte anevoios; este cazul aplicatiilor *multithreaded* sau distribuite. **Mesajele** de logare pot fi afisate la consola, scrise intr-un fisier, sau chiar trimise pe retea.

Desigur, exista si **dezavantaje** ale acestui procedeu:

- **diminuarea vitezei de rulare**
- **dificultatea urmaririi output-ului**, daca este prea detaliat.

log4j

`log4j` reprezinta un mecanism de logare pentru Java. Il puteti obtine de aici. Gasiti o descriere introductiva la aceasta adresa. Pachetul exista si in variante corespunzatoare altor limbaje: `log4cxx`, `log4php` etc.

Elemente de baza

`log4j` ofera posibilitatea de a:

- defini ***logger*-e**, obiecte ce realizeaza logarea. Pot fi **ierarhizate**, pe baza numelui acestora
- asocia **prioritati** mesajelor de logare si a stabili un **prag** al acestor prioritati, astfel incat sa se genereze doar mesajele a caror prioritate depasesc pragul
- asocia unul sau mai multe ***appender*-e** cu un *logger*, acestea stabilind **destinatia** output-ului: consola, fisier, retea etc.
- stabili **formatul** mesajelor de logare, folosind ***layout*-uri**
- defini **fisiere de configurare**, ce manipuleaza comportamentul de logare la rulare, fara modificarea codului sursa

Iata un prim exemplu de program:

```
import org.apache.log4j.*;

public class Test {
    static Logger logger = Logger.getLogger(Test.class); //sau Logger.getLogger("Test");

    public static void main(String[] args) {
        BasicConfigurator.configure();
        logger.info("Hello");
    }
}
```

Se observa:

- directiva `import`. De asemenea, trebuie sa aduagati fisierul `log4j-1.2.15.jar` la `CLASSPATH`
- clasa `Logger` si metoda statica `getLogger`. Aceasta primeste ca parametru **numele** *logger*-ului, care, cel mai adesea, este numele clasei care il contine
- apelul metodei `info`, care logheaza mesajul dat ca parametru, semnaland **prioritatea** `INFO`. Alte prioritati sunt: `DEBUG` < `INFO` < `WARN` < `ERROR` < `FATAL`. Pe langa acestea mai exista `ALL` si `OFF`, cel din urma folosindu-se cand se doreste dezactivarea logarii pentru mesajele asociate unui anumit obiect *logger*
- apelul `BasicConfigurator.configure()`, care ii asociaza *logger*-ului un *appender* de consola. Este suficienta o **unica** invocare pentru intreaga aplicatie. Aceasta are loc, de obicei, in metoda `main`.

Output-ul default este:

```
0 [main] INFO Test - Hello
```

avand semnificatia:

```
durata_de_la_inceputul_executiei [thread] prioritate nume - mesaj
```

Metoda `getLogger` reflecta 2 **design patterns**:

- ***factory***: produce obiecte pe baza parametrului
- ***singleton***: pentru acelasi nume, returneaza intotdeauna **aceeasi referinta**

Prioritati (log levels)

Prioritatile, mentionate mai sus, sunt asociate, de obicei, urmatoarelor **situatii**:

1. `FATAL`: esec ce conduce la **terminarea** aplicatiei. Exemplu: imposibilitatea de a incarca un modul al aplicatiei.
2. `ERROR`: eroare ce permite **continuarea** aplicatiei, fiind asociata, de obicei, unei exceptii Java. Exemplu: esecul conectarii la baza de date. Legatura poate fi restabilita ulterior.
3. `WARN`: probleme **minore**, externe aplicatiei. Exemplu: furnizarea unor date incorecte de catre utilizator
4. `INFO`: eveniment din fluxul **normal** de executie a aplicatiei. Exemplu: citirea parametrilor de configurare
5. `DEBUG`: informatie de o granularitate foarte **fina**. Exemplu: valoarea unei variabile la un anumit moment

Pragul de prioritate (*log level*) **inhiba** generarea mesajelor cu prioritate **inferioara**. Daca, spre exemplu, am adauga secventa:

```
logger.setLevel(Level.FATAL);
logger.info("Hello");
```

mesajul `Hello` nu ar mai fi afisat, deoarece nivelul `INFO` este inferior celui `FATAL`.

Se poate construi o **ierarhie de *logger*-e**, botezandu-le cu nume separate prin caracterul ".", ca in cazul pachetelor si claselor Java. De exemplu, *logger*-ul `idp.gui` este parinte pentru *logger*-ul `idp.gui.Gui`. Exista un *logger* parinte, universal, ce poate fi obtinut prin metoda `getRootLogger`. Relatia ierarhica se refera la **mostenirea**:

- tuturor ***appender*-elor** de la toti stramosii
- **pragului de prioritate** de la parintele direct

Appender-e si layout-uri

Un ***appender*** reprezinta o **destinatie** posibila a mesajului de logare, iar un obiect *logger* poate poseda **oricate** *appender*-e. Exemple:

- `ConsoleAppender`: pentru afisarea la consola
- `FileAppender`: pentru scrierea in fisier
- `RollingFileAppender`: permite generarea unor fisiere de log de o **dimensiune maxima**, urmand ca, la depasirea acesteia, sa se comute la alt fisier de log. De asemenea, permite precizarea **numarului maxim** de astfel de fisiere, ce se doresc pastrate.
- `JDBCAppender`: pentru logare intr-o baza de date
- `SocketAppender`: pentru trimiterea mesajului pe retea.

Pragul de logare poate fi stabilit si la nivel de *appender*, folosind parametrul `Threshold`.

Un ***layout*** este **asociat** unui *appender* si determina **formatarea** mesajului respectiv. Exemple uzuale:

- `SimpleLayout`: sirul logat contine doar mesajul original
- `PatternLayout`: permite imbogatirea mesajului cu informatie suplimentara, precum numele fisierului, al metodei curente, al firului de executie etc. Formatarea se realizeaza prin indicatori asemanatori celor folositi de `printf` (vezi fisierul de configurare din sectiunea urmatoare).

Fisiere de configurare

Prezentam, in continuare, un fisier de configurare (`log4j.properties` - nume consacrat), aceasta fiind modalitatea recomandata, pentru a **evita modificarea** codului sursa cand se doreste schimbarea comportamentului:

log4j.properties

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger = DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1 = org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout = org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern = %-4r [%t] %-5p %c %x - %m%n
```

Comportamentul este acelasi ca in cazul `BasicConfigurator.configure()`, apel care acum trebuie inlocuit cu `PropertyConfigurator.configure(„log4j.properties")`. Pentru a intelege semnificatia specificatorilor de mai sus, cititi aici.

Diagnostic Contexts

In cazul in care aplicatia ruleaza pe **mai multe fire**, devine utila evidentierea mesajelor de logare referitoare la aceleasi prelucrari. Simpla tiparire a numelui firului de executie, in dreptul mesajului, este insuficienta in cazul in care se intrebuinteaza un mecanism de reciclare a firelor. Ar fi utila posibilitatea definirii unor **identificatori** ai fiecarui fir, care sa insoteasca fiecare mesaj de logare.

In sprijinul acestei probleme vine clasa `MDC` (*mapped diagnostic context*), ce ofera functionalitatea unui **dictionar** (*map*), cu precizarea ca asocierile cheie-valoare se mentin la nivel de **fir**. De remarcat ca metodele clasei sunt **statice**:

```
MDC.put("name", "Deirdre"); // per thread

// layout-urile pot fi comandate sa tipareasca aceasta informatie, prin indicatori ca
%X{name}

logger.info("Hello");

MDC.remove("name");
```

Un output posibil este:

```
Deirdre: Hello
```

Debugging

Depanarea reprezinta procesul de ***identificare*** si ***inlaturare*** a erorilor dintr-un program, referindu-se, in general, la **momentul rularii** (*runtime*). Pentru diminuarea comportamentului eronat al unui program se recomanda tratarea corespunzatoare a situatiilor **exceptionale** (semnalate prin **valori speciale** intoarse de functii sau prin mecanisme bazate pe **exceptii**, ca in cazul Java). O eroare netratata la timp se poate manifesta mult **mai tarziu** in fluxul de control, cand este dificila stabilirea legaturii cu momentul aparitiei.

In programele mari se pune problema **localizarii** erorilor, a determinarii zonei in care acestea se produc. Acest obiectiv se atinge **iterativ**, **restrangand**, treptat, zona de interes, prin simplificarea contextului in care se manifesta problema. De exemplu, daca la sfarsitul unei interactiuni complexe cu o interfata grafica, se obtine un comportament eronat, se va incerca reproducerea situatiei repetand doar anumiti pasi din cei anteriori. Sarcina este cu atat mai greu de indeplinit cu cat anumite bug-uri au o comportare ***nedeterminista***, putand aparea in momente diferite de la o rulare la alta a programului, sau chiar deloc. Cel mai adesea, in aceasta categorie se incadreaza problemele de **sincronizare** (*deadlocks*, *race conditions*).

Amintim cateva elemente din **terminologia** specifica, ce se regasesc si in depanatorul integrat al Eclipse, pe care il vom folosi in exercitii:

- ***breakpoint***: punctul in care executia programului va fi suspendata, pentru analiza detaliata. Pot fi:
  - *instruction breakpoint*: instructiune indicata de utilizator
  - *data breakpoint*: eveniment din executia programului, precum modificarea unei locatii de memorie
- ***stepping***: executia instructiunii curente, in scopul evaluarii individuale. Daca aceasta este un apel de functie, se poate face:
  - *step into*: se va sari la prima instructiune din corpul functiei
  - *step over*: se va considera apelul ca fiind atomic, si se va trece la urmatoarea
- ***watch***: mecanismul de urmarire a valorii unor variabile/expresii, pe masura ce acestea se modifica

- ***stack trace***: lantul de apeluri de functii care a condus fluxul de control in punctul curent

Pentru o prezentare mai detaliata, aveti la dispozitie un **tutorial** pentru Eclipse Debugger. Acesta beneficiaza de o facilitate denumita ***on-the-fly code fixing***, care se refera la posibilitatea de a actualiza, manual, valorile variabilelor in timpul procesului de debugging.

In Java, in momentul aparitiei unei **exceptii**, se afiseaza automat *stack trace*-ul. Acest fapt indica o posibila intrebuintare a acestui mecanism in scopuri de debugging. O instructiune de genul:

```
new Exception("I'm about to crash").printStackTrace();
```

va afisa la consola *stack trace*-ul curent, fara a interfera cu fluxul obisnuit de control. Acelasi efect se obtine prin apelul Thread.dumpStack().

Sa luam cazul unui program care **se blocheaza** (ar putea fi o bucla infinita sau un deadlock). Pornirea debugger-ului nu ar fi de mare folos, intrucat s-ar bloca si asa. Am avea nevoie de un hint despre zona care produce eroarea respectiva, pe baza careia sa putem fixa un *breakpoint*, la care executia programului sa fie suspendata. Ar trebui sa aflam ce executa de fapt programul in **momentul blocarii**. Pentru astfel de situatii ne vine in ajutor mecanismul de ***thread dumping***. In orice moment al rularii unui program Java, puteti intrebuinta combinatia `Ctrl+\` pe Linux, sau `Ctrl+Break` pe Windows pentru a solicita masinii virtuale Java sa realizeze un ***thread dump***, ce va contine informatii despre fiecare fir ce rula in acel moment:

- numele firului
- starea (rulabil, blocat la un lock etc)
- *stack trace*-ul

Iata un extras dintr-un *thread dump*:

```
"Finalizer" daemon prio=8 tid=0x01435000 nid=0x17b4 in Object.wait()
[0x035ef000..0x035efd00]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)(Test.java:13)
```

In versiunile mai noi, *thread dump*-ul poate contine informatii despre ***deadlock***-urile survenite.

O practica utila intr-o situatie precum cea de mai sus este obtinerea liniei la care firul se afla si setarea corespunzatoare a unui *breakpoint*.

Profiling

Profiling-ul reprezinta o modalitate de **studiere** a comportamentului programului pe masura ce acesta se executa. Scopul general este determinarea zonelor care necesita optimizari (CPU/memorie).

O regula confirmata adesea in practica este **80**/**20**: "80% din timpul unui program este petrecut in 20% din cod". Aceasta inseamna ca zonele critice (*bottlenecks*), ce ar trebui optimizate, trebuie alese cu grija. Altfel, riscati depunerea unui efort de optimizare a unor secvente executate foarte rar, obtinand o imbunatatire nesemnificativa a timpului de rulare.

Exista doua metode raspandite pentru realizarea profiling-ului:

- **instrumentare**: inserarea de instructiuni suplimentare de monitorizare a timpului si frecventei de executie a functiilor. **Dezavantajul** consta in posibila interpretare gresita a rezultatelor datorita overhead-ului acestor functii auxiliare, care acum se executa impreuna cu codul studiat

- **esantionare (sampling)**: verificarea periodica a starii programului, urmata de estimarea statistica a duratelor si frecventelor de rulare. Pentru aceasta metoda exista sprijin din partea kernel-ului si a hardware-ului: intreruperi, contoare de temporizare, contoare de cicli de ceas etc. Putem simula acest procedeu prin realizarea repetata de *thread dumps*.

Profiling-ul se poate face la diferite nivele:

- **CPU**: durata de executie si frecventa de apel pentru fiecare functie, ce permit sesizarea intrebuintarii ineficiente a procesorului
- **threads**: probleme de sincronizare:
  - deadlocks
  - lock contention (un thread tine o durata prea lunga un lock iar celelalte fire asteapta eliberarea)
- **memorie**:
  - numarul de instante alocate: permite determinarea unui abuz de memorie
  - referintele „vii“ la obiecte, ce impiedica eliberarea memoriei de catre garbage collector: permite detectarea *memory leaks*

Pentru profiling se poate folosi aplicatia TPTP, plugin pentru Eclipse (versiuni mai vechi, in Juno de exemplu nu mai este suportata).

Pentru instalare urmati pasii de mai jos:

- `Help → Install New Software`
- Introduceti http://download.eclipse.org/releases/helios `(sau altul mai recent)`
- Selectati `Test and Performance`
- Finalizati procesul de instalare.

Static checking

In afara de cele doua metode descrise mai sus, ce realizeaza analiza **dinamica**, la executie, exista si metode de verificare **statica**, pe baza codului sursa sau a celui binar.

Doua astfel de utilitare, ce apartin proiectului JLint, sunt:

- **JLint**: analizeaza ***bytecode***-ul Java, continut in fisierele `.class`, si semnaleaza diferite probleme, printre care:
  - posibilitatea aparitiei unui *deadlock*, din cauza obtinerii de *lock*-uri in alta ordine in fire diferite
  - definirea intr-o clasa copil a unei metode cu acelasi nume cu o metoda din clasa mama, dar cu parametri diferiti. In acest caz utilitarul intuieste intentia programatorului de a supradefini, de fapt, metoda din clasa de baza, si atrage atentia asupra acestui fapt
- **AntiC**: analizeaza **cod sursa** `.java`, adresand, mai degraba, probleme de nivel sintactic, ca, de exemplu:
  - = in loc de ==
  - prioritatea operatorilor

Pentru Linux, utilitarele sunt disponibile in forma codului sursa, ce trebuie compilat executand `make`.

Mersul lucrarii:

- Utilizati scheletul de laborator (fisierul TIDPP_Lab_4a_skel.rar pentru *logging* si TIDPP_Lab_4b_skel.rar pentru *debugging & profiling*)
- Importati proiectul in Eclipse (File→Import→General→Existing projects in workspace).
- Punctele marcate cu (T) au asociat un comentariu TODO in cod. Exemplu: pentru exercitiul 1 gasiti comentariul // TODO 1
- Rulati aplicatia pentru a observa dispunerea componentelor. Aruncati o privire asupra claselor din proiect.

Enunturi (logging)

1. (**1p**) Adaugati instructiuni de *logging*, cu **diferite** prioritati, in mai multe puncte ale programului (inclusiv in clasa `ListWorker`), pentru a putea urmari un lant **complet** de apeluri, de la interfata grafica pana la executarea actiunii (GUI - mediator - *state manager* etc.).
    a. Botezati *logger*-ii cu numele claselor in care sunt definiti.
    b. Afisati mai intai la consola (`BasicConfigurator`).
2. (**1p**) Creati un fisier `log4j.properties`, la **acelasi nivel** cu fiserele `.class`, cu continutul din textul laboratorului. Plasand fisierul in aceasta locatie, va fi citit automat dupa **inlaturarea apelului** `BasicConfigurator.configure()`. Ce reprezinta `rootLogger` din fisier? Schimbati *log level*-ul (prioritatea) si rulati aplicatia.
3. (**1p**) Modificati fisierul de configurare, prin adaugarea un ***file appender*** (manual log4j). Rulati.
    a. Adaugati un ***pattern layout*** pentru noul *appender*, asemanator cu *layout*-ul pentru `ConsoleAppender`, din fisier, si tipariti numele fisierului, numele metodei si linia din fisier (cititi despre `PatternLayout`). Studiati fisierul de log.
4. (**1p**) Adaugati un ***rolling file appender***, ce permite generarea unor fisiere de log de o **dimensiune maxima**, urmand ca, la depasirea acesteia, sa se comute la alt fisier de log. De asemenea, permite precizarea **numarului maxim** de astfel de fisiere, ce se doresc pastrate.
    a. Stabiliti, doar pentru acest *appender*, un prag de logare mai mare. **Hint**: parametrul `Threshold`.
5. (**1p**) Folositi un *mapped diagnostic context* pentru **gruparea** mesajelor de logare, provenite de la diferiti `SwingWorker`-i.

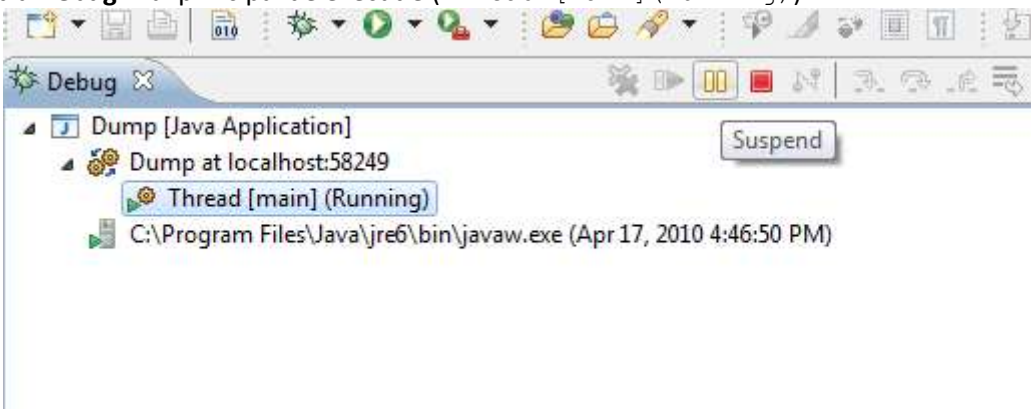
Enunturi(debugging & profiling)

Utilizati *TIDPP_Lab_4b_skel.rar* pentru partea de debugging.
Majoritatea exercitiilor au un comentariu TODO asociat in cod, de exemplu: pentru exercitiul 2, subpunctul 4, cautati `// TODO 2.4`. Exercitiile trebuie rezolvate **in ordine**.

1(2p). Debugging & Thread dumping

1. Cercetati clasa `Dump`. Rulati.
2. Porniti procesul de debugging (`F11`). Observati ca nu este de ajutor, programul ramanand blocat.
3. Obtineti un thread dump al programului, in timp ce acesta ruleaza:
    - rulati programul dintr-o consola (directorul bin)
    - in timpul rularii, tastati `Ctrl+\` pe Linux, sau `Ctrl+Break` pe Windows
    - la consola vor aparea informatii despre stiva de apeluri (stack trace) a fiecarui fir de executie in parte.
    - rulati din nou programul, de data aceasta redirectand, din consola, iesirea intr-un fisier. Repetati secventa de mai sus pentru stocarea informatiilor in fisier si studiati-l.
    - identificati firul `main`, si linia de program la care firul se afla in momentul obtinerii dump-ului
    - observati, de asemenea, starile in care se aflau firele (majoritatea sunt fire interne JVM). Ca o precizare, starile sunt cele percepute de masina virtuala, care pot diferi de cele din sistemul de operare.

4. (Alternativa la punctul anterior) Din Eclipse, puteti obtine un *stack trace* astfel:
   - executati programul `Dump` in modul depanare (`F11`)
   - din meniul Window, deschideti perspectiva **Debug**
   - selectati in fereastra **Debug** firul principal de executie (`Thread [main](Running)`)
   - **suspendati** executia curenta (apasati pe suspend, butonul prezentat in figura alaturata)
   - veti obtine un **dump** al thread-ului curent.



5. Folositi-va de linia determinata in fisier (sau de *dump*-ul obtinut in Eclipse) pentru a stabili un **breakpoint**:
   - executati dublu-click pe bara laterala, in dreptul liniei cu pricina si porniti debugging-ul
   - rulati instructiune cu instructiune, folosind `Step Into (F5)`.
   - observati cum evolueaza valorile variabilelor locale si rezolvati problema.
6. Pe masura ce parcurgeti instructiunile, incercati sa:
   - editati valorile variabilelor direct in tabelul de variabile (inclusiv membrul `value` al lui `str`)
   - porniti **urmarirea** expresiei `str.charAt(i)` (selectare si alegere Watch din meniul contextual)
   - rezolvati problema astfel incat programul sa ajunga la final fara blocare
7. Analizati clasele `WordAnalyzer` si `WordAnalyzerTester`. Exemplul construieste un analizor de cuvinte, clasa `WordAnalyzer`, si un test pentru aceasta clasa.
   - Fara sa rulati programul `WordAnalyzer`, incercati sa preziceti *output*-ul acestuia. Presupuneti ca metoda `firstRepeatedCharacter` functioneaza corect.
   - Rulati programul. Analizati rezultatul. Din *stack trace*-ul rezultat incercati sa identificati problema aparuta.
   - Analizati linia 24, `word.charAt(i+1)`. Ce erori pot fi aruncate de aceasta instructiune? Incercati sa reparati problema aparuta in executia anterioara.
   - Modificati `WordAnalyzerTester` si adaugati un apel de tipul `test(null)`. Unde este eroarea aruncata?
   - Ar fi mai indicat sa avem o **exceptie** aruncata chiar din constructor, astfel am fi putut identifica mai repede aparitia problemei cauzate de obiectul `null`. Cel mai usor mecanism pentru captarea acestei probleme consta in folosirea de **asertiuni**. O asertiune este o conditie ce trebuie sa fie adevarata pentru ca un program sa functioneze corect. Daca acea conditie nu este indeplinita, atunci programul esueaza cu un mesaj de eroare corespunzator. Un astfel de aspect conduce la o identificare mai usoara, la runtime, a unor probleme cu executia codului, fara sa fie necesar sa depistam ca rezultatele finale sunt gresite. Putem mai usor identifica locul aparitiei unui anumit defect.
   - Adaugati ca prima instructiune in constructorul clasei `WordAnalyzer` instructiunea **`assert`**, precum in exemplul:

```
public WordAnalyzer(String aWord) {
    assert aWord != null;
    word = aWord;
}
```

   - Rulati din nou programul. Ce observati?
   - Asertiunile sunt dezactivate implicit, din ratiuni de performanta. Pentru activarea acestora, trebuie sa rulam programul cu optiunea `-ea` (sau `-enableassertions`):

```
> java -ea WordAnalyzerTester
```

   - Rulati programul cu aceasta optiune. Care este *output*-ul programului in acest caz?

2(0.5p). Deadlock debugging

1. Studiati clasa `Deadlock`.
2. Utilizati una din solutiile de mai sus pentru a obtine un **thread dump** si studiati-l.
3. Rezolvati problema.
4. Descarcati si despachetati JLint de aici. (Versiunea 3.0 contine executabilul pentru Windows)
5. In forma initiala a programului, rulati `JLint` din consola:

   ```
   jlint Deadlock.class
   ```

   Ce observati?

3(2p). Debugging

1. Cercetati clasa `Fibonacci`. Rulati programul cu argument 2,3,… Observati eroarea
2. Realizind debugging in cod detectati sursa problemei si reparati codul.
3. Verificati F(6)=8

4(0.5p). Antic

1. Rulati clasa Static. Ce observati?
2. Modificati x a.i long x = 0x1l; Salvati fisierul.
3. Rulati programul `antic`, obtinut in urma compilarii de la punctul 2, pe aceasta clasa:

   ```
   antic Static.java
   ```

4. Faceti schimbarile necesare si rulati inca o data `programul antic`.

Resurse utile

- Pagina oficiala log4j
- Download log4j
- Manual log4j
- Tutorial Eclipse Debugger
- JProfiler si un manual JProfiler
- JLint