НАЦІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ

ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК

КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ

МЕТОДИЧНІ ВКАЗІВКИ

щодо виконання лабораторних робіт
з курсу “Об'єктно-орієнтоване програмування”
Частина 2. Мова програмування JAVA

КИЇВ — 2009

Методичні вказівки щодо виконання лабораторних робіт з курсу “Об'єктно-орієнтоване програмування”. Частина 2. Мова програмування JAVA / Укл. Баклан І.В., Степанкова Г.А. - К.: НАУ, 2009. - 52 с.

ЗМІСТ

ЛАБОРАТОРНА РОБОТА 1: Встановлення та запуск Eclipse

Ціль роботи:

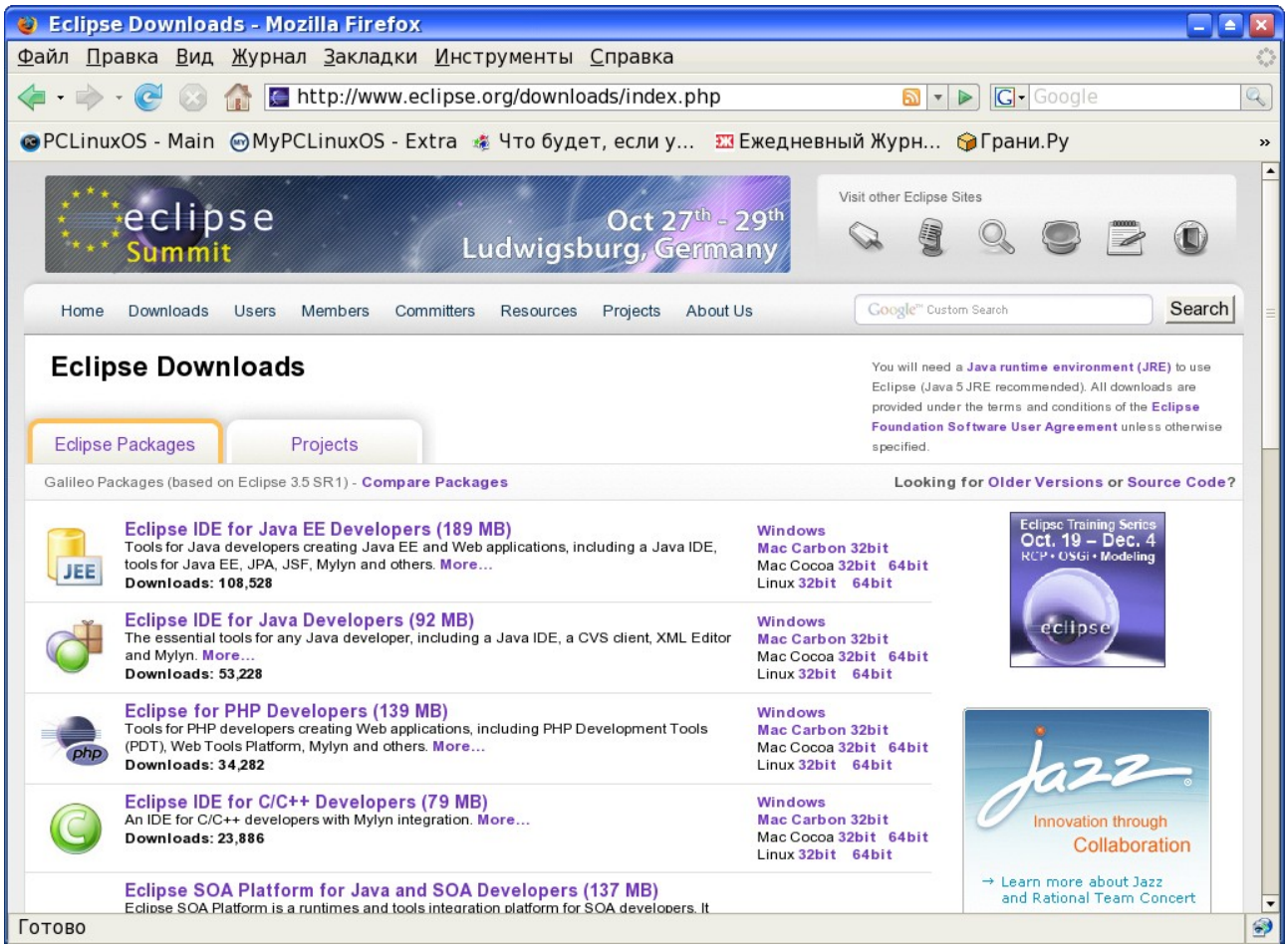
Eclipse - програмне забезпечення з відкритим кодом, яке в цілому використовується як Інтегроване Середовище Розробки (IDE) і як основа для програмних продуктів.

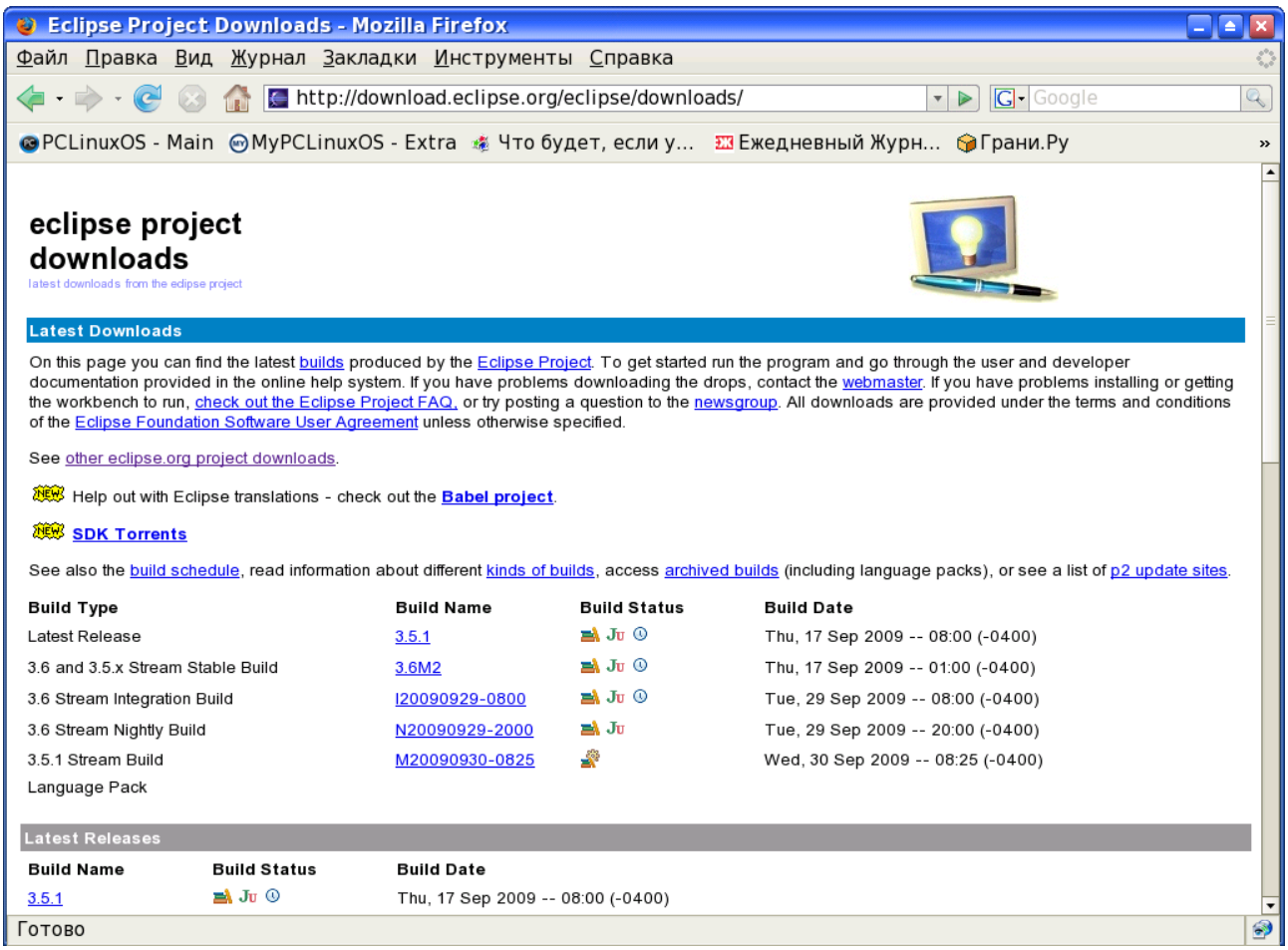
В цій лабораторній роботі ви будете інсталювати та запускати Eclipse. Перш, ніж ви почнете встановлювати його, продукт повинен бути завантажений із сайту www.eclipse.org. Після того, як ви завантажите файл, ви можете приступати до інсталяції. Коли інсталяція завершиться, ви можете запустити Eclipse.

Виконання роботи:

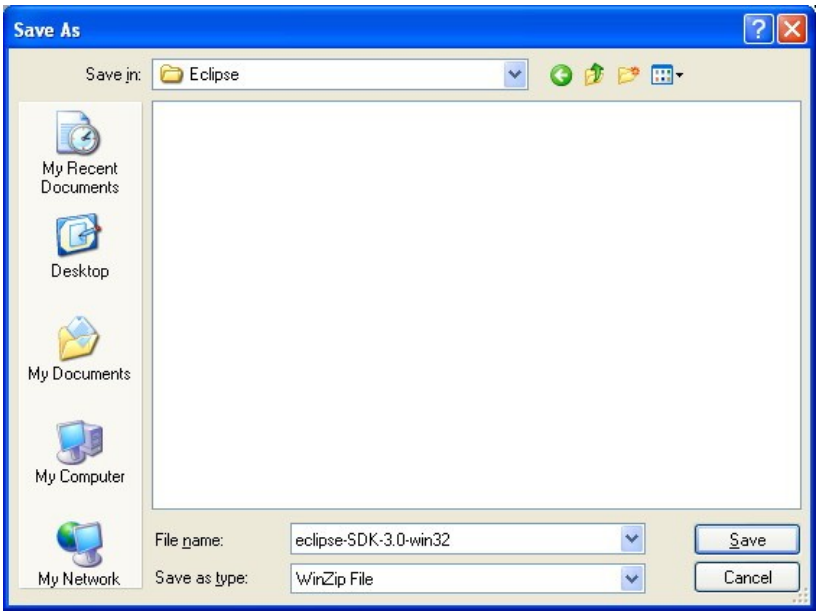
Крок 1: Завантаження Eclipse

Програмне забезпечення може бути завантажене із сайту www.eclipse.org/downloads/index.php. Із цієї сторінки виберіть відповідну географічну область (Північна Америка, наприклад) і виберіть пакет, який ви хочете завантажити. У більшості випадків ви захочете завантажити самий останній пакет. Клацніть на останньому пакеті, а потім виберіть платформу, на якій ви хочете інсталювати продукт.





Вам буде задане питання, що робити з файлом? Виберіть збереження файлу на локальному диску, а потім задайте місце для збереження файлу.

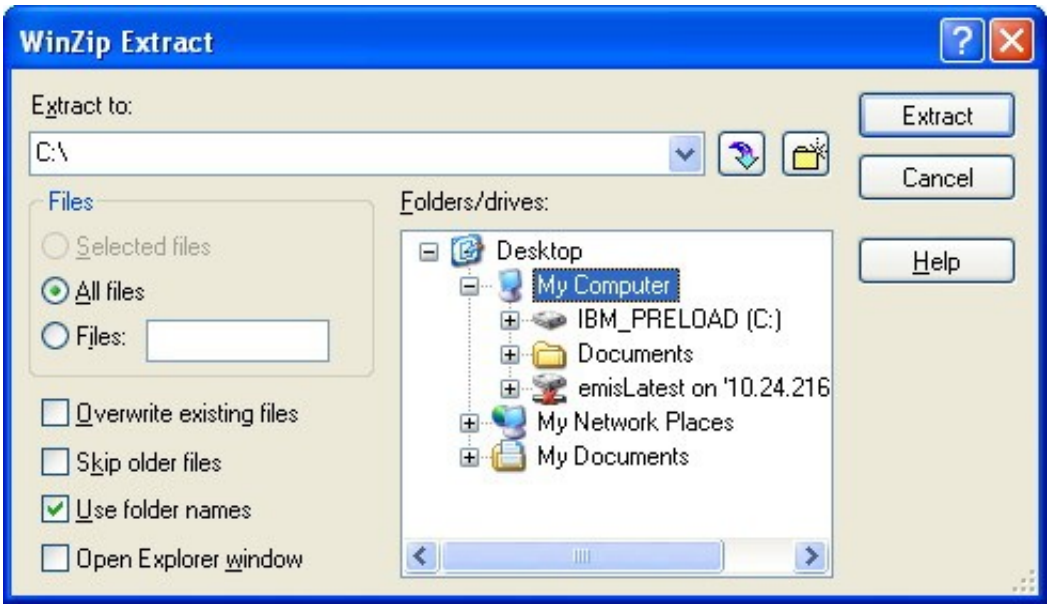


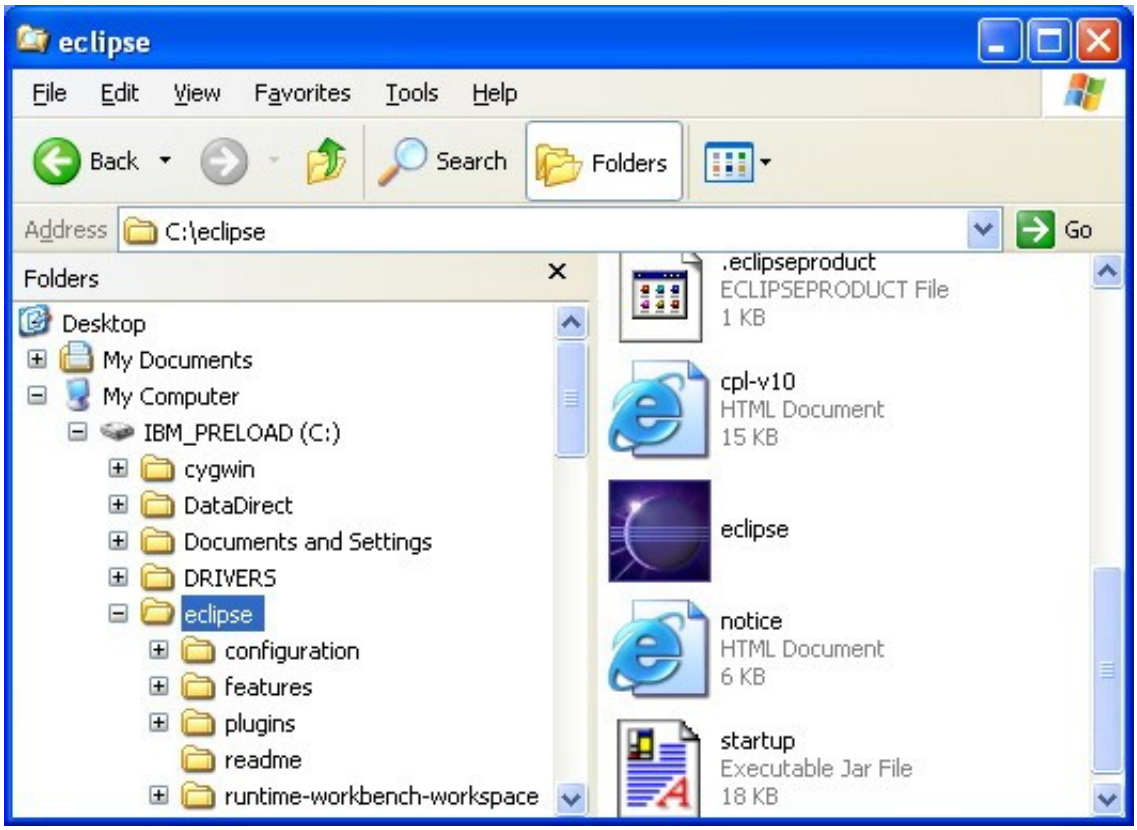
Тепер ви вивантажили Eclipse на вашу локальну машину.

Крок 2: Інсталяція Eclipse

Коли ви завантажили Eclipse на ваш локальний диск, ви можете приступитися до

інсталяції. Файл, що ви вивантажили - це ZIP-файл, що містить повне програмне забезпечення Eclipse. Щоб інсталювати Eclipse, ви просто розпакуйте вивантажений файл на локальний диск, наприклад, на диск C:\. Розпакування ZIP-файлу на диск C:\ призведе до створення каталогу C:\eclipse, що містить виконуваний файл Eclipse (eclipse.exe).



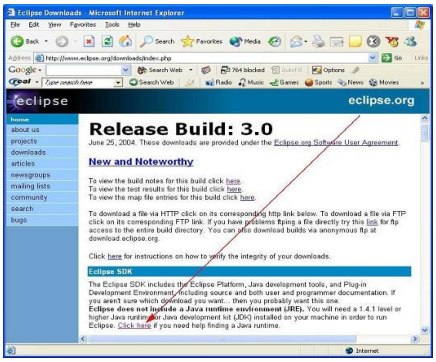


Крок 3: Запуск Eclipse

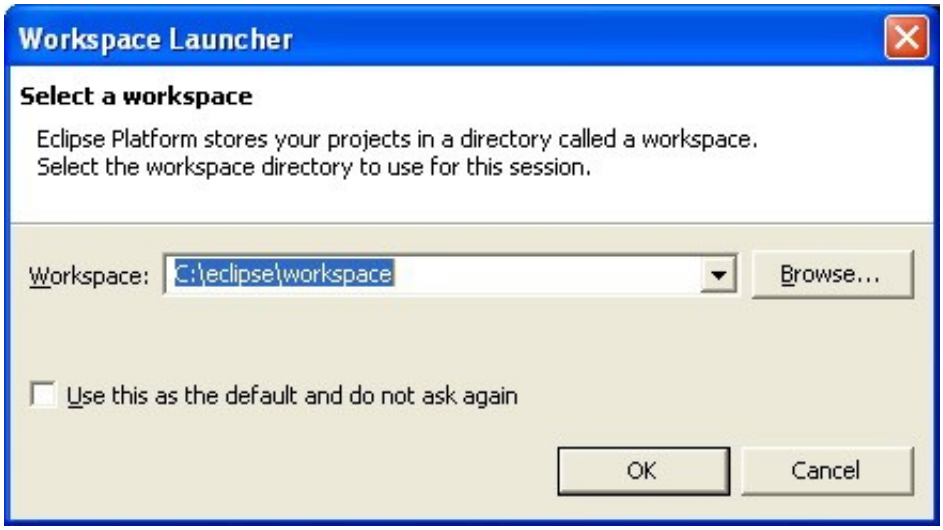
Клацніть двічі на файлі eclipse.exe, щоб запустити Eclipse. Оскільки Eclipse побудований на Java, програмне забезпечення вимагає для виконання Java runtime environment (JRE) або Java Development Kit (JDK) з javaw.exe. Якщо у вас не встановлений маршрут для javaw.exe, з'явиться діалогове вікно з повідомленням про відсутність JRE/JDK.

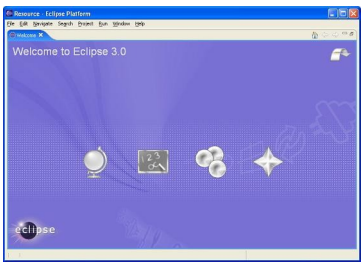
У такому випадку, встановіть змінну оточення JAVA_HOME, щоб вона вказувала на каталог вашої інсталяції JDK. Якщо на вашому локальному диску немає установки JDK або JRE, ви можете вивантажити JRE із сайту Eclipse (http://eclipse.org/downloads/index.php).

Коли ви встановили на локальному диску JRE, ви можете встановити змінну оточення JAVA_HOME, якщо це ще не зроблено.



Після того, як ви будете мати JAVA_HOME на місці й двічі клацніть на файлі eclipse.exe, інсталяція буде завершена запрошенням створити каталог робочого простору (workspace) - каталог робочого простору за замовчуванням є підкаталогом у каталозі інсталяції, наприклад, c:\eclipse\workspace - і з'явиться Робоче Місце (Workbench).





Висновки:

У цій роботі ви встановили та запустили Eclipse.

ЛАБОТАТОРНА РОБОТА 2: Об'єктно-орієнтована концепція. Гра життя

Ціль роботи:

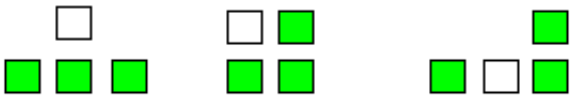
У цій роботі ви вивчите предметну область, у якій ви будете працювати всю іншу частину лабораторного практикуму. У цій роботі пояснюються правила Гри життя, яку ви спробуєте моделювати. Гра життя - одна з найбільш відомих 2-мірних моделей клітинної структури. Це не дійсна гра, тому що в ній немає гравців, немає переможців і переможених. Це гра, у якій початкова позиція й правила визначають усе, що трапиться після цього. Більше інформації про гру і її обґрунтування див. по посиланнях
http://www.math.com/students/wonders/life/life.html і
http://mathworld.wolfram.com/CellularAutomaton.html.

Правила Гри життя:

Гра життя відбувається на сітці із квадратних кліток - як шахівниця, - яка розширюється довільним образом в усіх напрямках. Клітка може бути ***живою*** або ***мертвою***.

Оцінка на клітці показує живу клітку. Порожня клітка є мертвою. Кожна клітка в сітці має вісім суміжних кліток в усіх напрямках, включаючи діагональні. Щоб виконати один крок гри, підрахуйте число живих сусідів для кожної клітки. Що трапиться далі - залежить від цього числа:

1. Правило народження: Мертва клітка, що має рівно три живих сусіди стає живою кліткою.



2. Правило виживання: Жива клітка із двома або трьома живими сусідами залишається жити.

3. Правило смерті: У всіх інших випадках клітка вмирає або залишається мертвою.

Число живих сусідів завжди обчислюється ***до*** застосування правила.

Виконання роботи:

Крок 1: Визначення об'єктів у грі

Подивіться на наведені вище вимоги гри та спробуйте визначити об'єкти, залучені в гру

(об'єкти звичайно визначаються пошуком у вимогах іменників).

Які об'єкти ви можете визначити в грі?

От кілька питань, які можуть вам допомогти визначити об'єкти:

- Як представляється сама гра?
- У чому полягає зміст гри?
- Що впливає на розвиток гри в часі?

Оскільки на питання, наведені вище, може бути кілька підходящих відповідей, можливо, правильним буде сказати, що найбільш очевидними об'єктами є **гра**, **дошка** й **правило**, і ці об'єкти становлять гру.

Крок 2: Визначення основних класів

На підставі результатів попереднього кроку, які класи становлять систему гри? Які імена класів, їхні дані та поводження?

Основними класами гри можуть бути Game, Board та Rule. Гра має дошку та ряд правил, а також індекс "покоління" для визначення того, скільки поколінь дошок відслідковується. Дошка має клітки, і вона ініціалізуються при створенні деякою конфігурацією живих і мертвих кліток. Правило може мати ім'я й тип та обчислюється для кожної клітки на дошці. Типом правила може бути: народження, виживання або смерть. Коли воно обчислюється, перевіряється тип правила, і на базі типу виконуються різні обчислення.

Висновки:

У цій роботі ви проаналізували предметну область гри, що ви реалізуєте в інших роботах.

ЛАБОРАТОРНА РОБОТА 3: Огляд Java

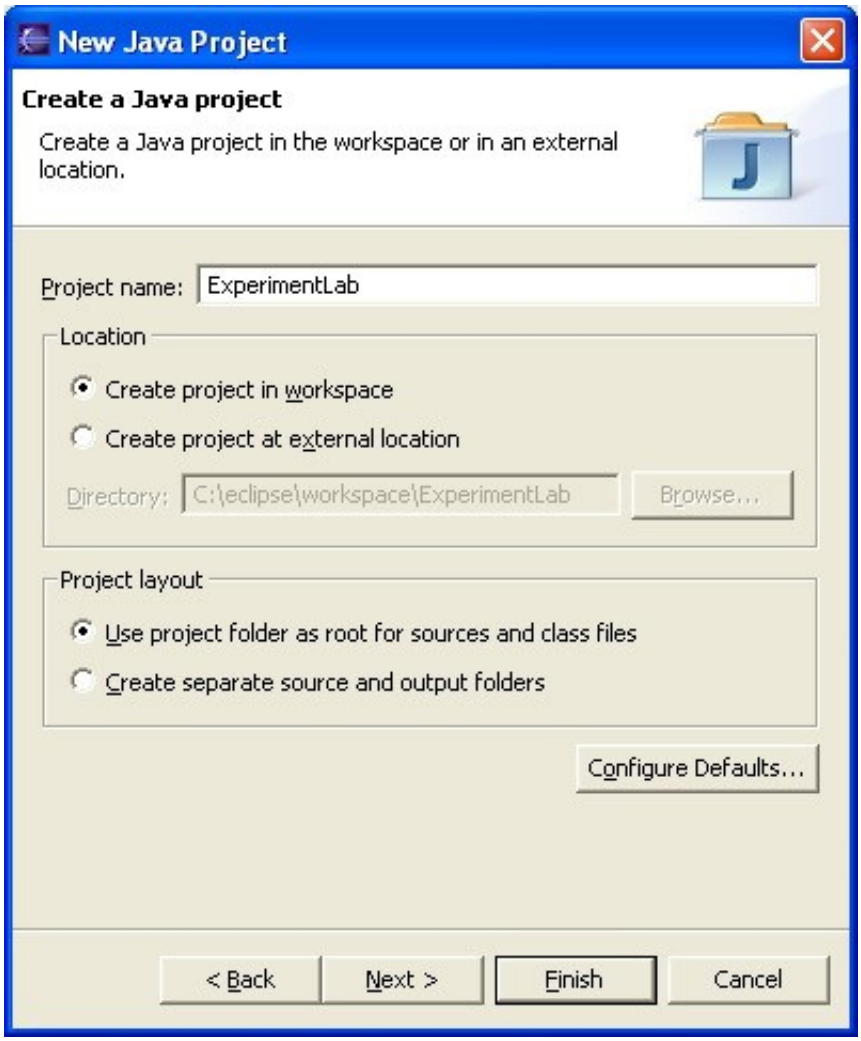
Ціль роботи:

В цій роботі ви будете використовувати Scrapbook для обчислення деякого Java-коду. Scrapbook використовується в Eclipse для інспектування та обчислення коду. Спочатку вам потрібно створити проект, а в проекті - новий Scrapbook. Це вправа ближче познайомить вас із середовищем Eclipse, а також з основним синтаксисом і правилами мови Java.

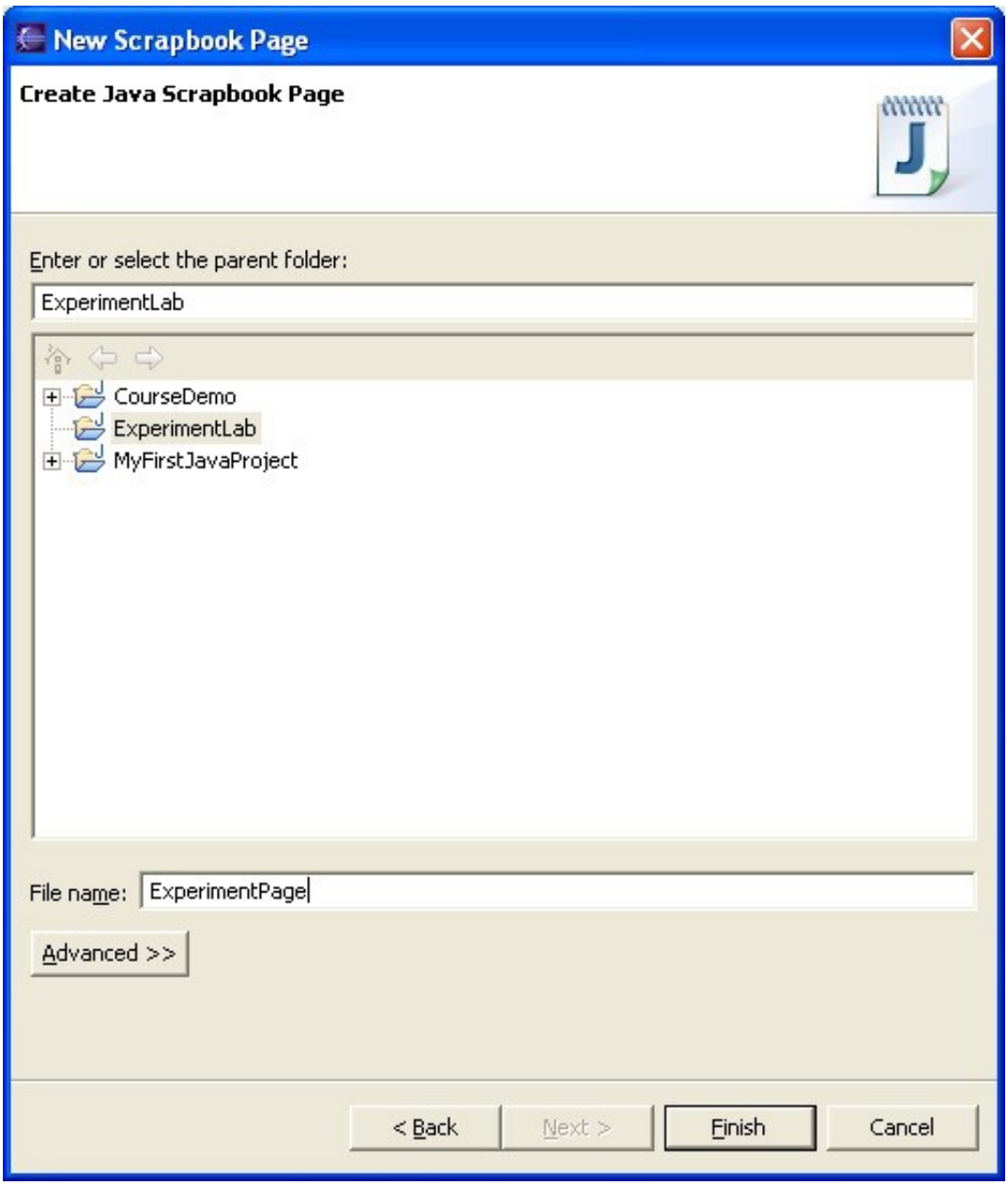
Виконання роботи:

Крок 1: Створення сторінки Scrapbook

У середовищі Eclipse виберіть File -> New -> Project: -> Java -> Java Project, задавши ім'я проекту ExperimentLab, і клацніть на Finish.



Створюється новий проект Java. З меню, що випадає, виберіть File -> New -> Other: -> Java -> Scrapbook Page, потім задайте ім'я сторінки (ExperimentgPage) з обраним проектом і клацніть на Finish.



Редактор сторінки відкриє сторінку.

Крок 2: Обчислення та відображення виразів

Напишіть та відобразіть результат обчислення наступного виразу:

5 + 2 * 10/4 - 7;

Який вийшов результат? Чи такий він, як ви очікували?

Відповідь: В обчисленні наведеного вище виразу спочатку обчислюється 2*10 з результатом 20 , а потім обчислюється 20/4 з результатом 5. Після обчислення 5+5-7 виходить фінальний результат 3.

Крок 3: Експерименти із примітивними типами та змінними

Напишіть і відобразіть результат обчислення наступних виразів:

int a = 6;

double b = a;

b;

Що відбулося? Чи перетворився int в double і чому?

Відповідь: Примітивний тип double - більш широкий тип, ніж int, і JVM у стані перетворити int в double автоматично, неявним перетворенням типів.

Тепер напишіть і відобразіть результат обчислення наступних виразів:

double a = 6;

int b = a;

b;

Що відбулося тепер? Чому ви одержали повідомлення "Type mismatch: cannot convert from **double** to **int**"? Що, якщо ви обчислите (**int**)b наприкінці?

Відповідь: Тип double не може бути перетворений у тип int, оскільки він більш широкий і неявне перетворення типів не виконується. За допомогою явного перетворення (**int**)b ви повідомляєте JVM, що можна привласнити більш широкий тип вузькому.

Напишіть і відобразіть результат обчислення наступних виразів:

long a = 45L;

double b = a;

b;

Який тепер результат обчислення виразів?

Напишіть і відобразіть результат обчислення наступних виразів:

double a = 45.00;

long b = a;

b;

Який тепер результат обчислення виразів? Чи одержали ви той же результат, що й при попередньому обчисленні? Чому?

Відповідь: Обидва типи мають однакову ширину. При обчисленні першого виразу тип long привласнюється double, і це обробляється JVM, оскільки double має більшу точність. Однак, у другому виразі тип більшої точності привласнюється типу меншої точності, і ви одержите помилку. Навіть якщо буде застосоване явне перетворення, JVM не зможе обробити його.

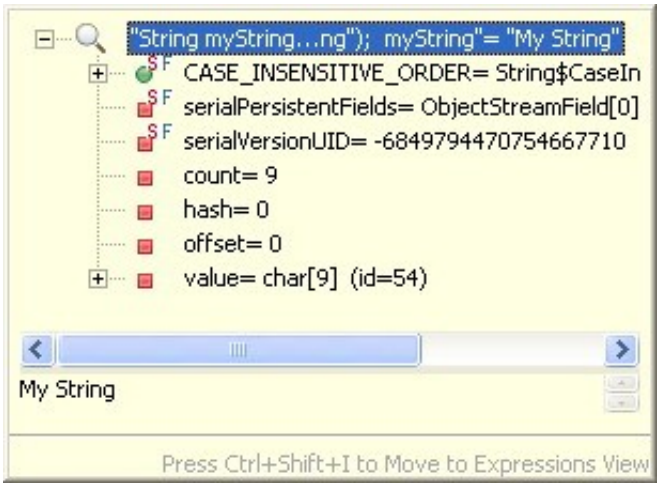
Шаг 4: Повідомлення та об'єкти

Проінспектуйте наступний вираз з Scrapbook:

String myString = **new** String("My String");

myString

Інспектор покаже наступне. Перегляньте всі символи, які становлять строку, що перевіряється.



Тепер обчисліть наступний вираз:

String myString = **new** String("My String");

System.out.println(myString);

Яке повідомлення ви посилаєте в об'єкт виводу? Де буде надрукований рядок?

Відповідь: println є посилкою повідомлення в об'єкт out. Строку друкується на стандартній консолі Java, що представляє в Eclipse подання Console.

Крок 5: Літерали

Проінспектуйте літерал "This is a string" в Scrapbook. Яким видом об'єкта він є? У чому розходження між літералом та обчисленням виразу **new** String("This is a string")?

Відповідь: "This is a string" є літеральним строковим об'єктом. Він - те ж, що й строковий об'єкт, створений за допомогою виразу **new** String("This is a string").

Крок 6: Масиви

Створіть і проінспектуйте масив з 5 цілих чисел, проініціалізованних нулями.

Створіть і проінспектуйте багатомірний масив з 5 масивів, створених на попередньому кроці.

Висновки:

У цій роботі ви використали Eclipse Scrapbook для обчислення й виконання простого коду Java.

ЛАБОРАТОРНА РОБОТА 4: Побудова класів Java

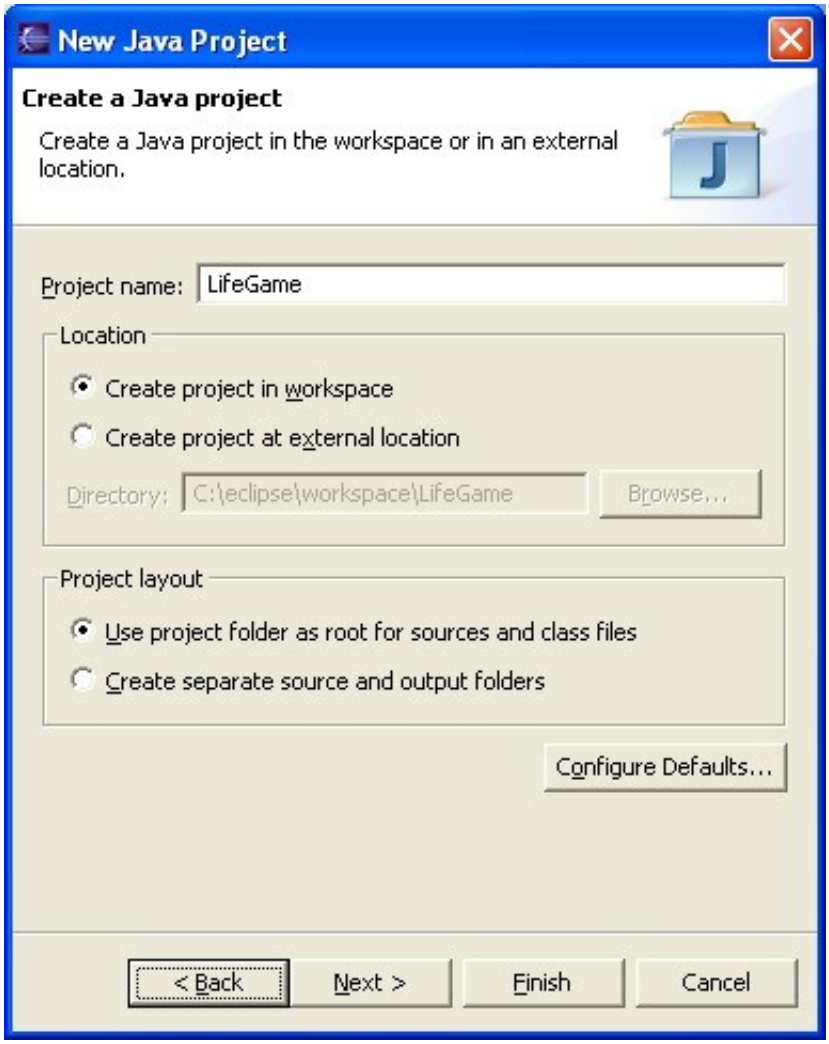
Ціль роботи:

У цій роботі ви ближче познайомитеся з Java Development Tools в Eclipse. Фактично, ви побудуєте основні класи гри. Після того, як ви закінчите цю роботу, ви повинні бути знайомі зі створенням класів за допомогою Eclipse, а також з використанням інструментів рефакторинга Java в Eclipse.

Виконання роботи:

Крок 1: Створення проекту

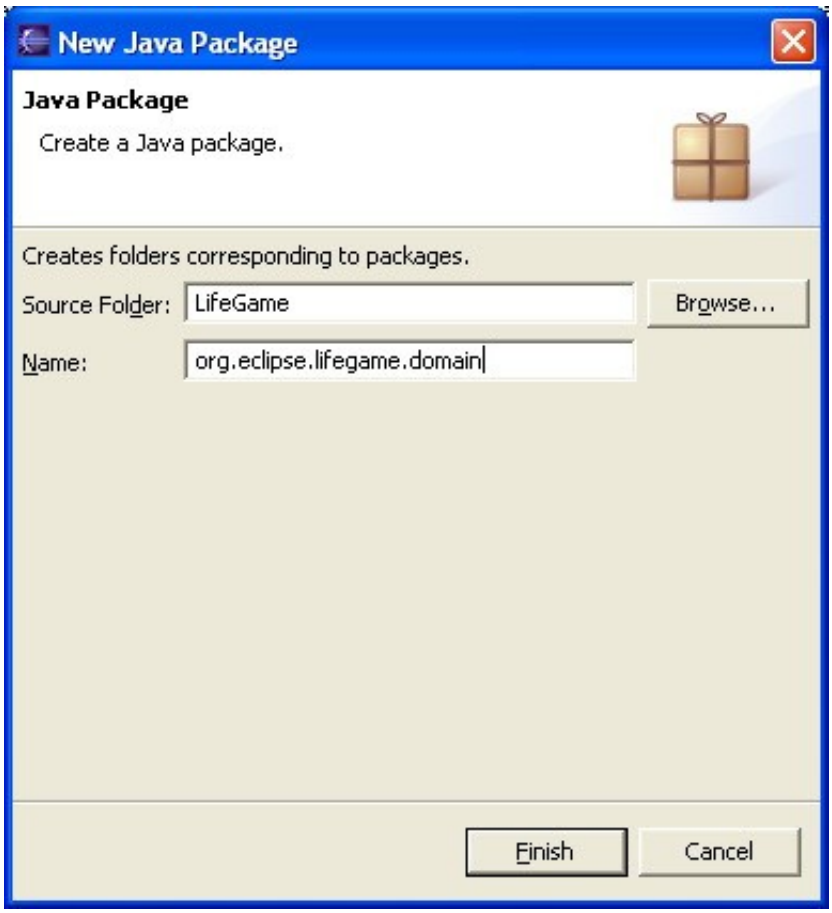
Створіть новий проект Java з ім'ям LifeGame. Для створення проекту виберіть із меню, що випадає: File -> New -> Project -> Java -> Java Project і клацніть на Next. Задайте ім'я проекту й клацніть на Finish.



Крок 2: Створення пакета

Створіть пакет, що буде містити прикладні класи для гри. Пакети групують класи по

функціональності та по їхніх просторах імен. Звичайно класи, які представляють модель, перебувають в одному пакеті, класи, які використовуються для тестування, перебувають в іншому пакеті й т.д. Для створення нового пакета виберіть проект і виберіть New -> Package з контекстного меню. Використайте як ім'я пакета org.eclipse.lifegame.domain і клацніть на Finish.



Крок 3: Створення класів Game та Board

Створіть клас Board, що буде мати закрите поле cells, що є 2-мірним масивом цілих чисел. Поле cells повинне бути 10x10 і містити після запуску гри нулі або одиниці.

Згенеруйте методи-акцесори для поля. Додайте конструктор класу Board, що буде приймати як параметр 2-мірний масив цілих чисел. Він буде представляти шаблон (початковий стан кліток) для гри, коли вона запуститься. Також замініть конструктор за замовчуванням, щоб просто ініціалізувати клітки в 2-мірному (10x10) масиві при створенні.

Визначте в класі два методи розгортання: evolve(Game) та evolve(Game,int). Ці методи повинні мати тип, що повертається, void та повинні проходити через клітки й на підставі правил розвивати наступну стадію. Метод розвитку, що приймає цілий параметр, розвиває дошку через багато стадій (залежно від параметра). Залишіть ці методи поки порожніми, ви реалізуєте їх у наступних роботах.

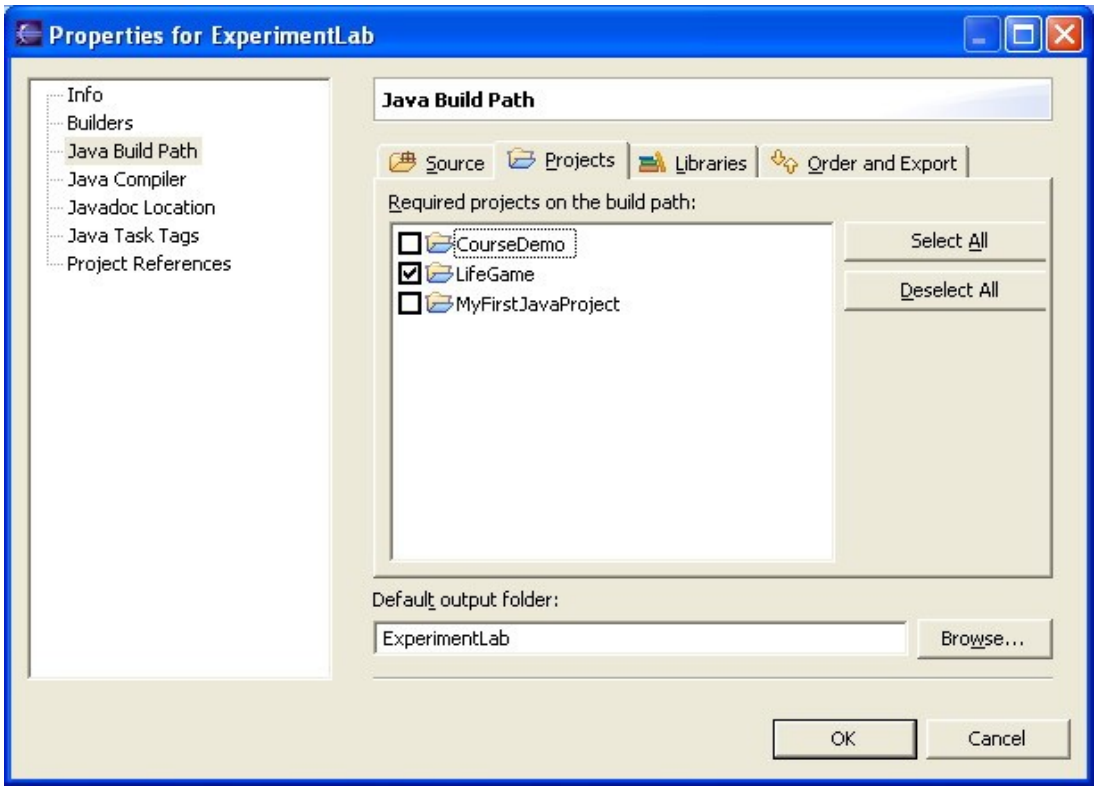
Замістіть метод toString() класу Board, щоб він просто повертав рядок "Board for game of life" . У наступних роботах ви додасте більш поводження в цей метод.

Створіть клас Game, що має поле board типу Board і поле generation типу int (яке показує скільки поколінь дошка розвивалася). Згенеруйте методи-акцесори для полів. Додайте конструктор класу Game, що буде приймати як параметр дошку й ініціалізувати поле board значенням параметра, а покоління - числом 1.

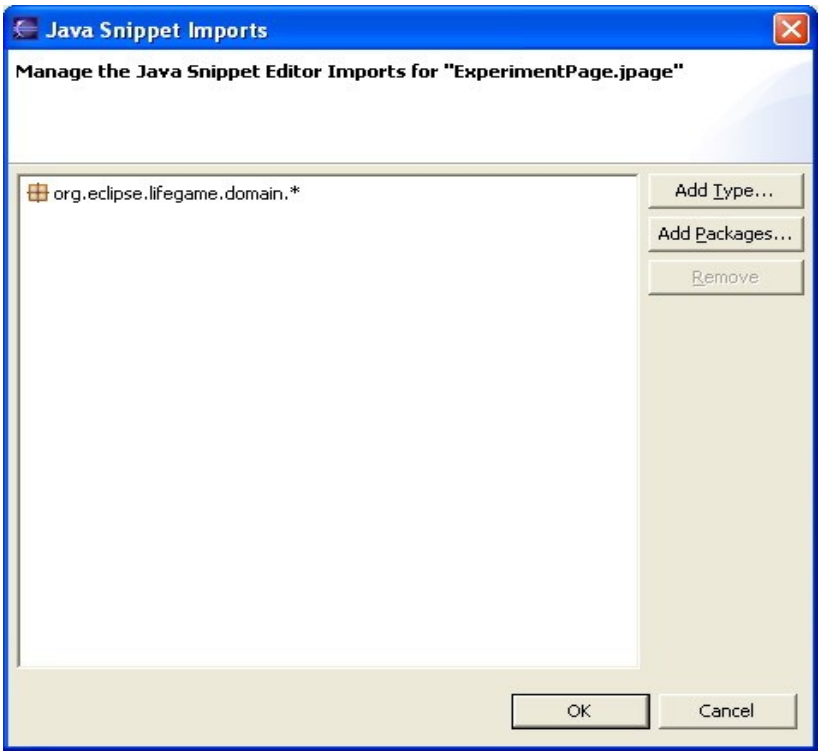
Шаг 4: Створення екземплярів Game та Board

Використайте Scrapbook з попередньої роботи для створення екземпляру класу Board. Класи не будуть видні в Scrapbook. Спочатку ви повинні додати проект LifeGame, як залежність, до проекту, що містить Scrapbook.

Виберіть проект ExperimentLab та виберіть Properties з контекстного меню. Клацніть на Java Build Path та виберіть закладку Projects. Виберіть проект LifeGame і клацніть на OK.



В Scrapbook виберіть Set Imports з контекстного меню й виберіть кнопку Add Packages:. Виберіть пакет, що містить класи, і клацніть на OK двічі.



Створіть екземпляр класу Game, передаючи йому створений об'єкт дошки. Проінспектуйте об'єкт гри.

В Scrapbook ви повинні одержати код, схожий на наступний:

```
int [][] boardCells = {
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1}};
Board board = new Board(boardCells);
Game game = new Game(board);
game
```

Висновки:

У цій роботі ви створили базові класи для Гри життя, і ви познайомилися з написанням конструкторів, методів і визначенням полів.

ЛАБОРАТОРНА РОБОТА 5: Використання відладчика

Ціль роботи:

В цій роботі ви ближче познайомитеся з Java Development Tools в Eclipse, оскільки ви будете застосовувати деякі з найчастіше використовуваних інструментів, а саме, Debugger, знову Scrapbook та Inspector. Ці інструменти досить корисні при розробці Java-коду. У цій роботі ви по кроках пройдете через виконання конструктора Board і постараєтеся визначити та виправити неправильне поводження.

Виконання роботи:

Крок 1: Створення класу Rule

У проекті LifeGame створіть новий клас Rule (в тім же пакеті, що й інші класи гри), що має закрите поле name типу String. Згенеруйте методи-акцесори для поля. Клас буде використаний пізніше в наших роботах для подання різних правил гри. Поки ви створюєте основу для цього класу.

Додайте конструктор класу, що приймає параметр типу String і привласнює його полю name. Конструктор повинен виглядати так:

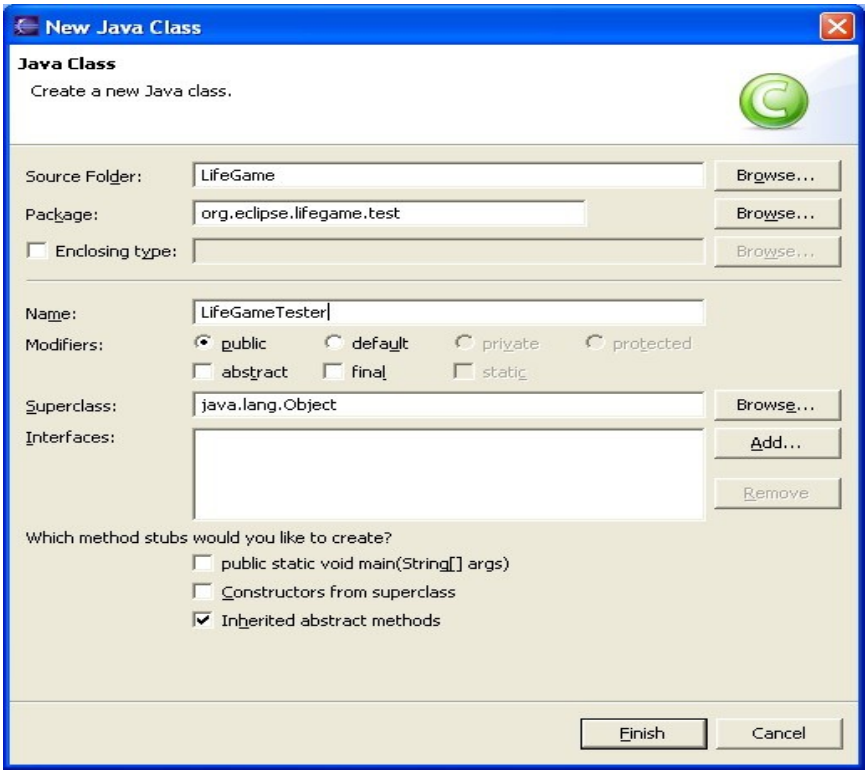
```
public Rule(String name){
    name = name;
}
```

Крок 2: Створення екземпляра Rule

Створіть в Scrapbook новий екземпляр правила, використовуючи створений конструктор, і проінспектуйте його. Яке значення поля name?

Очевидно, що поле name не встановлене в конструкторі з кількох причин. Вам знадобиться використати Debugger, щоб побачити, що відбувається, коли ім'я передається в конструктор.

У проекті LifeGame створіть новий пакет org.eclipse.lifegame.test та додайте в пакет клас LifeTester. При створенні класу переконаєтеся, що ви вибрали створення методу main().

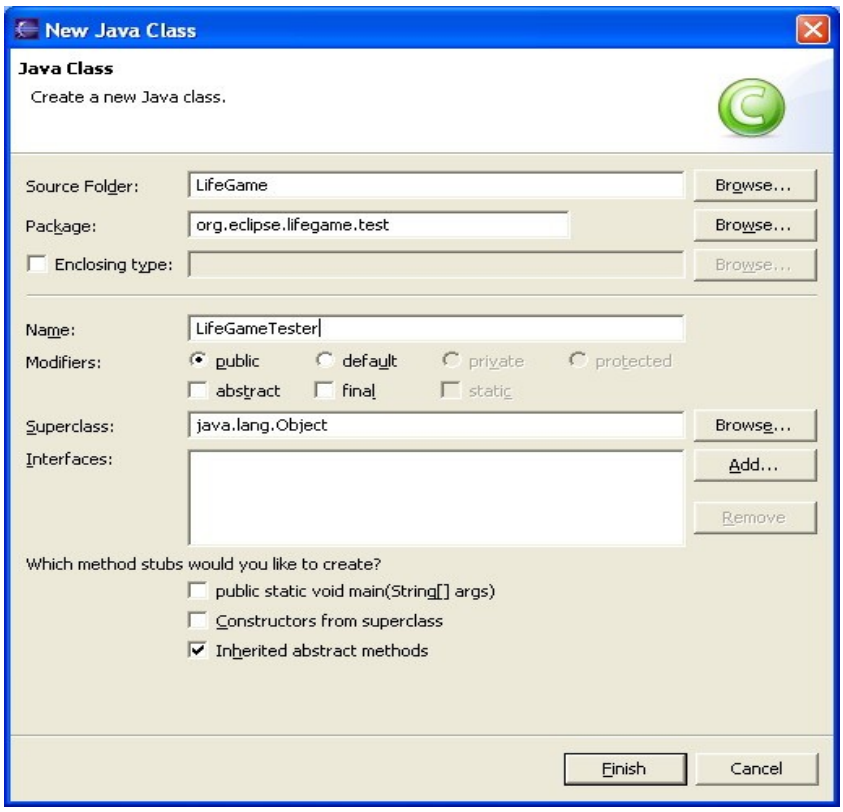


Ви будете використати цей клас для виклику Debugger й інспектування деякого коду.

У методі main() напишіть код з Scrapbook. Код повинен виглядати так:

```
public static void main(String[] args) {
    Rule rule = new Rule("Survival");
    System.out.println(rule.getName());
}
```

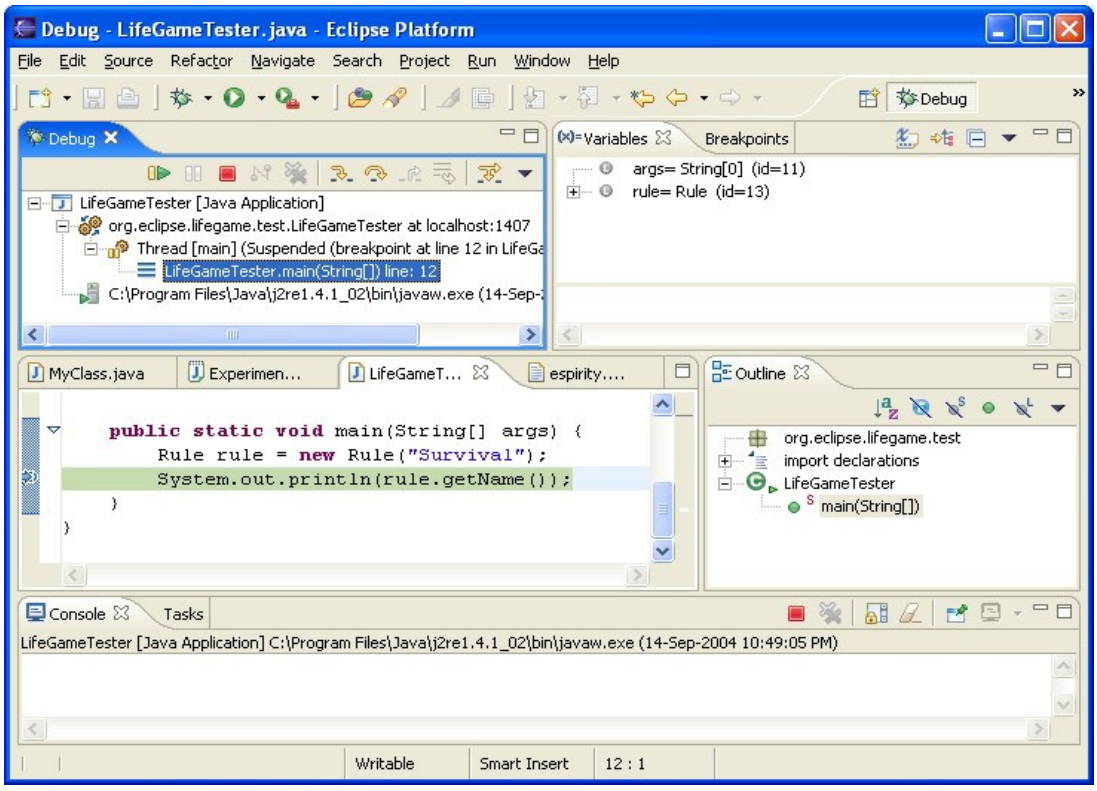
Помітьте, що клас Rule перебуває в іншому пакеті, і що вам потрібно додати оператор імпорту. Запустите клас, вибравши Run -> Run As -> Java Application з меню, що випадає. В Console виведеться null.



Шаг 3: Налагодження конструктора

Встановіть точку останова в методі main і знову запустіть LifeGameTester. Чи з'явиться Debugger? Щоб викликати Debugger, вам потрібно вибрати опцію Debug замість опції Run для класу LifeGameTester.

Debugger з'являється й зупиняє виконання в методі main, де була встановлена точка останова.



Пройдіть конструктор по кроках й обчисліть переданий параметр, а також і поле name. Чи зможете ви побачити, у чому проблема в коді?

Конструктор використає вираз name = name; де name - це параметр, переданий у конструктор, і в той же час - поле класу. Коли код обчислюється, передане значення імені призначається переданій змінній name. Це не дає ніякого ефекту й Eclipse розпізнає це й відображає попередження слідом за цим виразом.

Правильний вираз повинний бути this.name = name; де this.name означає поле, а name означає переданий параметр.

Якщо ви обчислили код із точкою останова в Scrapbook, відкриється Debugger. Якщо код має попередження, Debugger не відкриється.

Висновки:

У цій роботі ви познайомилися з використанням Eclipse Debugger.

ЛАБОРАТОРНА РОБОТА 6: Оператори керування

Ціль роботи:

У цій роботі ви будете використовувати оператори керування для реалізації деякого поводження класів гри. Зокрема, ви реалізуєте шаблон Singleton для класу Game та метод toString() класу Board.

Виконання роботи:

Крок 1: Реалізація шаблона Singleton для класу Game

Скільки екземплярів класу Game знадобиться вам одночасно в одному додатку? Зрозуміло, що, швидше за все, вам знадобиться тільки один екземпляр класу Game в одному додатку. Цей екземпляр буде представляти саму гру. Якщо в одній JVM може бути створено не більше одного екземпляра класу, це називається шаблоном Singleton (одинак). На цьому кроці ви реалізуєте цей шаблон для класу Game.

Визначте закрите статичне поле з ім'ям theInstance у класі Game. Це поле буде зберігати екземпляр класу, так що тип поля повинен бути Game. Додайте закритий статичний метод getInstance() у клас Game, що буде повертати екземпляр класу, збережений у поле theInstance. У методі перевірте, насамперед, чи не має поле значення null, якщо так, то створіть новий екземпляр класу Game і привласніть його цьому полю. Така реалізація називається ледачою ініціалізацією.

Змініть конструктор класу Game на закритий, щоб тільки сам клас міг його використати. Це не дасть можливості комусь ще створювати екземпляри класу.

Використайте створений зараз шаблон в Scrapbook для інспектування коду. Проінспектуйте вираз Game.getInstance();

Крок 2: Метод toString() класу Board

У попередніх роботах ви визначили метод toString() класу Board таким, що повертає просту String. Метод toString() є текстовим поданням об'єкта. Коли об'єкт друкується на стандартну консоль Java, використовується цей метод. Дошка містить клітки, які є нулями або одиницями. Коли дошка друкується на консолі, для кліток, що містять нулі, повинні друкуватися пробіли, а для кліток, що містять одиниці повинні друкуватися символи X.

Реалізуйте метод toString(), щоб він повертав рядок на підставі цієї специфікації. Це дозволить нам побачити на Console, як виглядає наша дошка.

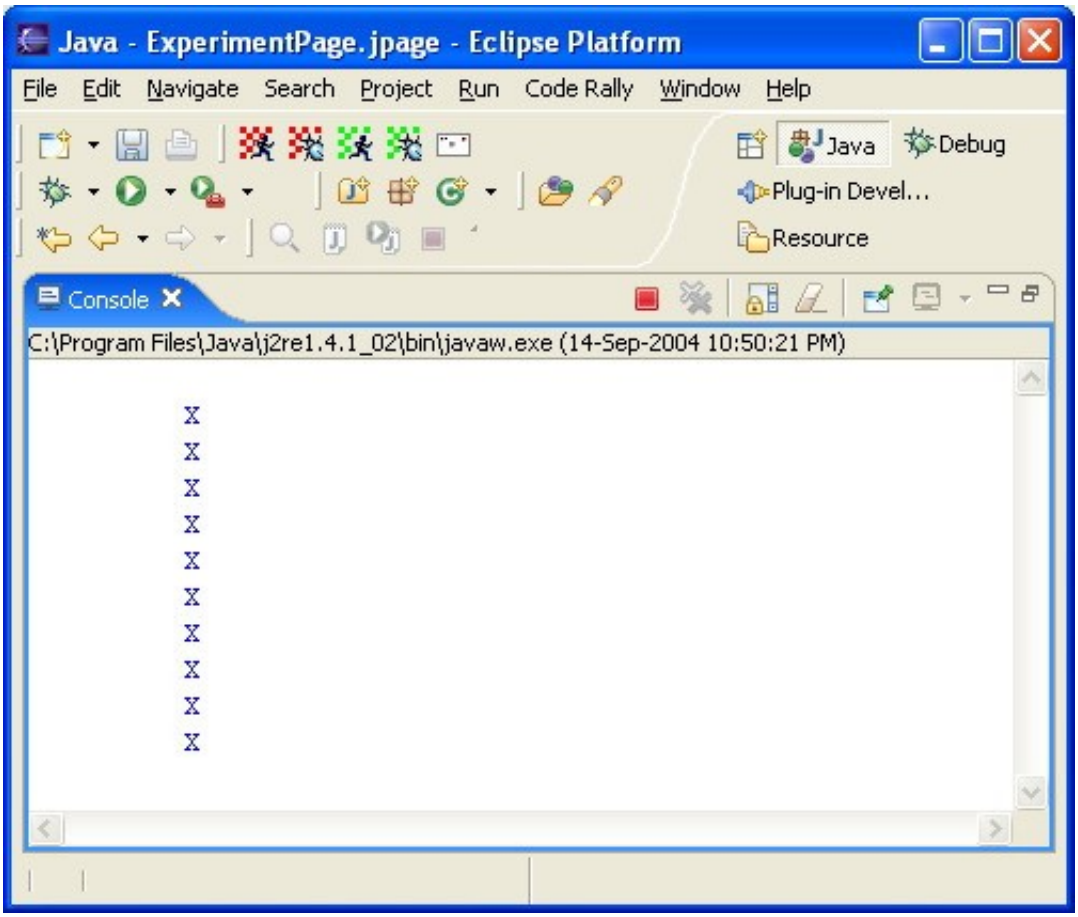
Щоб реалізувати правильне поводження, вам знадобиться перебрати масив масивів і перевірити кожен осередок. Використайте клас StringBuffer для додавання символів у кінець рядка.

Використайте Scrapbook, щоб перевірити метод toString() шляхом створення й роздруківки екземпляра класу на Console. Наприклад, код Scrapbook може виглядати в такий спосіб:

```
int [][] boardCells = {
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
```

```
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}};
Board board = new Board(boardCells);
System.out.println(board);
```

Цей код буде виведений на Console так:

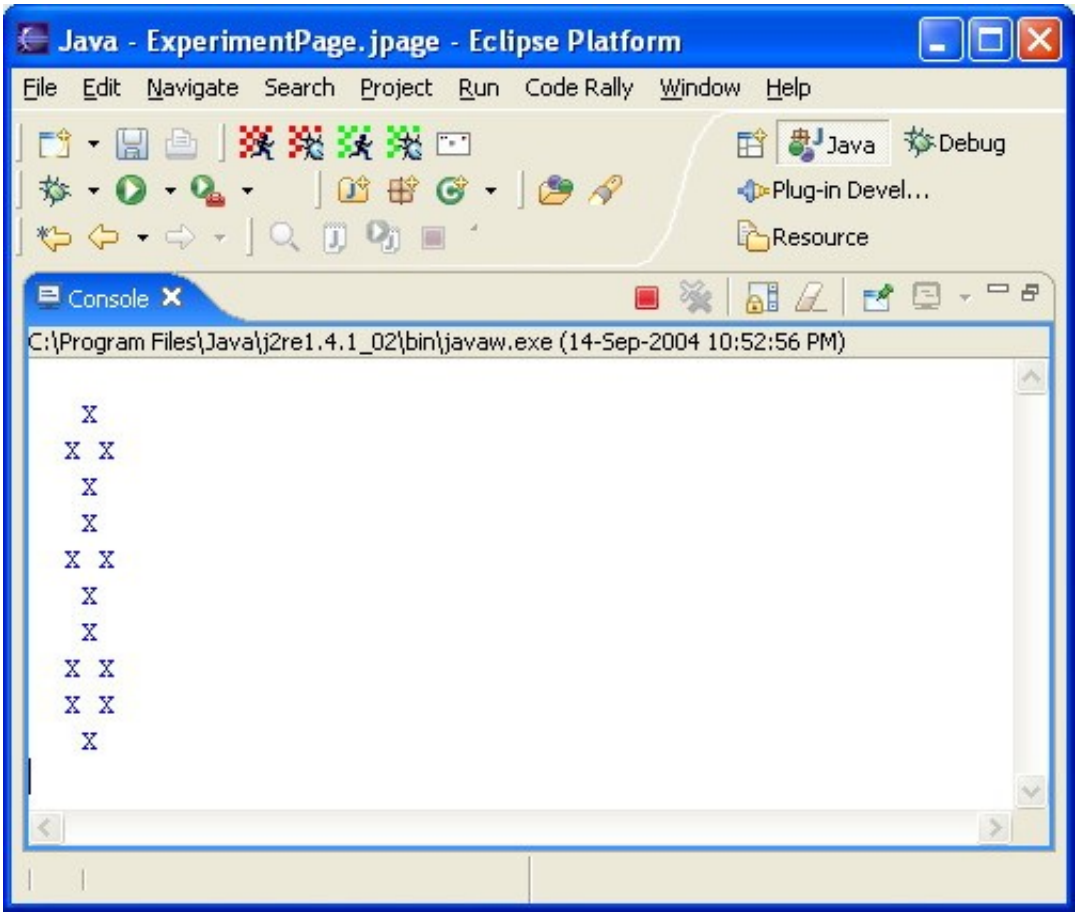


Або код може бути таким:

```
int [][] boardCells = {
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
```

```
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0}};
    Board board = new Board(boardCells);
    System.out.println(board);
```

І він буде виведений так:



Висновки:

У цій роботі ви використали оператори керування для реалізації шаблона Singleton для класу Game і методу toString() класу Board.

ЛАБОРАТОРНА РОБОТА 7: Спадкування

Ціль роботи:

У цій роботі ви будете використовувати спадкування для поліпшення проекту, підвищення гнучкості й можливостей супроводу системи.

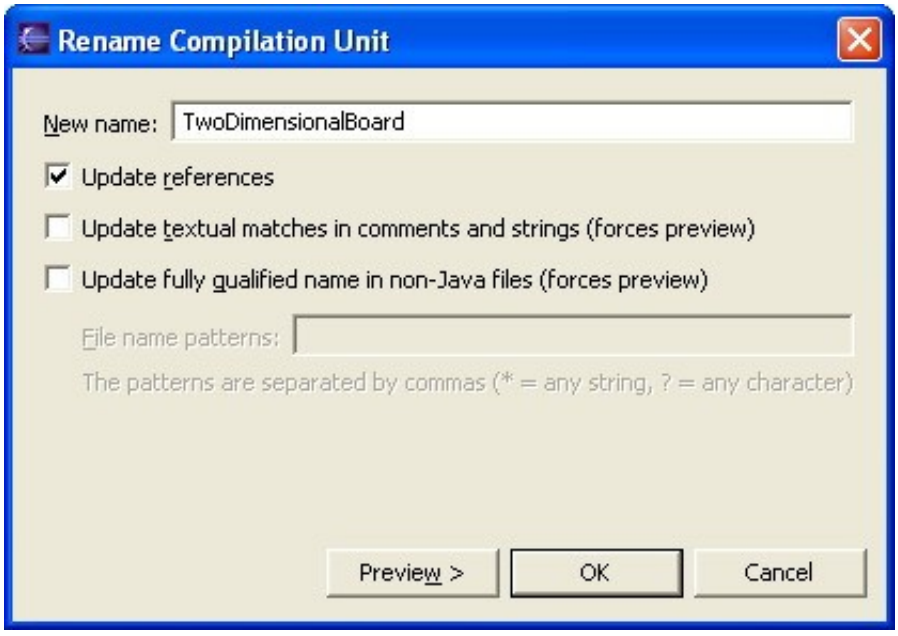
Спадкування допомагає при рефакторинзі коду, а також при правильному поданні об'єктів (тобто воно робить об'єкт більше близьким до реального життя). Досить часто деякі загальні властивості схожих класів абстрагуються в абстрактному суперкласі й повторно використовуються в підкласах. Абстрактний клас - це клас, що не може мати екземплярів. Інтерфейси використовуються в мові Java для більшого абстрагування, оскільки вони визначають тільки об'єктний протокол. Інтерфейси також використовуються для крос-ієрархічного поліморфізму (перехресного спадкування). У цій роботі ви будете використовувати й абстрактні класи, і інтерфейси. У цих роботах ви зосередитесь на 2-мірному моделюванні клітинної структури в Грі життя. Конкретно в цій роботі ви будете робити систему гнучкою для обробки різних версій ігор моделювання кліток.

Виконання роботи:

Крок 1: Перепроектування класу Board

Клас Board, що ви визначили в попередніх роботах, містить 2-мірну клітинну структуру, це означає, що він у реальності є 2-мірною дошкою. Що буде, якщо ви захочете ввести 3-мірну дошку? Чи зміниться клас Board? А як щодо 1-мірної? Щораз, коли ви захочете ввести нову гру (задачу), вам знадобитися міняти клас. Що загального повинні мати всі дошки, і які розходження між ними? Розходження між дошками складається в їхніх клітках, оскільки 1-мірна дошка є 1-мірним масивом, 2-мірна дошка є 2-мірним масивом і т.д. Також розрізняється реалізація протоколу evolve(), оскільки на різних дошках гра розвивається по-різному.

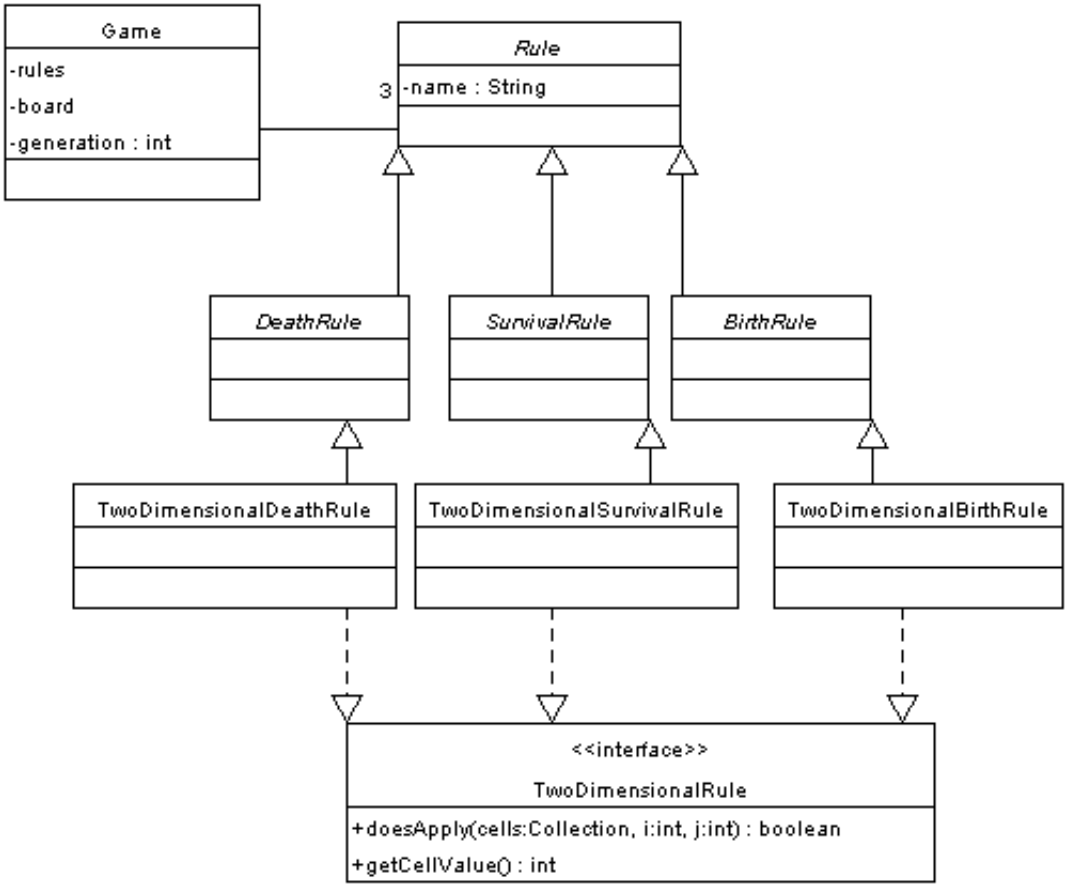
Змініть ім'я класу Board на TwoDimensionalBoard, оскільки це те, що насправді реалізується в класі Board. Щоб змінити ім'я класу, виберіть клас і виберіть опцію Refactor -> Rename з контекстного меню. Eclipse відкриє діалог для зміни імені:



Створіть абстрактний клас Board з абстрактним протоколом розвитку - методами evolve(Game, int) і evolve(Game). Це змусить всі підкласи класу Board реалізувати в собі ці методи. Змініть суперклас класу TwoDimensionalBoard на Board (:**extends** Board:).

Коли ви змінюєте ім'я класу, змінюються всі посилання, це означає, що поле board у класі Game тепер буде типу TwoDimensionalBoard. Ви хочете зберегти це поле родовим, так що змініть його тип на Board. Змініть його методи-акцесори, щоб вони повертали й приймали також тип Board.

Крок 2: Перепроектування класу Rule - шаблона Strategy



Клас Rule, що ви визначили до цього, має поле name. Воно може бути індикатором типу правила. Правило повинне мати метод doesApply(), що повертає значення boolean, що залежить від того, чи застосовне правило для поточної клітки чи ні. Протокол повинен спочатку перевіряти ім'я правила, а потім на його підставі робити різні обчислення. При такому підході можливі деякі потенційні проблеми в супроводі при введенні нового правила, оскільки метод повинен буде змінитися. Кращим підходом є наявність специфічного класу Rule для кожного правила в грі. Кожен клас повинен буде реалізовувати власний протокол doesApply(). Якщо ми зробимо так, то при введенні нового правила в гру, буде створюватися новий специфічний клас зі своєю реалізацією методу doesApply(). Ви можете абстрагувати поводження правил в абстрактному класі Rule. Змініть визначення класу Rule на abstract.

Створіть абстрактні класи DeathRule, SurvivalRule й BirthRule як підкласи класу Rule, створіть TwoDimensionalDeathRule, TwoDimensionalSurvivalRule й TwoDimensionalBirthRule - підкласи тільки що створених абстрактних класів.

Метод doesApply() досить специфічний для різних розмірностей гри. Інакше кажучи, метод 2-мірного правила приймає як параметр 2-мірний масив кліток. Як ми можемо змусити всі 2-мірні правила мати однаковий протокол й у той же час успадковувати різним

суперкласам? Відповіддю є перехресне спадкування й інтерфейси.

Створіть новий інтерфейс TwoDimensionalRule і визначте методи:

```
public int getCellValue();
public boolean doesApply(int[][] cells, int i, int j);
```

Реалізуйте метод getCellValue() в усіх 2-мірних класах. Метод повинен просто повертати значення, що повинне бути 1 для правил народження та виживання й 0 для правила смерті.

Змініть всі класи TwoDimensionalXYZRule, щоб реалізувати створений інтерфейс.

Додайте поле rules у клас Game. Поле буде масивом правил, і в ньому повинне бути точно 3 правила, записані в масив.

Шаблон, що ви зараз реалізували і який показаний на картинці вище, відомий як шаблон Strategy.

Крок 3: Імпорт класів Rule

Оскільки в класах правил багато кодування, просто імпортуйте всі класи правил (3), які забезпечені для вас. Ці класи реалізують методи на основі простих правил, заданих у вимогах для цієї гри.

Крок 4: Перепроектування класу Game

Зробіть раніше визначений клас Game абстрактним. Ви помітите, що Eclipse видає вам помилку, оскільки не можуть створюватися екземпляри цього класу, а ми створили екземпляр у реалізації "одинака". Створіть клас TwoDimensionalGame, що успадковує клас Game.

Конструктор за замовчуванням у класі Game повинен ініціалізувати правила, і він повинен виглядати в такий спосіб:

```
public Game(){
    this.rules = new Rule[3];
}
```

Ініціалізуйте масив rules у конструкторі класу Game, щоб створити екземпляри відповідних класів 2-мірних правил, визначених на попередньому кроці. Конструктор може виглядати у такий спосіб:

```
private TwoDimensionalGame(){
setBoard(new TwoDimensionalBoard());
getRules()[0] = new TwoDimensionalBirthRule();
getRules()[1] = new TwoDimensionalSurvivalRule();
getRules()[2] = new TwoDimensionalDeathRule();
}
```

Вам також знадобиться перенести реалізацію "одинака" із класу Game у його підкласи.

Крок 5: Реалізація протоколу Board

Реалізуйте методи розвитку в класі TwoDimensionalBoard. Метод evolve(Game, int) просто виконує цикл до переданого йому індексу (int), у кожній ітерації викликає метод

evolve(Game) і друкує себе на стандартній консолі. Код повинен виглядати в такий спосіб:

```
public void evolve(Game aGame, int index){
if (index == 0) return;
for (int i = 1; i < = index; i++){
      evolve(aGame);
      System.out.println(this);
      }
}
```

Метод evolve(Game) перебирає клітки дошки й для кожної клітки перевіряє, чи застосовне до неї якесь із правил гри. Якщо правило застосовується, то поточне значення клітки встановлюється в значення правила. Якщо ніяке правило до клітки не застосоване, то просто виводиться на консоль повідомлення про те, що до поточної клітки не застосоване ніяке правило до клітки. Код повинен виглядати в такий спосіб:

```
public void evolve(Game aGame){
 boolean doesAnyApply = false;
 TwoDimensionalRule rule = null;
 for (int i = 0; i < cells.length; i++){
  for (int j = 0; j < cells[0].length; j++){
  doesAnyApply = false;
   for (int k = 0; k < aGame.getRules().length; k++){
    rule = (TwoDimensionalRule)aGame.getRules()[k];
    if (rule.doesApply(getCells(), i, j)) {
     getCells()[i][j] = rule.getCellValue();
     doesAnyApply = true;
     break;
    }
   }
   if (!(doesAnyApply))
    System.out.println("No rules apply for cell[" + i + "][" + j + "]");
  }
 }
}
```

Крок 6: Реалізація методу run у класі Game

Тепер ви маєте більшість частин, необхідних для виконання гри. Додайте метод run(), який повертає значення типу void у клас Game. Метод повинен просто викликати метод розвитку ігрової дошки й передавати йому покоління гри й саму гру як параметри.

Метод може виглядати так:

```
public void run(){
    getBoard().evolve(this, getGeneration());
}
```

Крок 7: Тестування гри

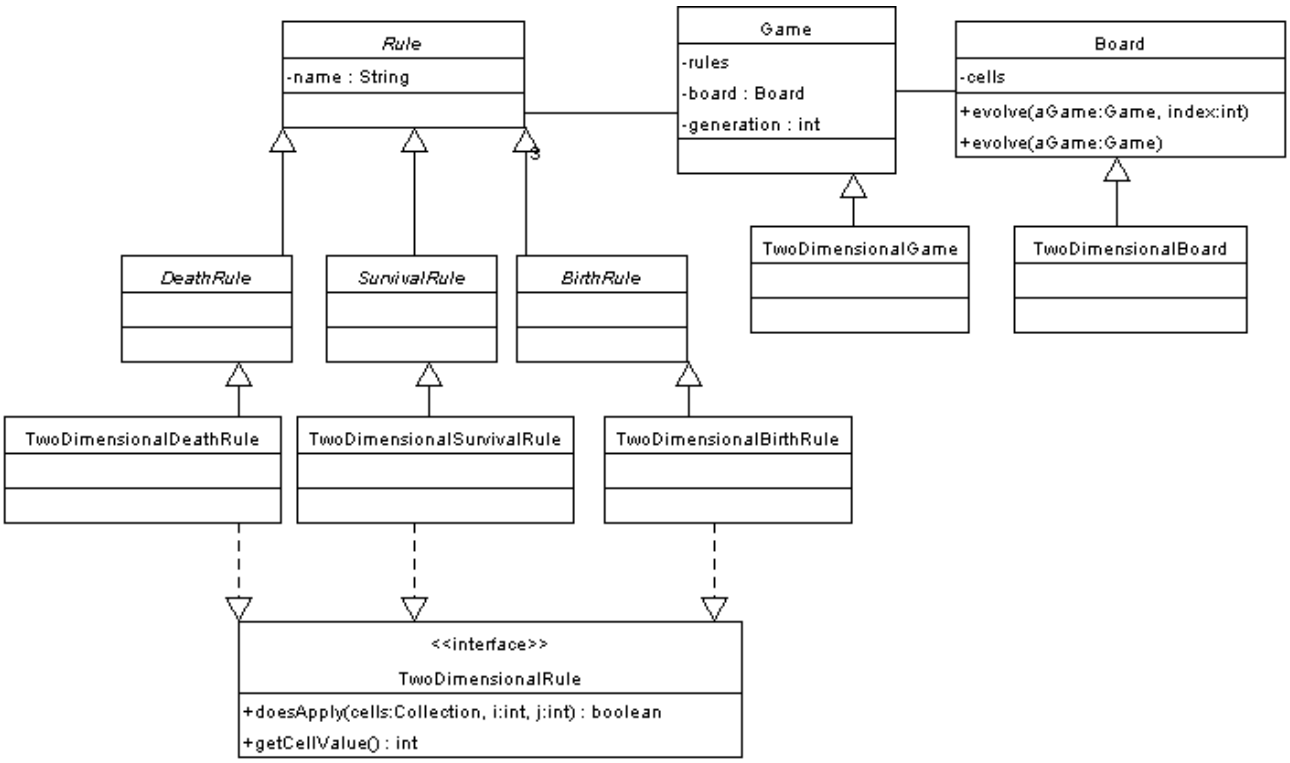
Змініть клас LifeGameTester, щоб створювати новий екземпляр TwoDimensionalBoard, привласнювати деякі ініціалізовані boardCells дошці, а потім одержувати екземпляр TwoDimensionalGame, призначте йому раніше створений board, призначте екземпляру 10 як generation й, нарешті, виконайте run для екземпляра.

Метод main може виглядати так:

```
public static void main(String[] args) {
    int[][] boardCells = {
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0 , 0, 0, 0}};
    TwoDimensionalBoard board = new TwoDimensionalBoard(boardCells);
    TwoDimensionalGame game = TwoDimensionalGame.getInstance();
    game.setBoard(board);
    game.setGeneration(10);
    game.run();
}
```

Запустіть клас тестера. Простежте поводження при багаторазовому запуску з різними значеннями кліток.

Після рефакторинга й перепроектування класів ви можете мати діаграму класів, як на картинці нижче:



Висновки:

У цій роботі ви використали спадкування, абстрактні класи, інтерфейси й інструменти рефакторинга Eclipse.

ЛАБОРАТОРНА РОБОТА 8: Колекції

Ціль роботи:

В цій роботі ви будете використовувати колекції для реалізації історії розвитку дошки в ході гри. Коли дошка розвивається, клітинки міняють свої значення, і дошка переходить у зовсім інший стан. Наприкінці гри дошка має стан нового покоління, а всі попередні стани загублені. Ви реалізуєте історію розвитку дошки в класі Game.

Виконання роботи:

Крок 1: Додавання поля boardHistory у клас Game

Визначте нове поле boardHistory типу ArrayList у класі Game. Згенеруйте для поля методи-акцесори. В конструкторі Game ініціалізуйте поле новим екземпляром ArrayList.

Крок 2: Додавання протоколу маніпулювання історією

Визначте новий метод addToHistory(Board) у класі Game. Метод повинен приймати як параметр екземпляр класу Board і просто додавати його до колекції boardHistory гри.

Визначте інший метод у класі Game, clearHistory(), що буде видаляти всі елементи з колекції boardHistory.

Крок 3: Використання протоколу маніпулювання історією

Змініть метод run() у класі Game, щоб він очищав історію дошки перед запитом до дошки на розвиток.

Після того, як дошка розвилася у свій новий стан, вона повинна бути додана до історії дошки перш, ніж вона розів'ється знову. Модифікуйте метод evolve(Game, int) класу Board, щоб додавати дошку до історії гри після кожного розвитку. Ви можете використати вираз aGame.addToHistory(**this**); у циклі **for**.

Крок 4: Реалізація роздруківки історії

У попередній роботі ви додали роздруківку дошки на стандартній консолі щораз, коли дошка розвивається, у метод evolve(Game,int). Це було зроблено з метою тестування, щоб ви бачили, як клітки дошки змінюють значення. У дійсності відповідальність за роздруківку дошки буде в трохи іншого об'єкта. Щоб підтримати це, ви повинні видалити оператор друку з методу evolve(Game,int). Після цього додайте в клас Game метод printHistory(), що буде перебирати колекцію історії й у кожній ітерації друкувати дошку на консолі. Майте на увазі, що кожен об'єкт у колекції повинен бути наведений до його типу. boardHistory буде містити об'єкти типу Board, так що в кожній ітерації по історії вам знадобиться приводити об'єкти до цього типу.

Метод може виглядати так:

```
public void printHistory(){
    Iterator iterator = getBoardHistory().iterator();
    Board board = null;
    while (iterator.hasNext()){
```

```
            board = (Board)iterator.next();
            System.out.println(board);
        }
    }
```

Крок 5: Додавання до "одинака"

Коли ви використаєте шаблон "одинака", ви використаєте єдиний екземпляр. У динамічному середовищі розробки, коли ви регулярно змінюєте стан і поводження об'єкта, це може привести до небажаного поводження. Вам потрібно очищати "одинака" щораз, коли ви робите зміни, щоб бути впевненими, що він веде себе правильно.

Додайте відкритий статичний метод у клас TwoDimensionalGame, що буде просто скидати екземпляр в null:

```
    public static void clearInstance(){
        theInstance = null;
    }
```

Ви можете використати цей метод щораз перед повторним використанням "одинака".

Крок 6: Тестування коду

Змініть клас тестера для використання реалізованого протоколу друку історії після виконання гри. Запустіть тестер знову.

Висновки:

У цій роботі ви використали колекції Java для відстеження й збереження історії поводження дошки в грі.

ЛАБОРАТОРНА РОБОТА 9: Обробка виключень

Ціль роботи:

Ціль цієї роботи складається в роботі з виключеннями. В цій роботі ви вивчите, як викидати та виловлювати виключення.

Виконання роботи:

Крок 1: Створення виключення в Грі життя

У Грі життя можливо створити ситуацію, коли до даної клітки не може бути застосоване жодне правило. У цьому випадку клітка ніколи не народиться і не вмре. Отже, клітка залишиться незмінною протягом всієї грі. Це виняткова ситуація, тому що, до кожної клітки завжди повинне бути застосоване, як мінімум, одне правило в кожнім поколінні клітки.

Створіть новий клас виключення LifeGameException у пакеті org.eclipse.lifegame.support. Клас виключення повинен успадковувати клас Exception. Забезпечте для класу два конструктори. Перший - конструктор за замовчуванням. Другий - конструктор, що має параметр типу String, що є повідомленням про помилку для виключення, він перекриває конструктор суперкласу. У цьому конструкторі ви можете просто викликати конструктор суперкласу, що встановлює передане повідомлення про помилку в поле повідомлення про помилку.

Крок 2: Викидання виключення

Змініть ваш клас Board, щоб він викидав виключення. Вам знадобиться фізично викидати виключення в методі вашого класу TwoDimensionalBoard, де ви визначаєте, що немає застосованого правила. Ви також повинні визначити, що відповідний метод викидає виключення. Це також повинне бути зроблене для методів абстрактного класу Board.

Крок 3: Виловлювання виключення

В тім місці, де ви запускаєте Game, ви тепер повинні виловлювати нове виключення. Додайте цю функціональність у вашу реалізацію. Коли викинуте виключення, ви повинні зупинити розвиток гри й видати повідомлення про помилку. Ви можете створити діалог помилки, використовуючи наступний код:

```
try {
    ...

} catch (LifeGameException exception) {
    Display display = new Display ();
    Shell shell = new Shell (display);
    MessageDialog.openError(shell, "Error with Game", exception.getMessage());
}
```

Щоб використати MessageDialog, вам знадобиться включити у ваш клас наступні оператори імпорту:

import org.eclipse.swt.widgets.Display;

import org.eclipse.swt.widgets.Shell;

І включити бібліотеки swt.jar й jface.jar у ваш маршрут побудови проекту (projects build path).

Крок 4: Перевірка виключення

Перевірте ваше виключення, створивши у вашому додатку ситуацію, що викидає виключення.

Висновки:

У цій роботі ви створили нове виключення, яке використовується для визначення виняткової ситуації у вашому додатку, і обробили його, коли ця ситуація себе проявляє.

ЛАБОРАТОРНА РОБОТА 10: Потоки

Ціль роботи:

Метою цієї роботи є робота з потоками. У цій роботі ви вивчите, як писати й читати дані в і з файлу, використовуючи потоки.

Виконання роботи:

Крок 1: Створення методів Game

Додайте два нових методи в клас Game з іменами:

public void saveCurrentBoardToFile(String filename);

public void loadBoardFromFile(String filename);

Ці методи повинні записувати ігрову дошку в заданий файл, а також читати дошку із заданого файлу. Інакше кажучи, метод запису повинен серіалізувати ігрову дошку у файл, а метод читання десеріалізувати об'єкт дошки з файлу та установлювати в нього ігрову дошку.

Крок 2: Створення абстрактних методів Board

Додайте два абстрактних методи в клас Board з іменами:

public abstract void saveCurrentBoardToFile(String filename);

public abstract void loadBoardFromFile(String filename);

Крок 3: Перевірка ваших методів

Ви можете перевірити вашу реалізацію шляхом обчислення наступного коду в Scrapbook (ви можете використати Scrapbook з ExperimentLab):

```
int[][] boardCells = {
        {0, 0, 0, 1, 0, 0, 1 , 0, 1, 0},
        {0, 0, 1, 0, 1, 0, 1 , 0, 0, 0},
        {1, 0, 0, 1, 0, 0, 1 , 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 1 , 0, 0, 0},
        {0, 0, 1, 0, 1, 0, 1 , 1 ,0, 0},
        {0, 0, 0, 1, 0, 0, 1 , 1, 1, 0},
        {0, 0, 0, 1, 0, 0, 1 , 1, 0, 1},
        {1, 0, 1, 0, 1, 0, 1 , 0, 1, 0},
        {0, 0, 1, 0, 1, 0, 1 , 0, 1, 1},
        {0, 0, 0, 1, 0, 0, 1 , 0, 0, 1}};
TwoDimensionalBoard board = new TwoDimensionalBoard(boardCells);
TwoDimensionalGame.clearInstance();
TwoDimensionalGame game = TwoDimensionalGame.getInstance();
```

```
game.setBoard(board);
game.saveCurrentBoardToFile("c:/game.txt");
int[][] newBoardCells = {
        {1, 1},
        {1, 1}};
board = new TwoDimensionalBoard(newBoardCells);
game.setBoard(board);
System.out.println(game.getBoard());
game.loadBoardFromFile("c:/game.txt");
System.out.println(game.getBoard());
```

У цьому коді ви спочатку створюєте дошку, а потім зберігаєте її у файлі. Потім ви створюєте нову, меншу дошку, привласнюєте її грі, чим, отже, видаляєте стару ігрову дошку. Після цього ви роздруковуєте нову дошку, завантажуєте стару дошку, привласнюєте її грі й роздруковуєте її, так що ви візуально перевірите, чи працюють завантаження й запис. Ви завжди можете перевірити, що було записано у файл, переглянувши його "вручну" за допомогою вашого улюбленого текстового процесора. При серіалізації об'єкта у файл записуються двійкові дані, але якщо ви записуєте інші дані (примітиви, текст), вони повинні бути читабельними.

Висновки:

У цій роботі ви використали потоки Java для збереження ігрової дошки у файлі й завантаження її з файлу.

ЛАБОРАТОРНА РОБОТА 11: Генерація Javadoc в Eclipse

Ціль роботи:

Ціль цієї роботи - познайомити вас із документуванням вашого Java-коду за допомогою коментарів Javadoc і тегів і з використанням інструмента Javadoc в Eclipse для вироблення Javadoc-виходу в стандартному доклеті.

Виконання роботи:

Крок 1: Повне коментування вашого класу Game

Візьміть ваш клас Game і повністю прокоментуйте його.

1. Забезпечте коментар перед класом, який буде також містити в собі автора, версію й теги see.
2. Прокоментуйте початок кожного методу. Переконаєтеся, що параметри й значення, що повертається, визначені за допомогою тегів param й return.
3. Якщо потрібно, додайте тег see і відповідний клас у коментарі вашого методу.

Корисно включати теги HTML у ваш текст javadoc. Наприклад, наступне є коментарем з тегами (<p>, <tt></tt>,), які ви теж можете вбудовувати у ваші документи. Спробуйте вивести це й подивитеся, що вони будуть робити.

```
 * <p>Many different types of exceptions can happen. For example
 * throw <tt>LifeGameException</tt> might be thrown.
 * It is wrong to write a program that depended on an
 * exception for its correctness:  <i>the fail-fast behavior
 * should not be used</i>
```

Крок 2: Експорт Javadoc

Експортуйте ваш пакет Гри життя як Javadoc. Виберіть тільки ваш пакет для створення Javadoc, у Майстру генерації Javadoc. Не забудьте встановити опції й цільовий каталог. Якщо все пройде добре, ви побачите щось схоже на наступне у вікні консолі. Якщо ви побачите коментар, пофарбований ЧЕРВОНИМ, у вас є проблеми. Прочитайте коментарі й виправте їх.

```
Loading source files for package org.eclipse.lifegame.test...
Loading source files for package org.eclipse.lifegame.support...
Loading source files for package org.eclipse.lifegame.domain...
Constructing Javadoc information...
Standard Doclet version 1.4.1

Generating C:\eclipse\workspace\LifeGame\doc\constant-values.html...
Building tree for all the packages and classes...
Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\class-
```

use\LifeGameTester.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\class-use\LifeGameException.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalSurvivalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalGame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalDeathRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalBoard.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\TwoDimensionalBirthRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\SurvivalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\Rule.html...

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\Game.java:48: warning - @return tag has no arguments.

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\Game.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\DeathRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\Board.html...

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\Game.java:34: warning - @return tag has no arguments.

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\class-use\BirthRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\package-use.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\package-use.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\package-use.html...

Building index for all the packages and classes...

Generating C:\eclipse\workspace\LifeGame\doc\overview-tree.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-1.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-2.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-3.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-4.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-5.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-6.html...

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\Game.java:90: warning - @return tag has no arguments.

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\TwoDimensionalBoard.java:22: warning - @return tag has no arguments.

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\Game.java:76: warning - @return tag has no arguments.

C:\eclipse\workspace\LifeGame\org\eclipse\lifegame\domain\Rule.java:19: warning - @return tag has no arguments.

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-7.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-8.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-9.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-10.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-11.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-12.html...

Generating C:\eclipse\workspace\LifeGame\doc\index-files\index-13.html...

Generating C:\eclipse\workspace\LifeGame\doc\deprecated-list.html...

Building index for all classes...

Generating C:\eclipse\workspace\LifeGame\doc\allclasses-frame.html...

Generating C:\eclipse\workspace\LifeGame\doc\allclasses-noframe.html...

Generating C:\eclipse\workspace\LifeGame\doc\index.html...

Generating C:\eclipse\workspace\LifeGame\doc\packages.html...

Generating C:\eclipse\workspace\LifeGame\doc\overview-summary.html...

Generating C:\eclipse\workspace\LifeGame\doc\overview-frame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\package-frame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\package-summary.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\package-tree.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\package-frame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\package-summary.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\package-tree.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\package-frame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\package-

summary.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\package-tree.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\test\LifeGameTester.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\support\LifeGameException.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\BirthRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\Board.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\DeathRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\Game.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\Rule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\SurvivalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalBirthRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalBoard.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalDeathRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalGame.html...

Generating C:\eclipse\workspace\LifeGame\doc\org\eclipse\lifegame\domain\TwoDimensionalSurvivalRule.html...

Generating C:\eclipse\workspace\LifeGame\doc\serialized-form.html...

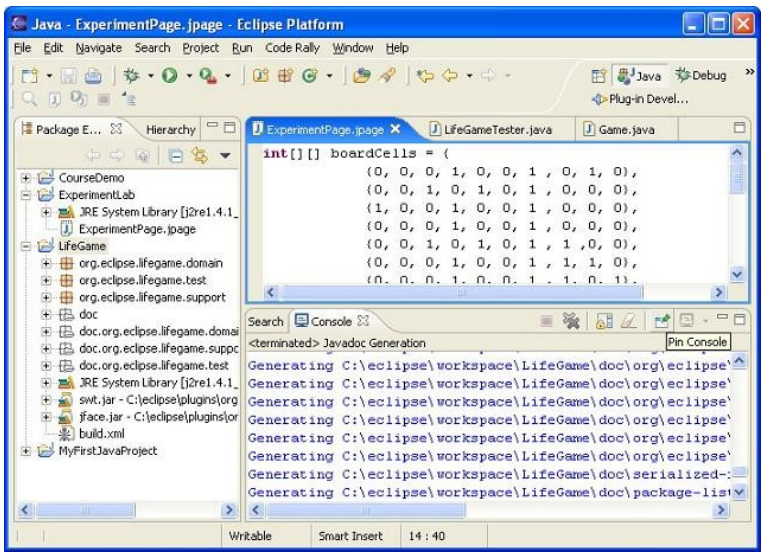
Generating C:\eclipse\workspace\LifeGame\doc\package-list...

Generating C:\eclipse\workspace\LifeGame\doc\help-doc.html...

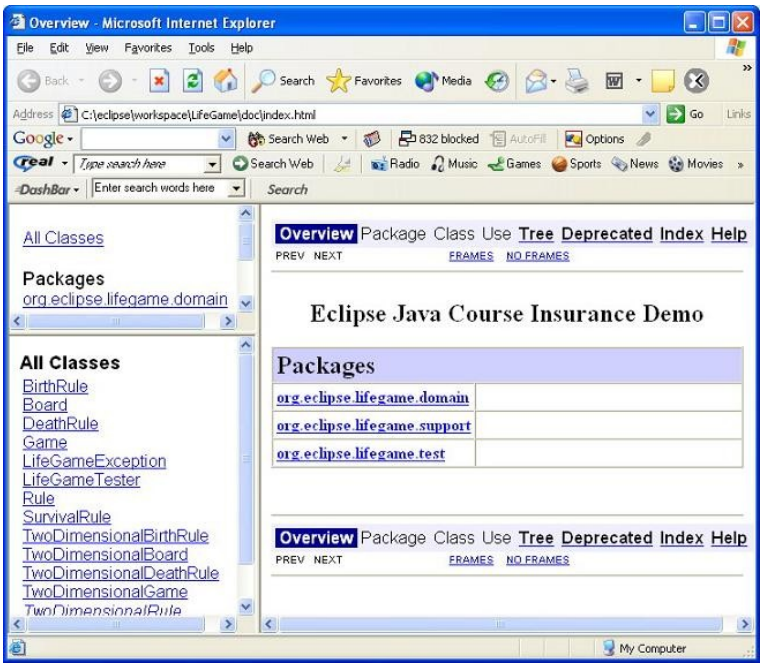
Generating C:\eclipse\workspace\LifeGame\doc\stylesheet.css...

6 warnings

Подивіться тепер у цільовий каталог, що ви визначили для Javadoc. Ви повинні побачити результуючі файли HTML. Якщо ви виберете вихідний каталог за замовчуванням, він буде створений у корені проекту в робочому просторі й також буде показаний в Eclipse Workbench.



Подвійне кликання на файлі index.html покаже Javadoc у такий спосіб:



ЛАБОРАТОРНА РОБОТА 12: Використання Ant в Eclipse

Ціль роботи:

Ціль цієї роботи - познайомити вас із діями, які ви повинні виконати, щоб використати Ant в Eclipse. Ці дії містять у собі написання файлу build.xml, а потім запуск різних задач, які ви повинні включити у файл. Другою метою є представлення вам деяких головних задач, які забезпечує Ant.

Виконання роботи:

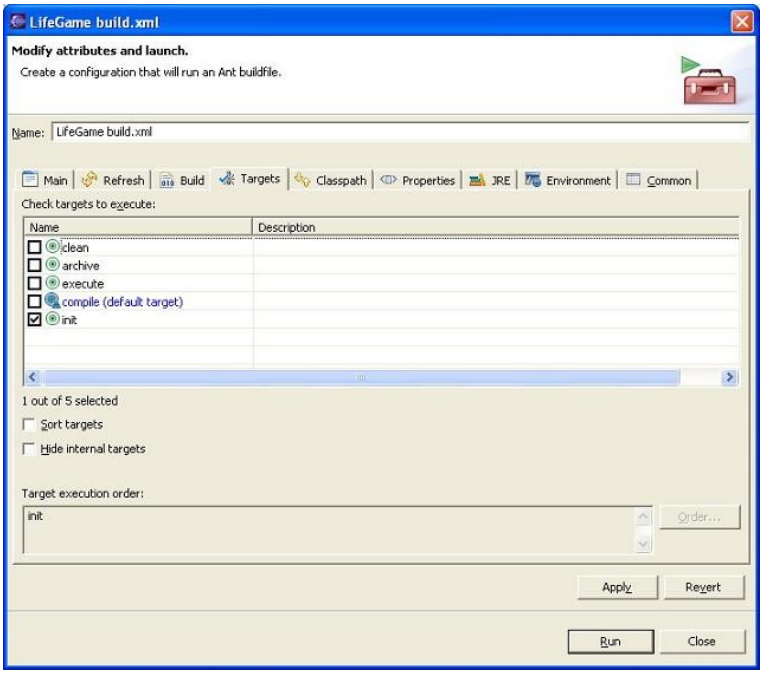
Крок 1: Створення файлу build.xml

У вашому проекті Гри життя створіть новий файл із ім'ям build.xml. Створіть файл побудови, що містить наступні задачі (tasks) і завдання (targets):

1. Завдання ініціалізації, що створює два каталоги: build/classes й bin.
2. Завдання компіляції, що залежить від завдання ініціалізації й компілює всі файли у вашому каталозі org/eclipse/lifegame/, поміщаючи файли класів у каталог build/classes.
3. Завдання архівації, що залежить від завдання компіляції й будує JAR-файл, bin/lifegame.jar, для класів у каталозі build/classes.
4. Завдання запуску, що залежить від завдання компіляції й запускає клас org.eclipse.lifegame.test.LifeGameTester з каталогу build/classes.
5. Завдання очищення, що залежить від завдання ініціалізації й видаляє каталоги build/classes й bin.

Крок 2: Виконання файлу build.xml тільки із завданням ініціалізації

Виберіть файл build.xml й, використовуючи праву клавішу мишки, виберіть Run Ant ... з меню, що випадає. Виберіть завдання ініціалізації й виберіть Run.



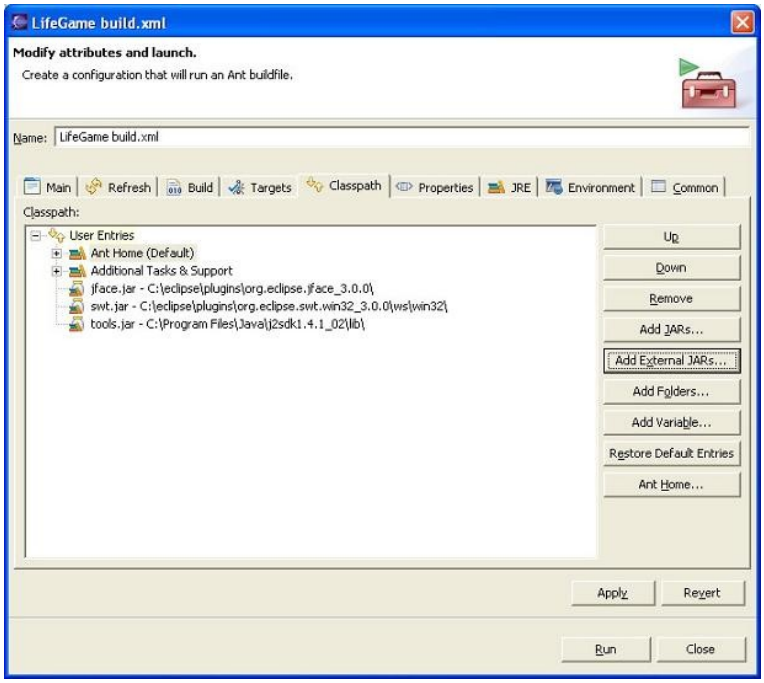
Ви повинні побачити щось схоже на наступне:

```
Buildfile: C:\eclipse\workspace\LifeGame\build.xml
init:
BUILD SUCCESSFUL
Total time: 300 milliseconds
```

Ви можете помітити, що каталоги build й bin не з'явилися у вашому середовищі Eclipse. Не турбуйтеся, ваш Package Explorer просто має потребу у відновленні. Виберіть ваш проект LifeGame, і в меню, що випадає під правою клавішею мишки, виберіть refresh. Тепер ви побачите ці каталоги. Не забудьте пробробляти цю операцію, коли ви щось додаєте або видаляєте в структурі каталогів.

Крок 3: Виконання завдання компіляції

Виберіть файл build.xml й, використовуючи праву клавішу мишки, виберіть Run Ant... з меню, що випадає. Виберіть завдання компіляції й виберіть Run. Не забудьте, що задача javac вимагає компілятора Java. Вам знадобиться спочатку додати файл tools.jar з вашої інсталяції JDK в classpath Ant. Також вам знадобиться додати файли залежностей jface.jar й swt.jar в classpath Ant, як показано нижче або ж ви можете вказати їх в аргументі classpath завдання файлу побудови.



Ви повинні побачити щось схоже на наступне:

```
Buildfile: C:\eclipse\workspace\LifeGame\build.xml
init:
compile:
    [javac] Compiling 14 source files to C:\eclipse\workspace\LifeGame\build\classes
BUILD SUCCESSFUL
Total time: 2 seconds
```

Перевірте, чи з'явилися нові файли в каталозі build/classes.

Крок 4: Виконання завдання архівації

Виберіть файл build.xml й, використовуючи праву клавішу мишки, виберіть Run Ant... з меню, що випадає. Виберіть завдання архівації й виберіть Run. Ви повинні побачити наступне:

```
Buildfile: C:\eclipse\workspace\LifeGame\build.xml
init:
compile:
   [javac] Compiling 14 source files to C:\eclipse\workspace\LifeGame\build\classes
archive:
     [jar] Building jar: C:\eclipse\workspace\LifeGame\bin\lifegame.jar
BUILD SUCCESSFUL
Total time: 2 seconds
```

Перевірте, чи з'явилися JAR-файли в каталозі bin, і чи містить він ваші файли класів.

Шаг 5: Виконання завдання запуску

Виберіть файл build.xml й, використовуючи праву клавішу мишки, виберіть Run Ant... з меню, що випадає. Виберіть завдання запуску й виберіть Run. Ви повинні побачити наступне:

```
Buildfile: C:\eclipse\workspace\LifeGame\build.xml
init:
compile:
   [javac] Compiling 14 source files to C:\eclipse\workspace\LifeGame\build\classes
execute:
     [java] X XXX
     [java] X X   X
     [java] X  X  XX
     [java] X XX X  X
     [java] X  X X X
     [java] X X X XX
     [java] XXX XX
     [java] XX
     [java] XX X
     [java] X X  XXX
     [java] X   XX  X
     [java] XX XX X XX
     [java] X X X X
     [java] XX X X XX
```

[java] X X XX
[java] XXX
[java] X
[java] X XX
[java] X X
[java] X XXX
[java] XXX   X
[java] X
[java] XX   X
[java] XX   X XX
[java] XX
[java] XX
[java] X
[java] X XX
[java] XX X
[java] X X X
[java] XXX   XX
[java] X   X X
[java] X X X X
[java] XX   X X
[java] XX
[java] XXXX
[java] XX XX
[java] X X X
[java] X X   X
[java] X   X X
[java] X   XX XX
[java] X XX X X
[java] XXXXXXX X
[java] XX
[java] XXX
[java] X
[java] X XX
[java] XX   X
[java] X X X
[java] X   X XX
[java] X X X X

[java] X X
[java] XXXXXXX X
[java] XX
[java] XXX
[java] X
[java] XX XX
[java] XX X XX
[java] X XX X X
[java] X X X XXX
[java] X X
[java] X X X XX
[java] XX XXX X
[java] X X XX
[java] XXX
[java] X
[java] X XX
[java] XX X XX
[java] X X X X
[java] X XX XXX
[java] X X X X
[java] X X X XXX
[java] XX X X
[java] XX
[java] XXX
[java] XXXXXX
[java] XX XX
[java] X XX XX
[java] X X X XXX
[java] X X X
[java] X X X XXX
[java] XX X X
[java] X X
[java] XXX
[java] X
[java] XXXXX X
[java] XX
[java] XXXXXX

```
    [java] X      XX
    [java] X    X
    [java] X X X XX
    [java] X   X
    [java] X
BUILD SUCCESSFUL
Total time: 2 seconds
```

Крок 6: Виконання завдання очищення

Виберіть файл build.xml й, використовуючи праву клавішу мишки, виберіть Run Ant... з меню, що випадає. Виберіть завдання очищення й виберіть Run. Ви повинні побачити наступне:

```
Buildfile: C:\eclipse\workspace\LifeGame\build.xml
init:
clean:
  [delete] Deleting directory C:\eclipse\workspace\LifeGame\build
  [delete] Deleting directory C:\eclipse\workspace\LifeGame\bin
BUILD SUCCESSFUL
Total time: 1 second
```

Обновіть ваше середовище Eclipse проекту й переконаєтеся, що каталоги вилучені.

Висновки:

У цій роботі ви використали можливості Ant в Eclipse, щоб компілювати, виконувати й упаковувати ваш додаток Гри життя.

ЛАБОРАТОРНА РОБОТА 13: Побудова графічного інтерфейсу користувача за допомогою SWT

Ціль роботи:

Ціль цієї роботи - познайомити вас із Standard Widget Toolket (SWT) в Eclipse. Точніше, ця робота допоможе вам вивчити, як використовувати один з найпопулярніших менеджерів розкладки: FormLayout, багато елементів (widgets) SWT, як будувати графічні користувальницькі інтерфейси, використовуючи елементи SWT і два шаблони, які обговорюються в слайдах: View Event Handler Pattern й Complete Update Pattern.

Виконання роботи:

Крок 1: Не забудьте

Для запуску незалежного додатка SWT ви повинні встановити аргументи VM у закладці Run Wizzard/argument у наступні (зверніть увагу на прямі слеши - /):

-Djava.library.path="C:/Program Files/eclipse/plugins/org.eclipse.swt.win32_3.0.0/os/win32/x86"

де C:/Program Files/eclipse заміняється вашим каталогом інсталяції Eclipse.

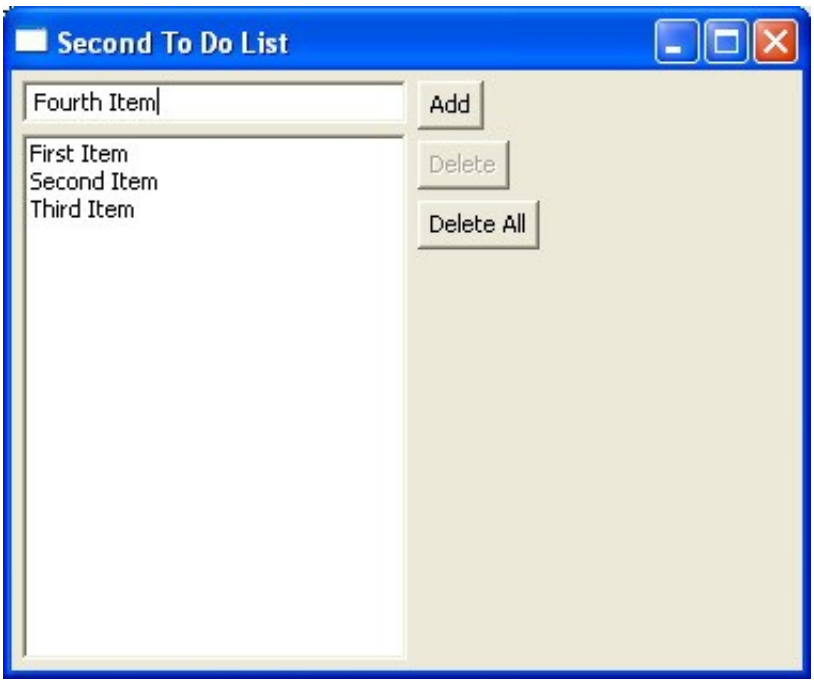
Крок 2: Розробка списку To Do

Розробіть представлення з ім'ям SecondToDoList, що аналогічно описаному в слайдах, за винятком наступних двох доповнень:

1. Елементи кнопок мають фіксований розмір
2. Додайте кнопку Delete All, що видаляє все зі списку To Do
3. Дозволяйте та забороняйте кнопку Delete All, коли це потрібно

Помітьте, що ви повинні додати зовнішній файл swt.jar у маршрут вашого проекту. Цей файл перебуває в каталозі підключення SWT каталогу інсталяції Eclipse.

Ви можете використати забезпечений для вас ToDoListView - перебуває в Lab13, - який містить код зі слайда, як стартову крапку для нового представлення.



Крок 3: Модель світлофора

Розробіть модель світлофора, що є візуальним поданням світлофора. Модель світлофора забезпечується, як клас TraficLight, у такий спосіб (**ЗАУВАЖЕННЯ**: Ви не повинні друкувати цей код, просто **скопіюйте наданий клас TraficLight у пакет**) - перебуває в Lab 13:

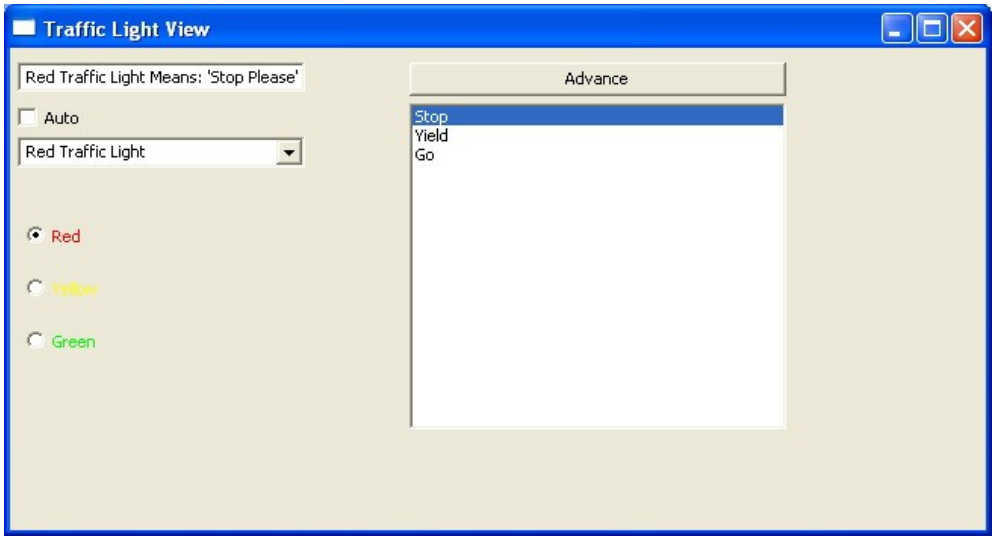
```
public class TrafficLight {


   int  currentState;


   // Конструктор, що створює червоний світлофор
   public TrafficLight() {
      currentState = 1;
   }


   // Переводення світлофора в наступний стан
   public int advance() {
      setState(getState() % 3 + 1);
      return currentState;
   }


   // Повернення стану світлофора (як числа від 1 до 3)
   public int getState() {
      return currentState;
```

```
    }

    // Установка стану світлофора (як числа від 1 до 3)
    // Якщо число виходить за межі, не робиться нічого
    public void setState(int newState) {
        if ((newState > 0) && (newState <4))
            currentState = newState;
    }

    // Повернення строкового представлення світлофора
    public  String toString() {
        switch (getState()) {
            case  1: return  "Red Traffic Light";
            case  2: return  "Yellow Traffic Light";
            case  3: return  "Green Traffic Light";
        };
        return  "Broken Traffic Light";
    }

    public boolean  isRed() {
        return  getState() == 1;
    }

    public boolean  isYellow() {
        return  getState() == 2;
    }

    public boolean  isGreen() {
         return  getState() == 3;
    }
}
```

Ваше подання буде містити наступні елементи:

1. Текст (Text)
2. Прапорець (CheckButton)
3. Список, що випадає (Combo)
4. Три радіокнопки (Radio Button)
5. Кнопку (Push Button) Advance
6. Список (List)

Ви можете розмістити ваші елементи будь-яким чином, тільки переконаєтеся, що вони не перекривають один одного. Загальне розміщення може виглядати так, як показано на картинці нижче.



Метою є мати представлення, що відображає модель. Кожен елемент може бути використаний для установки моделі, за винятком елемента Text (показаний у верхньому лівому куті). Наприклад, натискання на кнопку Advance змінює стан семафора. Ви можете вибрати кольори в List або використати Radio Buttons, або використати Combo. Установка прапорця Auto змушує кольори циклічно мінятися щосекунди. Важливою частиною вашого рішення є те, що незалежно від того, який елемент встановлює стан семафора, всі інші елементи повинні бути змінені для відображення нового стану.

Для цієї роботи може бути корисним:

Коли ви створюєте різні типи кнопок, ви задаєте різні параметри для конструктора кнопки:

```
pushButton = new  Button(shell, SWT.PUSH);     //створює звичайну кнопку
radioButton = new  Button(shell, SWT.RADIO);   //створює радіокнопку
checkBox = new  Button(shell, SWT.CHECK);      //створює прапорець
```

Для створення Timer, що посилає повідомлення кожні *n* мілісекунд, ви використовуйте наступне:

```
new  javax.swing.Timer (n, new  TimerEventHandler());
```

Повідомлення actionPerformed посилає в TimerEventHandler. Це клас, що ви створюєте й використовуєте для виконання дій через певні періоди. Наприклад:

```
private class  TimerEventHandler implements  ActionListener {
    public void  actionPerformed(ActionEvent e) {
            //Ваш код оброблювача повинен бути тут
    }
}
```

Більшість елементів має властивості кольорів. Ви можете подивитися їх у конкретних класах, але звичайно можна робити наступне.

```
aButton.setForeground (new  Color(display, 255,0,0));
```

У цьому прикладі ми встановлюємо кольори переднього плану для кнопки, що є типом Display, обумовленим display. Кольори, що ми вибрали - RED. Аргументом конструктора Color є значення RGB. У нашому випадку ми встановлюємо значення RED у максимум - 255. Ви можете встановить значення GREEN й BLUE самі. Підкажемо, що YELLOW - це тільки RED й GREEN.

Висновки:

У цій роботі ви використали SWT в Eclipse, щоб побудувати додаток із графічним інтерфейсом користувача.