# Esercizio 1

Si consideri un sistema software $\mathcal{S}_1$ che contenga la seguente interfaccia Java utilizzata per gestire delle *attività*:

```
package it.unipr.informatica.exercise;

public interface Activity {
  public Thread perform(Object o);
}
```

Scrivere il codice sorgente della classe `it.unipr.informatica.exercise.ActivityImpl` utilizzando unicamente le classi e le interfacce del package `java.lang` e rispettando le seguenti specifiche:

1. La classe `ActivityImpl` implementa l'interfaccia `Activity`;
2. La classe `ActivityImpl` aggiunge un metodo privato `print`, con un unico argomento di tipo `Object`, che scrivere a video il proprio argomento;
3. Il thread ritornato dalla chiamata `perform(o)`, una volta attivato mediante `start` dopo essere stato ritornato, esegue in *mutua esclusione* la chiamata `print(o)`; e
4. La mutua esclusione garantita da un thread ritornato dalla chiamata `w.perform(o)` coinvolge *tutti e soli* i thread ritornati dalle chiamate a `perform` sull'oggetto `w`.

Scrivere anche il codice sorgente della classe `it.unipr.informatica.exercise.Exercise01` che permetta di avviare un esempio di uso della classe `ActivityImpl`.

# Esercizio 2

Si consideri un sistema software $\mathcal{S}_2$ che contenga la seguente interfaccia Java utilizzata per gestire delle *attività*:

```
package it.unipr.informatica.exercise;

public interface Task {
  public void perform();
}
```

Si consideri la seguente interfaccia Java che utilizza l'interfaccia `Task`:

```
package it.unipr.informatica.exercise;

public interface Launcher {
  public void start(Task[] tasks);
}
```

Il metodo `start` dell'interfaccia `Launcher` è pensato per eseguire tutte le attività in modo concorrente usando un thread per ogni attività e ritornando immediatamente appena termina una delle attività. Ad esempio, se il metodo `start` viene chiamato con un argomento che contiene tre attività, queste tre attivià vengono eseguite in tre thread indipendenti e il metodo `start` termina appena una qualsiasi delle tre attività termina.

Scrivere il codice sorgente di un'implementazione dell'interfaccia `Launcher`, che si chiamerà `it.unipr.informatica.exercise.LauncherImpl`, utilizzando unicamente le classi e le interfacce del package `java.lang` e rispettando la specifica del metodo `start` descritta in precedenza. In più, scrivere il codice sorgente della classe `it.unipr.informatica.exercise.Exercise02` che permetta di avviare un esempio di uso della classe `LauncherImpl`.

# Esercizio 3

Si consideri un sistema software $\mathcal{S}_3$ che contenga la seguente interfaccia Java utilizzata per gestire delle *attività*:

```
package it.unipr.informatica.exercise;

public interface Task {
  public void run(Object startedMutex);
}
```

L'attività viene svolta all'interno del metodo `run` e l'argomento del metodo viene utilizzato per segnalare che l'attività è effettivamente iniziata. Infatti, un'attività ha sempre una fase preparatoria in cui non è possibile considerarla effettivamente iniziata. Quindi, è richiesto che le implementazioni del metodo `run` notifichino il proprio argomento appena sia possibile considerare l'attività effettivamente iniziata.

Si consideri la seguente interfaccia Java che utilizza l'interfaccia `Task`:

```
package it.unipr.informatica.exercise;

public interface Executor {
  public void launch(Task[] tasks);
}
```

Il metodo `launch` dell'interfaccia `Executor` è pensato per eseguire tutte le attività in modo concorrente e ritorna appena tutte le attività sono effettivamente iniziate.

Scrivere il codice sorgente di un'implementazione dell'interfaccia `Executor`, che si chiamerà `it.unipr.informatica.exercise.ExecutorImpl`, utilizzando unicamente le classi e le interfacce del package `java.lang` e rispettando la specifica del metodo `launch` riportata in precedenza. In più, scrivere il codice sorgente della classe `it.unipr.informatica.exercise.Exercise03` che permetta di avviare un esempio di uso della classe `ExecutorImpl`.

# Esercizio 4

Si consideri un sistema software $\mathcal{S}_4$ che contenga degli oggetti *launcher* e degli oggetti *monitor* utilizzabili, rispettivamente, per attivare l'esecuzione di metodi in thread dedicati e per mettersi in attesa della loro terminazione. Gli oggetti launcher implementano la seguente interfaccia Java:

```
package it.unipr.informatica.exercise;

public interface Launcher {
  public Monitor launch(Runnable r);
}
```

Gli oggetti monitor implementano la seguente interfaccia Java:

```
package it.unipr.informatica.exercise;

public interface Monitor {
  public void await() throws InterruptedException;
}
```

Scrivere il codice sorgente della classe `it.unipr.informatica.exercise.LauncherImpl` che può essere utilizzata per creare oggetti launcher secondo la seguente specifica:

1. Il metodo `launch` attiva un thread per ogni chiamata per eseguire il *corpo* del proprio argomento;
2. Il metodo `launch` ritorna *immediatamente* un oggetto che implementa `Monitor`;
3. È possibile utilizzare il monitor ritornato dal metodo `launch` per mettersi in attesa, mediante il metodo `await`, della terminazione del corpo dell'argomento passato al metodo `launch`.

Si noti che un oggetto monitor si dice *sbloccato* se il metodo ad esso associato è terminato e, quindi, se una chiamata ad `await` non ha più alcun effetto.

Il sistema $\mathcal{S}_4$ mette a disposizione anche degli oggetti *monitor set* che implementano l'interfaccia `MonitorSet` del package `it.unipr.informatica.exercise`. L'interfaccia `MonitorSet`, e la relativa implementazione `it.unipr.informatica.exercise.MonitorSetImpl` da realizzare, offrono i seguenti metodi:

1. Il metodo `add`, con un argomento di tipo `Monitor`, che aggiunge un monitor al monitor set e ritorna `true` se il monitor è stato effettivamente aggiunto; e
2. Il metodo `await`, che si mette in attesa che almeno uno dei monitor del monitor set risulti sbloccato, eventualmente lanciando una `InterruptedException` nei casi opportuni.

Per lo svolgimento dell'esercizio, si ipotizza che nel sistema $\mathcal{S}_4$ sia presente una classe `Exercise04` nel package `it.unipr.informatica.exercise`. Il metodo principale della classe `Exercise04` crea un monitor set, aggiunge al monitor set gli $N = 100$ monitor ottenuti dalle $N$ richieste di esecuzione del metodo `work` presente nella classe `Exercise04` e poi si mette in attesa che tutti gli $N$ monitor risultino sbloccati. Il metodo `work` ha il seguente comportamento:

1. Viene generato un intero casuale $0 \leq K < 100$;
2. Viene eseguita un'attesa di $100 + K$ millisecondi; e
3. Viene stampato $K$ a video.

# Esercizio 5

Si consideri un sistema software $\mathcal{S}_5$ che contenga $R > 0$ *risorse* numerate da 0 a $R - 1$ mediante un identificativo univoco. Ogni risorsa è un oggetto che implementa la seguente interfaccia:

```
package it.unipr.informatica.exercise;

public interface Resource {
  public int getID();
  public int use();
  public void release();
}
```

Le risorse nel sistema vengono tutte create da un *manager delle risorse*. Ogni risorsa è inizialmente in stato *libera* e passa in stato *acquisita* solo quando il manager delle risorse la restituisce mediante il metodo:

```
public Resource[] acquire(int id);
```

che ritorna un array che contiene le tre risorse con identificativi `id`, `(id+1) % R` e `(id+2) % R`, purché tutte e tre le risorse siano in stato libera. Se almeno una delle tre risorse non è in stato libera, allora il metodo `acquire` si mette in attesa che tutte e tre le risorse siano libere. Una risorsa ritorna in stato libera solo quando viene chiamato il metodo `release` sulla risorsa. Solo le risorse in stato acquisita possono essere usate mediante il metodo `use`. Il metodo `use` chiamato su una generica risorsa con identificativo univoco `id` si limita a ritornare un numero casuale intero tra `id` e `id+99`, estremi inclusi.

Il sistema $\mathcal{S}_5$ contiene anche $W = R$ *worker* pensati per essere eseguiti con il massimo grado possibile di parallelismo. Ogni worker è numerato mediante un identificativo univoco tra 0 e $W - 1$. Un generico worker con identificativo univoco `id` è un oggetto con un proprio thread di esecuzione che ciclicamente chiede al manager delle risorse di acquisire tre risorse mediante `acquire(id)`. Una volta acquisite le tre risorse, il worker le usa invocando il seguente metodo:

```
public static void useAndPrint(Resource r1, Resource r2, Resource r3) {
  int t = r1.use() + r2.use() + r3.use();

  System.out.println(t);
}
```

della classe `it.unipr.informatica.exercise.Exercise05`, che contiene anche il metodo principale di $\mathcal{S}_5$. Il ciclo di esecuzione di ogni worker prevede che dopo ogni chiamata a `useAndPrint`, il worker liberi le risorse utilizzate e si metta in attesa per 100 millisecondi. Si noti che la classe `Exercise05` permette di attivare $\mathcal{S}_5$ nell'ipotesi $W = R = 9$.

# Esercizio 6

Si consideri un sistema software $\mathcal{S}_6$ che contenga degli oggetti *barrier* che implementano la seguente interfaccia:

```
package it.unipr.informatica.exercise;

public interface Barrier {
  public void add(Object object);
  public void remove(Object object);
  public void await() throws InterruptedException;
}
```

Un barrier ha le seguenti caratteristiche:

1. Quando viene chiamato `await`, il barrier si mette in attesa che uno qualsiasi degli oggetti che sono stati preventivamente aggiunti mediante `add`, e non ancora rimossi mediante `remove`, venga segnalato mediante una chiamata a `notify` o a `notifyAll`;

2. Quando viene chiamato `add`, l'oggetto passato come argomento viene considerato nelle attese attualmente in corso e in quelle future, prima che venga eventualmente rimosso mediante `remove`; e

3. Quando viene chiamato `remove`, l'oggetto passato come argomento non viene più considerato nelle attese attualmente in corso e in quelle future, a meno che non venga nuovamente aggiunto mediante `add`.

Quindi, un barrier permette di mettersi in attesa su un gruppo di oggetti e l'attesa viene terminata segnalando uno degli oggetti del gruppo.

Nel sistema $\mathcal{S}_6$ è presente un unico barrier che viene associato a $N = 50$ oggetti mediante il metodo `add`. In più, nel sistema è presente un oggetto *waiter* che è dotato di un proprio thread di esecuzione ed esegue ciclicamente le seguenti attività:

1. Si mette in attesa chiamando `await` sul barrier; e

2. Scrive a video il valore di un contatore che viene incrementato ad ogni iterazione del ciclo, partendo da uno.

Infine, nel sistema sono presenti $N$ oggetti *notifier*, ognuno dei quali è associato in modo univoco ad uno degli oggetti usati per le attese del barrier. Ogni notifier è dotato di un proprio thread di esecuzione che esegue ciclicamente le seguenti attività:

1. Viene chiamato `Thread.sleep` per operare un'attesa casuale uniformemente distribuita tra 0 e 100 millisecondi; e

2. Viene terminata l'attesa del barrier mediante il proprio oggetto.

Realizzare il sistema software $\mathcal{S}_6$ in Java aggiungendo tutte le classi necessarie e, in particolare, la classe `Exercise06` nel package `it.unipr.informatica.exercise`. Il metodo principale della classe `Exercise06` crea gli oggetti barrier, waiter e notifier e, quindi, attiva il sistema.

# Esercizio 7

Si consideri un sistema software $\mathcal{S}_7$ che contenga le seguenti quattro interfacce utilizzate per realizzare un modulo in grado di valutare espressioni aritmetiche:

```
package it.unipr.informatica.exam;

public interface Expression {
}
```

```
package it.unipr.informatica.exam;

public interface Number
    extends Expression {
  public double getValue();
}
```

```
package it.unipr.informatica.exam;

public interface Add
    extends Expression {
  public Expression getLeft();
  public Expression getRight();
}
```

```
package it.unipr.informatica.exam;

public interface Multiply
    extends Expression {
  public Expression getLeft();
  public Expression getRight();
}
```

Rispondere alle seguenti domande scrivendo codici sorgenti in Java che, se ritenuto utile, possono utilizzare unicamente il package `java.lang`:

1. Riscrivere i codici sorgenti delle quattro interfacce modificandoli opportunamente per renderli utilizzabili per valutare espressioni aritmetiche mediante il design pattern *visitor*. Non è necessario che un codice venga riscritto se non sono necessarie modifiche.

2. Scrivere i codici sorgenti delle interfacce che è necessario aggiungere a quelle scritte al punto precedente per completare la struttura del design pattern visitor.

3. Scrivere i codici sorgenti di tutte le classi necessarie ad implementare le interfacce scritte ai due punti precedenti, in particolare scrivendo anche il codice sorgente della classe `EvaluateVisitor` da utilizzare per valutare le espressioni aritmetiche.

# Esercizio 8

Si consideri il semplice linguaggio di programmazione funzionale $\mathcal{L}_8$ descritto dalla seguente grammatica in formato BNF:

$$
\begin{array}{rcl}
Script & \rightarrow & GlobalExpr^* \\
GlobalExpr & \rightarrow & DefExpr \mid Expr \\
DefExpr & \rightarrow & (\text{def } Id\ Expr) \\
Expr & \rightarrow & CondExpr \mid EvalExpr \mid ArithExpr \mid Func \mid Int \mid Id \\
CondExpr & \rightarrow & (\text{if } Expr\ Expr\ Expr) \\
EvalExpr & \rightarrow & (\text{eval } Expr\ Expr) \\
ArithExpr & \rightarrow & (+\ Expr\ Expr) \mid (-\ Expr\ Expr) \mid (*\ Expr\ Expr) \\
Func & \rightarrow & (\text{lambda } Id\ Expr) \\
Int & \rightarrow & \{\text{an integer}\} \\
Id & \rightarrow & \{\text{an identifier}\}
\end{array}
$$

La semantica informale dei costrutti del linguaggio $\mathcal{L}_8$ è la seguente:

1. Uno script è una sequenza di *GlobalExpr* che vengono valutate tutte nell'ordine in cui vengono scritte e il risultato di ogni valutazione viene stampato a video.
2. Una $DefExpr$ lega il suo primo argomento al risultato della valutazione del suo secondo argomento nel contesto di valutazione globale.
3. Una *CondExpr* valuta il suo terzo argomento se il suo primo argomento viene valutato a zero e valuta il suo secondo argomento altrimenti.
4. Una *EvalExpr* valuta il suo primo argomento e applica la funzione così ottenuta al suo secondo argomento.
5. Una *ArithExpr* valuta un'espressione aritmetica.
6. Una $Func$ definisce una funzione anonima con un solo argomento senza legare la funzione al suo contesto di valutazione (usando il dynamic scoping).
7. Un $Id$ viene sostituito dal valore a cui è legato nel contesto di valutazione.

Si noti che uno script termina tutte le volte che viene richiesta un'operazione non ben definita.

Il seguente è un esempio di script nel linguaggio $\mathcal{L}_8$ che definisce la funzione `fact` (il fattoriale di un numero) e poi valuta 5!.

```
(def fact
  (lambda n
    (if n
      (* n (eval fact (- n 1)))
      1
    )
  )
)

(eval fact 5)
```

Nell'ipotesi di disporre di un modulo $\mathcal{P}_8$ in grado di costruire una struttura ad oggetti partendo dal testo di uno script (analizzatore lessicale, analizzatore sintattico e analizzatore semantico), rispondere alle seguenti domande eventualmente utilizzando i package `java.lang`, `java.util` e `java.io`:

1. Scrivere le interfacce Java del package `it.unipr.informatica.exercise.l8.model` necessarie a $\mathcal{P}_8$ per descrivere uno script scritto in $\mathcal{L}_8$ sapendo già che verrà utilizzato il design pattern *visitor* con queste interfacce.

2. Scrivere una classe Java `it.unipr.informatica.exercise.l8.L8Interpreter` che utilizzi il design pattern visitor per interpretare uno script del linguaggio $\mathcal{L}_8$ ricevuto mediante oggetti che implementano le interfacce realizzate al punto 1.

3. Scrivere una classe Java `it.unipr.informatica.exercise.l8.L8SemanticAnalyzer` che utilizzi il design pattern visitor per individuare staticamente gli errori semantici presenti in uno script del linguaggio $\mathcal{L}_8$ ricevuto mediante oggetti che implementano le interfacce realizzate al punto 1 (ci si limiti agli errori semantici dovuti all'uso errato degli identificativi).

4. Scrivere una classe Java `it.unipr.informatica.exercise.l8.L8SyntacticAnalyzer` in grado compiere il lavoro di $\mathcal{P}_8$ partendo da un oggetto di classe `java.io.Reader`, eventualmente sfruttando la classe Java `it.unipr.informatica.exercise.l8.L8LexicalAnalyzer`, realizzata per spezzare il testo di uno script in $\mathcal{L}_8$ in stringhe interessanti.