Laborator 2

Clase Imbricate

Limbajul de programare Java permite definirea unei clase in interiorul corpului unei alte clase. O asttel de clasa este denumita *clasa imbricata* si exista doua feluri de astfel de clase:

- clase imbricate statice
- clase imbricate non-statice, denumite *clase interne*

```
public class SomeOuterClass {
      ...
      static class SomeStaticNestedClass { ... }
      ...
      class SomeInnerClass { ...}
}
```

Clasele imbricate sunt considerate ca entitati membre ale clasei unde au fost definite. Clasele interne pot accesa restul membrilor clasei externe, in corpul careia a fost definita, chiar daca acesti membri prezinta modificatorul de acces *private*. Clasele imbricate statice nu au acces la alti membri ai clasei externe. O clasa imbricata poate fi declarata cu orice modificator de acces.

Clasele imbricate permit:

- inglobarea intr-un singur context logic a claselor care sunt utilizate intr-un singur loc
- cresterea incapsularii
- cresterea inteligibilitatii si a mentenabilitatii codului.

Accesul unei clase imbricate statice se face prin utilizarea operatorului "." la nivelul clasei externe: `SomeOuterClass.SomeStaticNestedClass.` De exemplu:

```
SomeOuterClass.SomeStaticNestedClass someObject =
                                  new SomeOuterClass.SomeStaticNestedClass();
```

Clase imbricate interne:

- sunt asociate la nivel de instante ale clasei externe, in corpul carora au fost definite.
- au acces la campurile si metodele obiectului instanta al clasei externe
- nu poate defini membri statici
- pentru a instantia un obiect al clasei interne trebuie mai intai sa existe o instanta a clasei externe:

```
SomeOuterClass someOuterObject = new SomeOuterClass();
SomeOuterClass.SomeInnerClass someInnerObject =
      someOuterObject.new SomeInnerClass();
```

In cadrul proiectului XAnd0 se poate inlocui variabila membra counter declarata in corpul clasei Xand0Environment cu o clasa interna specializata:

```
class Counter {
      private int counter;

      private int resetValue;
      ...}
```

Exercitiul 1: Creati clasa Counter astfel:

a) adaugati un constructor fara argumente care initializeaza cele doua variabile membre, de tip intreg, la valoarea 0;
b) adaugati un constructor cu un argument care initializeaza cele doua variabile membre la valoarea argumentului;
c) adaugati doua metode membre increment() si decrement() care realizeaza incrementarea, respectiv decrementarea cu o unitate a variabilei counter;
d) adaugati o metoda reset care egaleaza valoarea mebrului counter cu valoarea membrului resetValue;
e) adaugati o metoda membra isEqual(int) care primeste ca argument o valoare intreaga; daca variabila membra counter este egala cu variabila argument primita atunci metoda returneaza valoarea truealtfel returneaza false;
f) adaugati o metoda isDraw() pentru verificarea conditiei de remiza; folosind metoda isEqual verificati daca valoarea membrului counter este egala cu board.getCells().size(); metoda returneaza o valoare booleana.

Exercitiul 2: Modificati clasa Xand0Environment prin adaugarea clasei interne de mai sus.

Exercitiul 3: Modificati clasa Xand0Environment prin instantierea unui obiect Counter in constructorul clasei externe iar apoi utilizarea acestui obiect la nivelul metodei doNextMove(..) ca alternativa la variabila membra counter utilizata precedent.

Adnotari

Adnotarile reprezinta un mecanism prin care informatii suplimentare sunt furnizate la nivelul programului. Adnotarile nu prezinta un efect direct asupra operatiilor codului pe care il adnoteaza si sunt utilizate de regula pentru:

- a oferi informatii compilatorului – adnotarile pot fi utilizate de compilator in detectia erorilor sau pentru suprimarea mesajelor de avertizare
- pentru procesarea de la runtime
- procesare la compilare si instalare – anumite instrumente software pot apela la informatia suprinsa de adnotari pentru a genera cod, fisiere in anumite formate, completarea de informatie a anumitor campuri din baze de date, etc.

Adnotarile pot fi aplicate la nivelul:

- claselor
- campurilor
- metodelor
- alte elemente din program

Adnotarile au fost intalnite inca din exemplele din laboratorul 1. Astfel in clasa Xand0BoardCell, care implementeaza BoardCellInterface<Double>, s-au utilizat adnotarile @Override la suprascrierea metodelor din interfata BoardCellInterface. Adnotarea @Override este utilizata de compilator pentru a verifica ca in clasa de baza sau in interfata implementata exista metodele ce urmeaza sa fie suprascrise. In caz contrar se primeste o eroare de compilare. Alte adnotari de compilare sunt:

- @Deprecated care marcheaza o anumita metoda/clasa ca fiind depreciata si neindicata, de autorii API-ului, pentru utilizare
- @SuppressWarnings este o adnotare prin care se poate indica compilatorului sa ignore anumite tipuri de avertismente:

```
// use a deprecated method and tell compiler not to generate a warning
@SuppressWarnings("deprecation")
  void useDeprecatedMethod() {
        objectOne.deprecatedMethod(); //deprecation warning - suppressed
  }
```

Tipuri Generice

Tipurile generice au fost adaugata incepand cu versiunea 1.5 a SDK-ului Java urmanrind in primul rand stabilitatea codului si detectarea a cat mai multor erori de programare la compilare astfel incat acestea sa nuajunga in faza de runtime. Tipurile generice sunt extensiv folosite la nivelul API-ului pentru structuri de date ce reprezinta colectii (.v sectiunea Colectii).

In cadrul proiectului pentru implementarea jocului “X si 0” interfata BoardCellInterface a fost implementata astfel incat sa foloseasca tipuri generice.

Clasa s-a declarat folosind o *declaratie de tip generic*:

```
public interface BoardCellInterface<E>...
```

Declaratia interfetei BoardCellInterface introduce o *variabila de tip* E ce poate fi utilizata – cu anumite restrictii – oridunde in corpul interfetei.Acelasi mecanism de declarare si utilizare de variabile de tip este identitc atat pentru clase cat si pentru interfete. Variabila de tip E reprezinta un substituent pentru un tip concret. In acest moment nu se stie ce tip de date concret necesita interfata BoardCellInterface si ca urmare se foloseste mecanismul bazat pe tipuri generice. In contextul in care se va cunoaste ce tip de obiecte poate fifolosit in interfata BoardCellInterface se va inlocui variabila de tip cu tipul de obiecte necesar – variabila E nu poate fi inlocuita cu tipuri primitive. In cazul interfetei BoardCellInterface, variabila E este utilizata pentru a defini cele doua metode al;e ionterfetei:

```
public interface BoardCellInterface<E> {
/*
 * A cell on a board game either consumes/receives information from the
 * player (e.g. adds and X or an 0
 * in the X and 0 game) or it produces/specifies an action to the player
 * (e.g. Monopoly like games)
 */
      public void consume(E info);
      public E produce();
```

Din perspectiva jocurilor de tip board-game se poate interpreta ca suprafata de joc este formata din celule care fie adauga informatie pe suprafata de joc – consumatori (e.g. jocul X si 0, spanzuratoare, tetris) fie au o structura read-only prin care ofera informatii si instructiuni suplimentare participantilor la joc (e.g. jocul Monopoly). O clasa care implemeteaza interfata BoardCellInterface – o celul a suprafetei de joc – trebuie sa implementeze amebel metode si pentru aceasta fie foloseste la randul ei o variabila de tip fie specifica un tip de obiecte pentru variabila de tip de la nivelul interfetei:

```
public class Xand0BoardCell<T> implements BoardCellInterface<T> {...}
public class Xand0BoardCell implements BoardCellInterface<Double> {...}
```

Un tip generic poate folosi la nivelul declaratiei multipli parametri in cadrul operatorului <> pentru a folosi mai mult de o variabila de tip. Fiecare parametru insa trebuie sa fie unic (e.g. <E,E> produce o erorare in tip ce <E,F> este acceptat).

In cadrul proiectului XAnd0 s-a utilizat o clasa Pawn pentru definirea obiectelor de tip Xand0Player.

Aceasta clasa avea urmatoarea declaratie:

```
public class Pawn {
	// fields
	protected String name;
	protected Integer color;
	protected Double value;

	/**
	 * @param name
	 * @param color
	 */
	public Pawn(String name, Integer color, Double value) {
		this.name = name;
		this.color = color;
		this.value = value;
	}

	//getters and setters
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Integer getColor() {
		return color;
	}
	public void setColor(Integer color) {
		this.color = color;
	}

	public Double getValue() {
		return value;
	}

	public void setValue(Double value) {
		this.value = value;
	}

	@Override
	public boolean equals(Object o) {
		Pawn other = (Pawn)o;
		if(name.equals(other.getName()) && color.equals(other.color) &&
value.equals(other.value)) {

			return true;
		}
		return false;
	}
}
```

Deoarece clasa Pawn a fost dezvoltata pentru a suprinde conceptul de pion si afost creata specific pentru jocul “X si 0” ea nu reprezinta tip usor reutilizabil pentru alte jocuri. Cu toate acestea notiunea de pion estedestul de generala si este de regula parte a multor jocuri. Lipsa de reutilizare a acestei clase la nivelul altor jocuri provine din reprezentarea informatiei ce o poarta pionul ca un simplu tip de date Double. Pentru a asigura un tip mai puternic ca si nivel deabstractizare se poate

rescrie clasa Pawn ca un tip generic astfel:

```
	public class Pawn<T> {
		// fields
		protected String name;
		protected Integer color;
		protected T value;

		/**
		 * @param name
		 * @param color
		 */
		public Pawn(String name, Integer color, T value) {
			this.name = name;
			this.color = color;
			this.value = value;
		}

		//getters and setters
		public String getName() {
			return name;
		}
		public void setName(String name) {
			this.name = name;
		}
		public Integer getColor() {
			return color;
		}
		public void setColor(Integer color) {
			this.color = color;
		}


		public T getValue() {
			return value;
		}

		public void setValue(T value) {
			this.value = value;
		}

		@Override
		public boolean equals(Object o) {
			Pawn other = (Pawn)o;
			if(name.equals(other.getName()) && color.equals(other.color) &&
	value.equals(other.value)) {
				return true;
			}
			return false;
		}
	}
```

Aceasta redeclarare determina erori la nivelul claselor Xand0Logic si Xand0Environment deoarece acestea incearca sa obtina pebazaunui obiect Pawn o valoare de tip double.

Pentru a evita aceste erori, la nivelul clasei Xand0Player, se fac modificarile necesare astfel incat sa lucreze cu o variabila membra de tip Pawn<Double> si nu Pawn

```
	public class Xand0Player extends Player {
```

```
            // each player has a token or pawn in any board game; hear a pawn
represents the X or the 0 stamp
            protected Pawn<Double> pawn;
```

… modificati clasa Xand0Player astfel incat sa se elimine erorile de compilare si sa lucreze corect cu noul tip de variabila membra;

In orice clasa unde se doreste o referinta catre tipul generic Pawn<E> se realizeaza o *invocare de tip generic*:

```
Pawn<Double> pawn;
```

Acest lucru determina inlocuirea variabilei de tip cu un tip de obiect concret. Invocarea unui tip generic se cheama *tip parametrizat.*

Instantierea un tip parametrizat este similara cu instantierea oricarei clase, prin utilizarea operatorului new:

```
Pawn<Double> pawn = new Pawn<Double>(...);
```

In acest momemnt se pot invoca metodele obiectului, instantiat cu tipul concret specificat:

```
Double value = pawn.getValue();
```

Variabilele de tip nu sunt tipuri propriuzise (e.g. E,T utilizate mai sus nu sunt tipuri obscure de date) ci reprezinta doar modalitati specifice limbajului pentru a preciza informatie suplimentara la compilare. La compilare intreaga informatie despre tipuri generice dispare existand doar fisierul Pawn.class.

Tipurile ghenerice prezinta multe alte caracteristici si particularitati care sunt acoperite in detaliu la aceasta adresa.

Numere

In cadrul proiectului XAnd0 pentru tipurile de date reprezentand numere s-au utilizat clasele aferente si nu tipurile de baza:

```
Double value = pawn.getValue();
```

spre deosebire de: `double value = pawn.getValue();`

Acest lucru permite conversii mai precise intre numere precum si conversii catre siruri de caratere in format readble-friendly.

Toate aceste tipurti de nume sunt derivate din clasa java.lang.Number al carei API poate fi vizualizat aici.

Principalele motive pentru utilizarea unui tip Number in loc de un tip primitiv:

- fiind o clasa Number extinde clasa java.lang.Object si poate fi utilizata la nivelul metodelor care primesc ca parametru un obiect si nu un tip primitiv;
- aceste clase definesc o multime de constante (MIN_VALUE, MAX_EXPONENT) care specifica limitele inferioara si superioara pentru respectivul tip de date;
- fiecare clasa prezinta o serie de metode pentru conversia datelor de la si catre tipuri primitive, stringuri de caractere si diverse formate de reprezentare a numerelor.

```
long n = 461012;
System.out.format("%d%n", n);                  //  -->  "461012"
System.out.format("%08d%n", n);                //  -->  "00461012"
System.out.format("%+8d%n", n);                //  -->  " +461012"
System.out.format("%,8d%n", n);                //  -->  " 461,012"
System.out.format("%+,8d%n%n", n);             //  -->  "+461,012"

double pi = Math.PI;
System.out.format("%f%n", pi);                 //  -->  "3.141593"
System.out.format("%.3f%n", pi);               //  -->  "3.142"
System.out.format("%10.3f%n", pi);             //  -->  "     3.142"
System.out.format("%-10.3f%n", pi);            //  -->  "3.142"
System.out.format(Locale.FRANCE,
            "%-10.4f%n%n", pi);          //  -->  "3,1416"
```

De asemenea un set de functii utile pentru manipulari matematice avansate sunt cuprinse in clasa java.lang.Math. Metodele acestei clase sunt definite ca fiin statice si prin urmare pot fi apelate prin simpla utilizare a operatorului “.” precedat de numele clasei:

```
Math.cos(3.4);
```

Detaliile acestor clase se agsesc in API-ul java.

Mai multe detalii despre manipularea numerelor Java la aceasta adresa.

String-uri

Dupa cum s-a observat din proiectul XAnd0 sirurile de caractere sunt des utilizate in Java pentru a descrie stari, pentru a serializa anumite informatii in diverse formate sau pentru a oferi informatii utilizatorilor sau pentru a prelua informatii de la acestia. In laboratoarele urmatoare odata cu crearea iterfetelor grafice importanta stringurilor va deveni si mai evidenta. Un argument convingator in aceasta directie consta in existenta unei metode toString() la nivelul clasei din varful ierarhiei Java: Object. Metoda toString() retruneaza un sir de caractere care descriu obiectul pentru care se apeleaza aceasta metoda.

Drept urmare API-ul java ofera un suport extensiv pentru crearea si manipularea stringurilor. Majoritatea acestui suport este inclus in clasele: java.lang.String, java.lang.StringBuilder, java.util.StringTokenizer si java.util.Scanner. Detaliile acestor clase se regasesc in documentatia API-ului java (v. link-ul de mai sus).

Crearea unui string: `String string = "some string";`

```
// OR
char[] arrayChars = {'a','r','r','a','y',' ','o','f',' ','c','h','a','r','s'};
String otherString = new String(arrayChars);
```

Acest sir de caractere nu poate fi modificat (i.e. nu se pot adauga sau extrage caractere fara a crea in realitate un alt obiect String). Obiectele de tip StringBuilder sunt siruri de caractere ce pot fi modificate.

Metodele oferite de API-ul java permit:

- extragerea de informatii despre stringuri (e.g. lungime)
- concatenare de doua sau mai multe stringuri
- descrierea stringurilor in anumite formate
- manipularea/extragerea/inlocuirea de caratere si substringuri
- compararea string-urilor sau regasirea unui substring

Exercitiul 4: Implementati metoda Xand0Player.computeExperience() astfel incat sa se calculeze urmatoare formula folosind metodele clasei java.lang.Math si numai tipuri de date ne primitive.

$$\sqrt{experience^{\log(age)}}$$

Dupa ce se efecuteaza calculul, inainte de a returna valoarea calculata, se afiseaza rezultatul in urmatorul format: `+###,###.#####.`

Exercitiul 5: Modidificati metoda computeExperience() astfel incat sa primeasca un argument de tip String. Metoda realizeaza urmatorii pasi:

a) verifica daca string-ul este in format numeric; daca nu este in format numeric se arunca o exceptie (utilizand cuvantul cheie throws); daca mecanismul de tratare a exceptiilor nu a fost aprofundat atunci sa se afiseze un mesaj de avertizare si sa se returneze imediat cu valoarea null;

b) se foloseste valoarea numerica continuta in obiectul String pentru a calcula experienta user-ului pe baza formulei: $\sqrt{experience^{\log(\frac{argument}{age})}}$

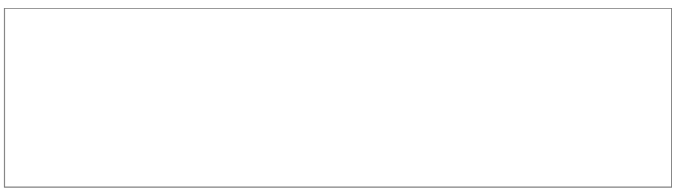
Colectii de date

In aceasta sectiune se prezinta pe scurt framework-ul de structuri pentru colectii de date. Acest framework prezinta un API important pentru platforma Java precum si pentru aplicatiile dezvoltate folosind aceasta plaftorma.

Colectiile sunt obiecte utilizate pentru a grupa elemente multiple si furnizeaza mecanismele necesare pentru stocarea, extragerea, manipularea si comunicarea datelor agregat. Framework-ul pentru colectii Java reprezinta o arhitectura unificata pentru reprezentarea si manipularea colectiilor si contine:

- interfete pentru a putea manipula interfetele intr-o maniera independeta de reprezentarea/implementarea acestora
- implementari specifice pentru a include mecanismele cele mai eficiente pentru reprezentarea si manipularea colectiilor de date
- algrotimi pentru diverse calcule des intalnite la nivelul colectiilor de date (e.g. cautare, sortare)

Interfetele esentiale care formeaza nucleul framework-ului pentru colectii sunt: Collection si Map, interfetele derivate din Collection – Set, List, Queue, interfata derivata din Map – SortedMap.



Observatie: Toate interfetele de mai sus sunt definite ca tipuri generice.
Interfata pariente Collection este declarata:

```
public interface Collections<E> { ... }
```

unde variabila de tip E specifica ce tip vor avea obiectele colectiei.

Apeland la documentatia Java API revizuiti metodele declarate in interfata Collections. Interfata asigura metode pentru aflarea dimensiunii colectiei (size, isEmpty), pentru verificarea existentei unui obiect in colectie (contains), pentru actualizarea colectiei prin extragere sau adaugare deelemente (add, remove) si pentru a asigura un iterator peste aceasta colectie (iterator).

Un Set reprezinta o colectie care nu poate contine elemente duplicat. Verificarea echivalentei obiectelor se face pe baza metodei equals.

Tipul List reprezinta o colectie ordonata de elemente care pot fi duplicate. Prezinta controlul exact asupra pozitiilor de inserare a elementelor precum si asupra accesarii acestora care se face pe baza de index intreg.

Tipul Queue reprezinta o colectie utilizata pentru stocarea unei multimi de obiecte precedent procesarii acestora. Pe langa operatiile specifice unei colectii o coada permite operatii aditionale de inserare, extragere si inspectare a elementelor colectiei.

Un obiect de tip Map reprezinta o colectie de perechi chei-valori. Accesul la valori se face pe baza cheilor motiv pentru care acestea nu pot fi duplicate in cadrul aceleasi colectii Map.

Traversarea colectiilor se poate face in doua feluri:

a) utilizand constructia de ciclare for-each

```
for(Object o: collection){
      if(o instanceof Integer) {
            System.out.println("Integer collection");
      }
}
```

b) utilizand Iteratori

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

In cadrul proiectului XAnd0 s-au folosit doua tipuri de colectii: ArrayList si HashMap:

```
public class Xand0Environment {

      ...

      protected ArrayList<Xand0Player> players; ...}


public class Xand0Board /*extends Board<String,Point>*/{

      protected HashMap<Point,Xand0BoardCell> cells; …}
```

Exercitiul 6: Utilizati in loc de ArrayList un LinkedList, un Vector iar in final cu un HashSet pentru a stoca colectia de jucatori. Eliminati erorile aparute. Dupa utilizarea tuturor tipurilor reveniti la colectia care vi s-a parut cea mai utila.
Exercitiul 7: Cititi descrierea LinkedHashMap din API-ul Java iar apoi utilizati aceast tip pentru stocarea colectiei de celule la nivelul tablei de joc.
Exercitiul 8: Completati clasa Xand0Environment cu doua metode similara doNextMove si anume: undoLastMove() si undoLastNMoves(int N). Pentru stocarea istoriei de mutari utilizati o colectie adecvata.

Sistemul standard I/O

Conceptul de baza ce sta la baza API-ului I/O in Java este stream-ul de date. Un stream I/O reprezinta o sursa de intrare si o destinatie de iesire. Astfel un stream poate reprezenta diferite tipuri de surse si destinatii fisiere pe disc, divece-uri, alte programe si aplicatii, matrici de memorie, etc). Stream-urile suporta o varietate de tipuri de date ce sunt trasmise intre sursa si destinatie: bytes, tipuri primitive, caractere, obiecte. Astfel modelul unui stream poate fi reprezentat printr-o secventa de date iar un program utilizeaza un *stream de intrare* pentru a obtine secvential date de la entitatea sursa si utilizeaza un *stream de iesire* pentru a scrie secvential date catre entitate sursa.

Ca orice limbaj de programare si limbajul Java suporta interactiunea cu device-urile standard de I/O – tastatura si monitorul. Java asigura aceasta interactiune prin intermediul a trei stream-uri de intrare iesire: System.in – intrarea standard (tastatura), System.out – iesirea standard (monitor) si System.err iesirea standard pentru stream-ul care suprinde erorile. API-ul pentru I/O Java este mult mai vast de atat si prezinta caracteristici care vor fi explicate pe parcurusl acestul laborator acolo unde vor aparea (e.g. interactiunea cu fisiere pe disc).

La nivelul proiectului XAnd0 a fost furnizata clasa de test xand0.test.testGameLogic pentru verificarea functionalitatii corecte a logicii jocului “X si 0”. In locul stream-ului de intrare System.in s-a utilizat un mecanism special pentru citirea datelor de la tastaura. Acest mecanism implica utulizarea clasei Scanner si este utilizat de regula pentru a asigura o translatare facila a stream-ului de intrare in formate corespunzatoare utilizatorilor.

```
Scanner scanner = new Scanner(new BufferedInputStream(System.in), "UTF-8");

int x = scanner.nextInt();
```

Dupa cum se observa obiectul scanner este intializat cu un stream de intrare care este chiar System.in.
Exercitiu 9: Utilizati System.out si Scanner pentru a intializa numele jucatorilor prin solicitarea informatiilor acestora de la tastaura: nume, culoarea pionului si tipul acestuia (X sau 0). (realizati modificarile in clasa xand0.test.TestGameLogic).

Thread-uri Java

Definirea si pornirea thread-urilor

Platforma Java este dezvoltata astfel incat sa suporte programarea concurenta. Prin concurenta se intelege abilitatea de a executas mai multe aplicatii/programe in acelasi timp. Unitatea de baza de executie la nivelul unui procesor este procesul. La un moment dat nu se poate afla in executie decat un singur proces. Primul pas catre mimarea efectului de concurenta a fost introducerea sistemelor de operare cu multiprocesare la nivelul carora mai multe procese puteau sa fie pornite iar acestea comutau intre ele astfel ca fiecare proces executa o perioada limitata de timp pe procesor functie de prioritatile

asignate. Deaorece comutarea intre procese este un mecanism greoi si de durata au fostconcepute thread-urile care reprezinta unitati de procesare simplificate. Thread-urile sunt unitati de executie in cadrul aceluiasi proces. Astfelele pot partaja o serie de informatii iar comutarea intre threaduri devine un mecanism mai simplu cu durata redusa.

Platforma Java a prezentat mereu suport pentru thread-uri iar incepand cu versiuna 1.5 a adaugat un API pentru high-level concurency.

Pentru crearea unui thread o aplicatie trebuie sa ontina o instanta a clasei Thread. Acest lcuru poate fi realizat in 2 feluri:

- instantierea unei clase care implementeaza interfata Runnable
  - interfata Runnable defineste o singura metoda run() ce trebuie definita si la nivelul careia se specifica functionalitatea/logica de executie a thread-ului:

```
public class HelloThread implements Runnable {

	String name;
	public HelloThread(String name) {
		this.name = name;
	}
	@Override
	public void run() {
		System.out.println("Hello " + name);
	}

	public static void main(String[] args) {
		Thread helloThread = new Thread(new HelloThread("Alex"));
		helloThread.start();
	}
}
```

- instantierea unei clase care extinde clasa Thread;
  - clasa Thread implementeaza interfata Runnable astfel ca pentru a determina o anumita functionalitate pentru noul Thread trebuie redefinita metoda run:

```
public class HelloThread extends Thread {

	String name;
	public HelloThread(String name) {
		super();
		this.name = name;
	}
	@Override
	public void run() {
		System.out.println("Hello " + name);
	}

	public static void main(String[] args) {
		HelloThread helloThread = new HelloThread("Alex");
		helloThread.start();
	}
}
```

Pentru a porni un thread se invoca metoda start() a acestuia.

Alegerea uneia dintre aceste metode depinde de dezvoltator si de analiza pe care acesta o realizeaza din perspectiva aplicatiei. In cazul in care clasa care implementeaza un thread trebuie sa

extinda o alta clasa atunci desigur se va utiliza varianta bazata peimplementarea interfetei Runnable.

In cazu in care aplicatiile sunt suficient de simple si nu se doreste o comportare prea complicata la nivelul Thread-ului al doilea mecanism poate fi utilizat cu succes. Dupa cum se observa instantierea in al doilea caz este simplificata iar starea thread-ului (i.e. variabilele membre) sunt incluse chair la nivelul obiectului thread. In primul caz starea thread-ului este continuta intr-un obiect aflat intr-o relatie de asociere 1:1 cu instanta trhead-ului.

De regula primul caz asigura cel mai bun nivel de flexibilitate si generalism astfel ca va fi folosit in exemplele urmatoare.

Un thread poate fi comutat in starea de asteptare prin metoda Thread.sleep. Invocarea acestei metode determina suspendarea thread-ului curent pentru o anumita perioada de timp:

```
public class HelloThread implements Runnable {

	String name;
	long sleepTime;
	public HelloThread(String name, long sleepTime) {
		this.name = name;
		this.sleepTime = sleepTime;
	}
	@Override
	public void run() {
		try {
			Thread.sleep(sleepTime);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("Hello " + name);
	}

	public static void main(String[] args) {
		Thread helloT1 = new Thread(new HelloThread("Alex",0));
		Thread helloT2 = new Thread(new HelloThread("Mihnea",2000));
		helloT2.start();
		helloT1.start();
	}
}
```

Metoda sleep poate determina aparitia unor exceptii iar aceste trebuie tratate corespunzator. Deoarece metoda run este definita cu un anumit header nu se poate adauga clauza throws in signatura functiei motiv pentru care Thread.sleep este inclusa intr-un bloc try-catch.

Pentru a determina un thread sa isi suspende executia pana cand un alt thread termina de executat metodei run (finalizarea metodei run induce finalizarea thread-ului) se foloseste metoda Thread.join:

```
public class HelloThread implements Runnable {

	String name;
	long sleepTime;
	public HelloThread(String name, long sleepTime) {
		this.name = name;
		this.sleepTime = sleepTime;
	}
	@Override
	public void run() {
```

```
            try {
                  Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("Hello " + name);
      }

      public static void main(String[] args) {
            Thread helloT1 = new Thread(new HelloThread("Alex",0));
            Thread helloT2 = new Thread(new HelloThread("Mihnea",2000));
            helloT2.start();
            helloT1.start();

            helloT1.join();
            helloT2.join();
      }
}
```

In exemplul de mai sus thread-ul principal (orice aplicatie detine un thread principal implicit care executa metoda main) asteapta pana cand cele doua thread-uri helloT1 si helloT2 finalizeaza.

Exercitiul 10: Implementati exemplele de mai sus astfel:

- clasa HelloThread se declara ca si calsa imbricata static in clasa Xand0Player;
- metoda main se defineste ca parte a clasei Xand0Player.

Exercitiul 11: Modificati timpul de suspendare pentru thread-ul helloT2 de la 2 secunde la 5 secunde.Induceti o asteptare de 3 secunde folosind Thread.sleep (verificati API-ul pentru metoda sleep) dupa cele doua linii care apeleaza join(). Dupa ce se realizeaza sleep afisati un mesaj catre standard output. Executati exemplul de 2 ori: prima oara in varianta cu join() iar a doua oara comentati cele doua linii unde se invoca metoda join().

Sincronizarea thread-urilor

Thread-urile comunica intre ele prin partajarea accesului la variabile membre si referintele catre obiecte referite de variabile membre(i.e. campuri). Aceasta metoda de comunicatie reprezinta unul din principalele avantaje ale utilizarii trhead-urilor fata de procese insa poate induce doua tipuri de erori:

- *interferente inter-thread*
- *inconsistenta memoriei*

Pentru a evita aceste erori se foloseste un mecanism de ***sincronizare*** intre thread-uri.

Interferente inter-thread

Interferentele inter-thread apar atunci cand doua sau mai multe operatii, executate in thread-uri diferite, opereaza intretesut asupra aceluiasi set de date (i.e. deoarece operatiile prezinta mai multi pasi de executie acesti pasi pot sa fie intercalati astfel ca rezultatul final asupra datelor sa nu fie cel corect).

Fie variabila intreaga **color** asupra careia se vor executa operatiile:

color = color + 0x0F;

si

color = color – 0x0F;

Dupa cum se observa o operatie presupune cel putin trei pasi: citirea valorii curente a variabilei **color**, scaderea unei valori din variabila si, in final, stocarea acesteia la locatia de memorie

corespunzatoare variabilei **color**.

Presupunem doua threaduri T1 si T2 care executa cele doua operatii. In cazul in care pasii acestor operatii se interpun datorita modului de executie al threadurilor poate aparea urmatorul scenariu (se presupune **color** = 155**)**:

1. T1 citeste **color**
2. T2 citeste **color**
3. T1 aduna 15 la valoarea citita
4. T2 scade 15 din valoarea citita
5. T1 stocheaza noua valoare; **color** = 170
6. T2 stocheaza noua valoare; **color** = 140

Dupa cum se observa executia thread-ului T1 este pierduta deoarece rezultatul calculat in urma aplicatii operatiei nu este pastrat.

Pentru a exemplifica acest fenomen vom transforma aplicatia XAnd0 intr-o aplicatie multithreaded.

Analiza cerintelor pentru XAnd0 *multihreaded*:

- entitatile jucator Xand0Player erau utilizate doar ca obiecte care stocau informatie despre jucatori; pentru a induce o implementare mai realista jucatorii vor fi implementati ca thread-uri de executie care realizeaza selectarea noii poizitii pe tabla si executa mutarea corespunzatoare;
- deoarece Xand0Player deriveaza clasa Player singurul mecanism ce poate fi utilizat pentru a face trecerea catre un thread este implementarea interfetei Runnable:

```
public class Xand0Player extends Player implements Runnable {
   ...
   @Override
   public void run() {...}
}
```

- deaorece jucatorul actioneaza acum asupra tablei de joc – i.e. asupra mediului in care se executa jocul – acesta are nevoie de o referinta la mediul jocului Xand0Environment; pentru aceasta:
  - se adauga o variabila membra privata Xand0Environment env
  - se adauga 4 constructori similari celor deja existenti in care se adauga un parametru necesar initializarii variabilei env; motivul pentru care se pastreaza si constructorii fara parametrul de intializare a mediului este pentru pastrarea vechii functionalitati (fara trhead-uri)

```
protected Xand0Environment env;

// explicit constructor

public Xand0Player(String name, Integer playerID, Xand0Environment env)
{
      this.name = name;
      this.playerID = playerID;
      this.age = new Double(0);
      this.env = env;

}
```

- deoarece jucatorul este cel care executa mutarea, este necesara o metoda de obtinere a datelor de la utilizator:

```
public Point selectNextPosition() {
```

```
            System.out.println("Player " + name + ": Choose next position: ");
            Scanner scanner = new Scanner(new BufferedInputStream(System.in));
            int x = scanner.nextInt();
            int y = scanner.nextInt();
            return new Point(x,y);
        }
```

- la nivelul metodei run:
  - se executa urmatorii pasi atata timp cat nu s-a atins finalul jocului;
  - se alege noua mutare prin obtinerea unei pozitii corecte pe tabla de joc de la utilizator; se foloseste metoda selectNextPosition
  - folosind mediul se executa mutarea

```
while(!env.isEndOfGame()) {
      Point selectedMove = selectNextPosition();
      env.doNextMove(selectedMove, pawn, this);
}
```

- se observa ca la nivelul mediului s-a supraincarcat metoda doNextMove prin completarea singaturii cu inca 2 parametri; se pastreaza in Xand0Environment atat metoda doNextMove(Point p) si se adauga supraincarcarea acesteia:

```
public void doNextMove(Point position, Pawn<Double> pawn, Xand0Player
currentPlayer) {
   board.getCells().get(position).consume(pawn.getValue());
   if(gameLogic.isWinerMove(board, currentPlayer, position)) {
         winner = players.indexOf(currentPlayer);
         currentPlayerTurn = -1;
   } else if (counter == board.getCells().size()){ // draw
         winner = players.size();
         currentPlayerTurn = -1;
   }/* else {
         currentPlayerTurn = (currentPlayerTurn + 1) % players.size();
   }*/
}
```

- Pentru a executa noua implementare se utilizeaza clasa TestThreadGameLogic din pachetul xand0.test:

```
public class TestThreadGameLogic {

      public static void main(String[] args) throws IOException,
InterruptedException {
            Xand0Environment env = new Xand0Environment(3, 3, 3);
            Xand0Player player1 = new Xand0Player("alex", 0, env, new
Pawn("X",155,0.0));
            Xand0Player player2 = new Xand0Player("radu", 0, env, new
Pawn("0",55,1.0));

            env.addPlayer(player1);
            env.addPlayer(player2);

            Thread tp1 = new Thread(player1);
            Thread tp2 = new Thread(player2);

            tp1.start();
            tp2.start();

            tp1.join();
```

```
            tp2.join();

            displayBoard(env.getBoard());

            Xand0Player winner = env.getWinner();
            if(winner == null)
                  System.out.println("It's a draw!!");
            else
                  System.out.println("Player " + winner.getName() + " is the
winner!");
      }

      public static void displayBoard(Xand0Board board) {

            // assertion that the board is a square matrix
            int size = (int)Math.sqrt(board.getCells().size());
            Xand0BoardCell[][] result = new Xand0BoardCell[size][size];
            for(Entry<Point,Xand0BoardCell> cell: board.getCells().entrySet()) {
                  result[cell.getKey().x][cell.getKey().y] = cell.getValue();
            }

            for(int i=0;i<size;i++) {
                  System.out.println("");
                  for(int j=0;j<size;j++)
                        System.out.print("\t " + result[i][j].getStatus());
            }
      }
}
```

Se observa la rulare ca operatiile se intercaleaza ceea ce face ca executia jocului sa fie eronata.

Inconsistenta memoriei

Erorile de incosistenta a memoriei apar atunci cand thread-uri diferite prezinta vederi diferite asupra aceleasi date. Pentru a evita astfel de erori trebuie asigurat mecanismul *happens-before*.

Fie threadurile T1 si T2 care partajeaza variabila intreaga **color** initializata la valoarea 0.

Threadul T1 realizeaza urmatoarea operatie asupra variabilei:

color += 255;

La un interval de timp foarte scurt dupa lansarea acestei operatii threadul T2 realizeaza operatia:

System.out.println(**color)**;

In acest context daca operatia trheadului T1 a fost realizata atunci T2 va afisa pe ecran 255 altfelv-a afisa 0. Pentru a asigura case observa in orice conditii valoarea actualizata, trebuie sa se impuna o relatie de tip *happens-before* intre cele 2 operatii.

Unul din mecanismele care asigura o astfel de relatie este mecanismul de sincronizare.

Metode sincronizate

Pentru a realiza o metoda sincronizata se adauga cuvantul cheie *synchronized* in declaratia acesteia:

```
      public synchronized void doNextMove(Point position, Pawn<Double> pawn,
Xand0Player currentPlayer) {
            board.getCells().get(position).consume(pawn.getValue());
            if(gameLogic.isWinerMove(board, currentPlayer, position)) {
                  winner = players.indexOf(currentPlayer);
                  currentPlayerTurn = -1;
            } else if (counter == board.getCells().size()){ // draw
```

```
                winner = players.size();
                currentPlayerTurn = -1;
            }/* else {
                currentPlayerTurn = (currentPlayerTurn + 1) % players.size();
            }*/
        }
```

Sincronizarea metodei doNextMove este importanta atata timp cat doua sau mai multe thread-uri Xand0Player pot sa o acceseze simultan prin intermediul referintei catre mediul jocului,obiectul de tip Xand0Environment.

Caracteristici ale metodelor sincronizate:

- este imposibil ca doua invocari asupra unor metode sincronizate ale aceluiasi obiect sa se execute intretesut
- la finalizarea unei metode sincronizare se asigura o relatie happens-before cu orice invocare ulterioara a unei metode sincronizate a aceluiasi obiect;
- asigura un mecanism simplu pentru evitarea cazurilor simple de interferenta inter-trhead si incositenta a memorie.

Pentru a intipari mai bine mecanismul metodelor sincronizate consideram implementarea exemplului de mai sus:

```
        public static class SyncronizedColorUpdater {

            private int color = 0;

            public synchronized void add(String threadName) {
                color += 0x0f;
                System.out.println("Added " + 0x0f + " to color by " +
threadName);
            }

            public synchronized void subtract(String threadName) {
                color -= 0x0f;
                System.out.println("Subtracted " + 0x0f + " to color by " +
threadName);
            }

            public synchronized void displayColor(String threadName) {
                System.out.println("Displaying color by " + threadName + " : " +
color);
            }
        }

        public static void main(String[] args) throws InterruptedException {
            final SyncronizedColorUpdater scu = new SyncronizedColorUpdater();

            Thread T1 = new Thread(new Runnable() {
                @Override
                public void run() {
                    scu.add("T1");
                }
            });
            Thread T2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    scu.subtract("T2");
```

```
            }
        });

        Thread T3 = new Thread(new Runnable() {
            @Override
            public void run() {
                scu.displayColor("T3");
            }
        });

        T3.start();
        T1.start();
        T2.start();

        T1.join();
        T2.join();
        T3.join();
    }
```

Mecanismul de sincronizare Guarded Blocks

Dupa cum s-a observat, prin executia proiectului XAnd0 modificat cu specificatiile de mai sus, nu s-a obtinut o sincronizare suficienta intre thread-urile jucator astfel incat executia sa fie coerenta.

Unul dintre cele mai populare mecanisme de sincronizare il reprezina un *guard block.* Un guard block reprezinta un bloc de instructiuni care incepe cu verificarea unei conditii ce trebuie sa fie adevarata astfel incat urmatoarele instructiuni ale blocului sa poata fi executate. O astfel de metoda poate fi simplu obtinuta prin utilizarea unei bucle while insa nu este un mecanism eficient si economic de utilziare a resurselor de calcul disponibile:

```
    public void guardedEOG() {
        while(!EOG) {
            System.out.println("End of Game not reached");
        }
    }
```

O metoda mai eficienta consta in utilizarea metodei Object.wait pentru a suspenda thread-ul curent. Invocarea acestei metode nu returneaza pana cand un al thread nu a lansat o notificare ca urmare a detectie unui eveniment special. Pentru a lansa aceasta notificare se apeleaza metoda Object.notify() sau Object.notifyAll() - se consulta documentatia API pentru a constata diferenta intre cele doua metode.

```
    public synchronized void guardedEOG() {
        while(!EOG) {
            try {
                System.out.println("End of Game not reached.. waiting");
                wait();
            } catch(InterruptedException e) {

            }
        }
        System.out.println("End of Game reached!!");
```

Exercitiul 12: Explicati de ce metoda gaurdedEOG este o metoda sincronizata.

Pentru a realiza sincronizarea jucatorilor pentru XAnd0 trebuie sa se asigure urmatoarea logica de executie:

a) unul din jucatori executa in timp ce altul este suspendat

b) dupa ce jucatorul care executa a terminat de efecutat mutarea corespunzatoare trebuie sa notifice celalalt jucator iar apoi sa isi suspende executia.

Pentru a implementa aceasta logica:

- se va implementa o clasa GuardedBlock care implementeaza doua metode: lock si unlock; clasa face parte dintr-un nou pachet xand0.util:

```
package xand0.util;

	public class GuardedBlock {
		boolean locked = false;

		public synchronized void lock() throws InterruptedException
	{
			if(locked)
				wait();
			else
				locked = true;
		}

		public synchronized void unlock() {
			notifyAll();
		}
	}
```

- se intializeaza o variabila membra protected GuardedBlock la nivelul Xand0Environment

```
public class Xand0Environment {
  ...
  protected GuardedBlock guard;
  ...
}
```

  Se adauga initializarea corespunzatoare in constructorii clasei Xand0Environment

- se modifica metoda run a Xand0Player corespunzator:

```
	@Override
	  public void run() {
		while(!env.isEndOfGame()) {
			try {
				env.guard.lock();
				Point selectedMove = selectNextPosition();
				env.doNextMove(selectedMove, pawn, this);
				env.guard.unlock();
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
```

Exercitiul 13: Implementati urmatoarea functionalitate:

- Xand0Environment devine un thread de executie
- un thread de tipul Xand0Environment monitorizeaza monitorizeaza din mutare in mutare finalizarea jocului;
- in caz de finalizare Xand0Environment anunta castigatorul sau remiza si isi finalizeaza

executia

- indicii: threadul Xand0Environment trebuie sa prezinte inca un guarded-block pentru a putea sa se sincronizeze cu jucatorii; astfel cand un jucator executa celalalt jucatoru si mediul sunt suspendati; dupa ce un jucator a realizat mutarea este activat trhead-ul mediului pentru ca acesta sa verifice noua stare a tablei de joc si a decide daca s-a ajuns la remiza sau finalizarea jocului; dupa ce mediul a verificat conditiile, in cazul in care jocul nu s-a terminat activeaza din nou jucatorii (care prin logica implementata deja isi asigura alternanta in mutarile pe tabla de joc).

NOTA: O descriere detaliata a tuturor conceptelor precedente se poate consulta la adresele 1, 2, 3 si 4.