Тема 6. Створення GUI. Бібліотека Swing

Лекція 8

Можливості та особливості різних бібліотек GUI

Раніше ми створювали програми, які були пов'язані з консоллю і запускались з командного рядка. Такі програми називаються *консольними додатками*. Вони розробляються для виконання на серверах, там, де не потрібен інтерактивний зв'язок з користувачем. Програми іншого призначення тісно взаємодіють із користувачем, сприймають сигнали від клавіатури і миші, працюють в графічному середовищі – це *додатки з графічним інтерфейсом користувача* (GUI).

Кожний GUI-додаток повинен створити хоча б одне вікно, в якому буде відбуватися його робота, і зареєструвати його в графічній оболонці операційної системи, щоб вікно могло взаємодіяти з операційною системою і іншими вікнами: перекриватися, переміщатися, змінювати розміри, згортатися до панелі задач і т. п.

Існує багато графічних систем: MS Windows, X Window System, Mac OS X та ін. Кожна з них має свої правила побудови вікон і їх компонентів: меню, полів введення, кнопок, списків, смуг прокрутки. Ці правила складні і заплутані, а графічні API зазвичай містять сотні функцій. Для полегшення створення вікон і їх компонентів для різних мов програмування написані спеціальні бібліотеки класів: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows та безліч інших. Кожен клас такої бібліотеки описує відразу цілий графічний компонент, керований методами цього та інших класів.

В технології Java справа ускладняється тим, що додатки повинні працювати однаково в будь-якому графічному середовищі (або по меншій мірі у багатьох таких середовищах). Таким чином, потрібна бібліотека класів, незалежна від конкретної графічної системи.

У першій версії JDK задачу вирішили наступним чином: були розроблені інтерфейси, що містять методи роботи з графічними об'єктами. Програми Java можуть їх використовувати для створення вікон, розміщення і переміщення графічних об'єктів, зміни їх розмірів, організації взаємодії графічних об'єктів. З іншого боку, для роботи з екраном в конкретній ОС і конкретній графічній оболонці ці інтерфейси реалізовані по-різному, з використанням API цієї оболонки, а також за допомогою графічних бібліотек даної операційної системи. Такі інтерфейси, за якими скриті різні реалізації для кожної графічної системи, були названі ***peer-інтерфейсами***.

Бібліотека класів Java на основі peer-інтерфейсів отримала назву **AWT** (Abstract Window Toolkit). При виведенні на екран об'єкта, створеного в додатку Java за допомогою peer-інтерфейсу, насправді створюється та відображається на екрані парний йому (peer-to-peer) об'єкт графічної підсистеми ОС. Ці два парних об'єкти тісно взаємодіють під час роботи програми. Тому графічні об'єкти AWT в кожному графічному середовищі мають вигляд, характерний для цього середовища: в MS Windows, Motif, OpenLook, OpenWindows – всюди вікна, створені в AWT, виглядають як "рідні" вікна відповідного графічного середовища.

Саме через таку реалізацію peer-інтерфейсів та інших методів, написану, головним чином, на мові C++, доводиться для кожної платформи випускати свій варіант JDK.

Наслідком описаних особливостей AWT є і основні її недоліки:

- AWT може використовувати тільки стандартні елементи ОС, в зв'язку з чим AWT не дозволяє створювати віджети довільної форми;
- набір елементів управління, загальних для всіх ОС, невеликий.

У версії JDK 1.1 бібліотека AWT була перероблена. Додана можливість створення компонентів, повністю написаних на Java і не залежних від peer-інтерфейсів. Такі компоненти стали називати ***"легкими"*** (lightweight) на відміну від компонентів, реалізованих через peer-інтерфейси, названих ***"важкими"*** (heavy).

В той же час була створена потужна бібліотека "легких" компонентів Java, названа **Swing**. У ній були переписані всі реалізації інтерфейсів бібліотеки AWT, так що бібліотека Swing може

використовуватися самостійно не дивлячись на те, що всі класи з неї розширюють класи і інтерфейси бібліотеки AWT.

"Легкі" компоненти всюди виглядають однаково, зберігають заданий при створенні вигляд (***look and feel***). «Look» визначає зовнішній вигляд компонентів, а «Feel» – їх поведінку. Більш того, є можливість вже після запуску додатку змінити його зовнішній вигляд на інший, обравши інший компонент look-and-feel. Ця цікава особливість "легких" компонентів отримала назву PL&F (Pluggable Look and Feel або “plaf”). Основні стандартні *LookAndFeel* компоненти наступні:

- *CrossPlatformLookAndFeel* – вигляд компонентів не залежить від ОС та її налаштувань;
- *SystemLookAndFeel* – всі компоненти використовують налаштування L&F, встановлені в ОС.

Все це стало можливим, тому що компоненти Swing є «Lightweight», тобто створюються як зображення на поверхні батьківського вікна без використання компонентів ОС. Як наслідок, в Swing-додатку вікнами є тільки компоненти верхнього рівня (наприклад, JFrame), і всі віджети малюються на них. Природньо, що при такому способі відтворення Swing працює повільніше, ніж AWT і SWT. На сьогоднішній день проблеми з продуктивністю Swing зведені до мінімуму. Тому зараз Swing є основною бібліотекою користувацького інтерфейсу в Java. Використання Swing майже завжди краще, за винятком тих випадків, коли потрібна зворотна сумісність з Java 1.

Також в Java 2 бібліотека AWT значно розширена додаванням нових засобів малювання, виведення текстів і зображень, які отримали назву Java 2D, і засобів, що реалізують переміщення об’єктів методом Drag and Drop. Ці ж засоби малювання використовуються і в Swing-додатках. Крім того, в Java 2 включені нові методи введення-виведення Input Method Framework і засоби зв'язку з додатковими пристроями введення-виведення, такими як світлове перо або клавіатура Брайля, названі Accessibility.

Разом всі ці засоби Java 2: AWT, Swing, Java 2D, DnD, Input Method Framework і Accessibility – склали бібліотеку графічних засобів Java, названу **JFC (Java Foundation Classes)**. Опис кожного з цих засобів досить великий, тому розглянемо лише основи роботи з бібліотекою Swing.

Перш ніж йти далі, потрібно відзначити, що існує і третя стандартна бібліотека графічних компонентів – ***SWT (Standard Widget Toolkit)***. Вона не є самостійною графічною бібліотекою в повному розумінні, а являє собою крос-платформову оболонку для графічних бібліотек конкретних платформ. Наприклад, під Linux вона використовує бібліотеку Gtk+. SWT отримує доступ до ОС-бібліотек через Java Native Interface і API оболонки. За рахунок цього SWT дозволяє отримати звичний зовнішній вигляд віджетів для кожної ОС. Використання SWT робить Java-додаток більш ефективним, але робить його більш залежним від операційної системи і устаткування, вимагає ручного звільнення ресурсів і в деякій мірі порушує концепцію платформи Java.

Компонент і контейнер

Основне поняття графічного інтерфейсу користувача (ГІК) – ***компонент*** (component) графічної системи. В загальному розумінні це слово означає просто складову частину, елемент чогось, але в GUI це поняття набагато конкретніше: воно означає окремий, повністю визначений елемент, який можна використовувати в графічному інтерфейсі незалежно від інших елементів. Прикладами компонентів є: поле введення, кнопка, рядок меню, смуга прокрутки, радіо-кнопка. Саме вікно програми – теж є компонентом. Компоненти можуть бути і невидимими, наприклад, панель, яка об'єднує компоненти, сама теж є компонентом.

В AWT компоненти представлені об'єктами класу **`Component`** та його численних нащадків. У класі `Component` зібрані загальні методи роботи з будь-яким компонентом GUI. Цей клас є центром бібліотеки AWT, над якою, в свою чергу, надбудовується бібліотека Swing (рис. 6.1).

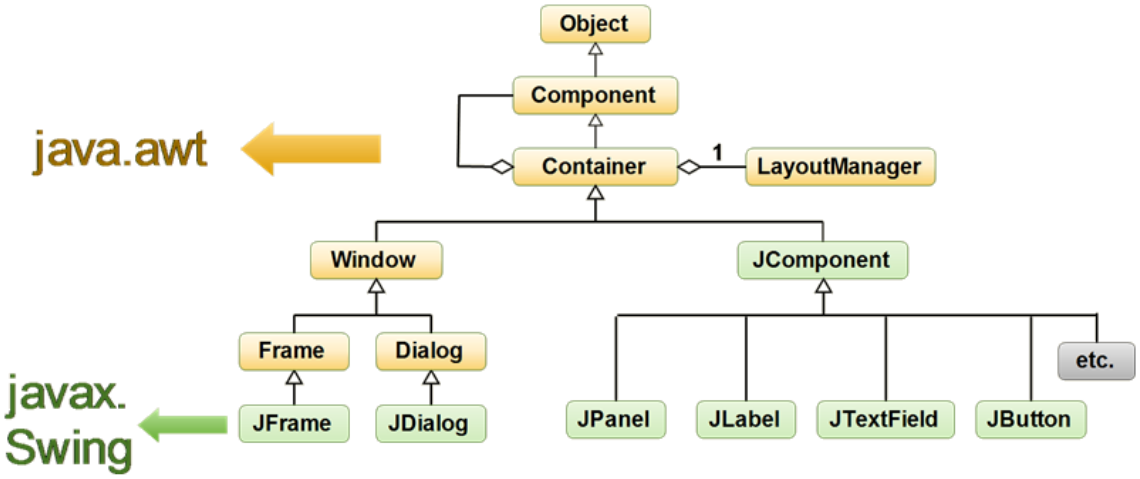

Рис. 6.1. Ієрархія класів компонентів

Кожен компонент перед виведенням на екран поміщається в *контейнер* (`Container`). Контейнер "знає", як розмістити дочірні компоненти на екрані. Зверніть увагу, що прямий спадкоємець цього класу – клас `JComponent` – є вершиною ієрархії компонентів бібліотеки Swing. Це означає, що *всі компоненти Swing є контейнерами* і можуть містити дочірні компоненти.

Створивши компонент (тобто об'єкт класу `Component` або його нащадка), слід додати його до попередньо створеного об'єкту класу `Container` або його розширення одним з методів `add()`. Оскільки клас `Container` розширює клас `Component`, в контейнер поряд з компонентами можна поміщати інші контейнери, досягаючи таким чином більшої гнучкості розташування компонентів.

Основне вікно програми, що активно взаємодіє з ОС, необхідно побудувати за правилами графічної системи. Воно повинно переміщатися по екрану, змінювати розміри, реагувати на дії миші і клавіатури. У вікні повинні бути, як мінімум, такі стандартні компоненти.

- *Рядок заголовка* (title bar), з лівого боку якого необхідно розмістити кнопку контекстного меню, а з правого – кнопки згортання і розгортання вікна і кнопку закриття програми.
- Необов'язковий *рядок меню* (menu bar) з випадними пунктами меню.
- Горизонтальна і вертикальна *смуги прокрутки* (scrollbars).
- Вікно повинне бути оточене *рамкою* (border), що реагує на дії миші.

Вікно зі всіма цими компонентами описано в класі `Frame`, а Swing-клас `JFrame` безпосередньо його успадковує. Таким чином, щоб створити вікно, потрібно або успадкувати власний клас від `JFrame`, або просто створити об'єкт класу `JFrame` в своєму класі програми. В обох випадках потім обов'язково окремо створити дочірні компоненти і помістити їх у вікно. У першому випадку це робиться в конструкторі вашого класу-вікна, а в другому – відразу після створення об'єкта `JFrame` (зазвичай в конструкторі класу додатка або в методі `main()`). Далі докладно розглянемо обидва способи.

Вікно JFrame

Клас `JFrame` представляє собою вікно і може використовуватися як для головного віна програми, так і для інших допоміжних вікон, що відкриваються під час роботи програми. Вікно `JFrame` має рамку і заголовок (з кнопками «Згорнути», «На весь екран» і «Закрити»). Воно може змінювати розміри і переміщатися екраном.

Конструктор `JFrame()` без параметрів створює порожнє вікно; інший конструктор `JFrame(String title)` створює порожнє вікно з заголовком title. Щоб написати просту програму, що виводить на екран порожнє вікно, необхідні ще три методи:

`setSize(int width, int height)` – встановлює розміри вікна. Якщо не поставити розміри, вікно буде мати нульову висоту незалежно від того, що в ньому знаходиться і користувачеві після запуску доведеться розтягувати вікно вручну. Розміри вікна включають не тільки робочу область, але також і рамку та рядок заголовка.

`setDefaultCloseOperation (int operation)` – дозволяє вказати дію, яку необхідно виконати, коли користувач закриває вікно натисканням на хрестик. Зазвичай в програмі є одне або кілька вікон, при закритті яких програма припиняє роботу. Для того, щоб запрограмувати таку поведінку, слід в параметр `operation` передати константу `EXIT_ON_CLOSE`, описану в класі `JFrame`.

`setVisible(boolean visible)` – цей метод керує видимістю вікна. Нове створене вікно за замовчанням невидиме. Щоб відобразити його на екрані, треба викликати даний метод з параметром true. Якщо викликати його з параметром false, вікно знову стане невидимим.

Тепер можемо написати програму, яка створює вікно, виводить його на екран і завершує роботу після того, як користувач закриває вікно. Зверніть увагу, при створенні GUI-додатків в кожній програмі нам буде потрібно імпортувати класи пакетів `java.swing` і `java.awt`.

```
import javax.swing.*;
public class MyClass {
     public static void main (String [] args) {
          JFrame myWindow = new JFrame("Пробне вікно");
          myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          myWindow.setSize(400, 300);
          myWindow.setVisible(true);
     }
}
```

Як правило, перед відображення вікна необхідно зробити набагато більше дій, ніж в цій простій програмі. Необхідно створити елементи управління, налаштувати їх зовнішній вигляд, розмістити в потрібних місцях вікна. Крім того, в програмі може бути багато вікон і налаштовувати їх всі в методі `main()` незручно і неправильно, оскільки це порушує принцип інкапсуляції. Логічніше було б, щоб кожне вікно займалося своїми розмірами і вмістом самостійно. Тому класична структура програми з вікнами виглядає наступним чином:

В файлі SimpleWindow.java оголошуємо клас вікна:

```
public class SimpleWindow extends JFrame {
     SimpleWindow(){
          super("Пробне окно");
          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setSize(250, 100);
     }
}
```

В головному класі програми створюємо екземпляр вікна та відображаємо його:

```
public class Program {
     public static void main (String [] args) {
          JFrame myWindow = new SimpleWindow();
          myWindow.setVisible(true);
     }
}
```

З цього спрощеного прикладу видно, що вікно описується в окремому класі, який є спадкоємцем `JFrame` і налаштовує свій зовнішній вигляд і поведінку в конструкторі (першою командою викликається конструктор суперкласу). Метод `main()` міститься в іншому класі, відповідальному за управління ходом програми. Кожен з цих класів дуже простий, кожен займається своєю справою, тому в них легко розбиратися, і такий код легко супроводжувати (тобто удосконалювати за необхідності).

Зверніть увагу, що метод `setVisible()` не викликається в класі `SimpleWindow`, що цілком логічно: за тим, де розташовані дочірні компоненти і які розміри вони мають, стежить саме вікно, а от приймати рішення про те, в який момент вікно виводиться на екран – це вже справа головного класу програми.

Розглянемо тепер приклад, більш наближений до реальних додатків.

Приклад. Каркас мінімального GUI-додатка.

```
public class MyWnd {              // Клас додатка

     private JFrame frame;        // Головне (і єдине) вікно

     public static void main(String[] args) {
          EventQueue.invokeLater(new Runnable() {          // Для створення вікна
               public void run() {                         // як правило запускається
                    try {                                  // окремий потік.
                         MyWnd window = new MyWnd();       // Створюємо вікно
                         window.frame.setVisible(true);    // та відображаємо його.
                    } catch (Exception e) {
                         e.printStackTrace();
                    }
               }
          });
     }

     public MyWnd() {   initialize(); }         // Конструктор класу додатку

     private void initialize() {                // Створення вікна та компонентів
          frame = new JFrame();                 // Створюємо саме вікно
          frame.setBounds(100, 100, 450, 300);
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // Тут код створення дочірніх компонентів та додавання для них обробників
     }
}
```

Зверніть увагу, що зазвичай процес створення вікна запускається окремим потоком, тому що створення складного вікна може займати багато часу, і це не повинно впливати на роботу основного потоку програми.

Контейнери

Swing має контейнери верхнього рівня, тобто вікна, які відповідають вікнам в ОС, і прості контейнери, які не відповідають ніяким об'єктам ОС, а просто вимальовуються на поверхні контейнерів верхнього рівня

Контейнери верхнього рівня:

- `JFrame` – вікно додатку;
- `JDialog` – діалог додатку;
- `JColorChooser` – діалог вибору кольору;
- `JFileChooser` – діалог вибору файлів та директорій;
- `FileDialog` – діалог вибору файлів та директорій (awt компонент);
- `JApplet` – головне вікно апплета.

Прості контейнери:

- `Jpanel` – проста панель для групування елементів, включаючи вкладені панелі;
- `JtoolBar` – панель інструментів (як правило це кнопки);
- `JScroolPane` – панель прокрутки, що дозволяє продивлятись вміст дочірнього елемента;
- `JDesktopPane` – контейнер для створення віртуального робочого стола або додатків на основі MDI (multiple-document interface);
- `JEditorPane`, `JTextPane` – контейнери для відображення складного документа як HTML або RTF;
- `JTabbedPane` – контейнер для керування закладками;

- `JSplitPane` – контейнер, що має дві області, розділені перегородкою, та дозволяє змінювати їх розмір, переміщуючи перегородку.

Все контейнери є нащадками класу `Container` та наслідують від нього ряд корисних методів:

- `add(Component component)` – добавляє компонент до контейнера;
- `remove(Component component)` – видаляє компонент з контейнера;
- `removeAll()` – видаляє всі компоненти з контейнера;
- `getComponentCount()` – повертає кількість компонентів у контейнері.

В класі `Container` визначено близько двох десятків методів для управління набором дочірніх компонентів. Ці методи схожі на методи класу-колекції. По суті контейнер і є колекцією, але колекцією візуальних компонентів, яка крім зберігання елементів займається також їх просторовим розташуванням і відображенням. Зокрема, будь-який контейнер має метод `getComponentAt(int x, int y)` – він повертає компонент, до якого належить точка з заданими координатами відносно лівого верхнього кута контейнера. Немає потреби розглядати абстрактний контейнер, тому відразу перейдемо до його найбільш часто використовуваного нащадка – класу `JPanel`.

Панель вмісту

Елементи управління ніколи не розміщуються безпосередньо у вікні. Для їх розміщення служить панель вмісту, що займає весь робочий простір вікна. Посилання на цю панель можна отримати за допомогою методу `getContentPane()` класу `JFrame`. За допомогою методу `add()` можна додати на неї будь-який елемент управління. Метод `setContentPane(JPanel panel)` дозволяє замінити панель вмісту вікна.

В наступних прикладах для простоти будемо використовувати тільки один елемент управління – кнопку. Цей елемент описується класом `JButton` і створюється конструктором з параметром типу `String` – надписом. Додамо кнопку в панель вмісту нашого вікна командами:

```
JButton newButton = new JButton(); getContentPane().add(newButton);
```

В результаті отримаємо вікно з кнопкою, що зображене на рис. 6.2. Кнопка займає всю доступну площу вікна. Такий ефект корисний не у всіх програмах, і тому далі ми розглянемо різні способи керування розташуванням елементів у контейнері.

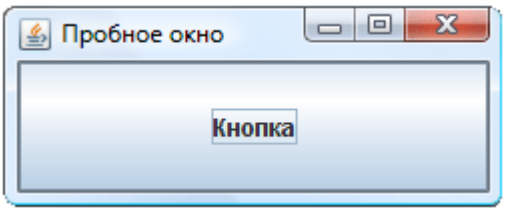

Рис. 6.2. Просте вікно із кнопкою

Клас JPanel

Панель `JPanel` – це прямокутний простір, на якому можна розміщувати інші компоненти за допомогою методів, успадкованих від класу `Container`.

У попередньому прикладі додана на панель вмісту кнопка зайняла весь її простір. Це відбувається не завжди. Насправді кожна панель має так званий *менеджер розміщення*, який визначає стратегію взаємного розташування елементів, що додаються на панель. Менеджер розміщення можна замінити методом `setLayout(LayoutManager manager)`. Але щоб передати в цей метод потрібний параметр, необхідно знати класи ієрархії `LayoutManager`.

Верстка форми і класи розміщень (layouts)

Для автоматичного позиціонування і розрахунку розмірів дочірніх елементів контейнери використовують спеціальні об'єкти – *менеджери розміщення* (або просто «розміщення»). Нижче наведено список стандартних класів розміщень:

- *FlowLayout* – розміщує елементи по порядку в тому ж напрямку, що і орієнтація контейнера (зліва направо за замовчанням), застосовуючи один з п'яти видів вирівнювання, зазначеного при створенні менеджера. Даний менеджер використовується за замовчанням в більшості контейнерів;
- *GridLayout* – розміщує елементи у вигляді таблиці. Кількість стовпців і рядків вказується при створенні менеджера. За замовчанням він має один рядок та число стовпців, рівне кількості елементів;
- *BorderLayout* – розміщує кожен елемент в одну з п'яти областей, яка зазначається при додаванні елемента в контейнер: "North", "South", "East", "West" і "Center";
- *SpringLayout* – розміщує елементи у відповідності з обмеженнями, які задані в кожному з них.

Якщо встановити для контейнера в якості розміщення порожній об'єкт викликом `setLayout(null)`, то всі дочірні компоненти необхідно буде позиціонувати вручну.

Менеджер послідовного розміщення FlowLayout

Найпростшим менеджером розміщення є `FlowLayout`. Він розміщує компоненти строго по черзі, зліва направо і зверху вниз, подібно тексту на сторінці, і в залежності від розмірів панелі. Якщо черговий елемент не поміщається в поточному рядку, то він переноситься на наступний рядок. Розглянемо приклад, в якому змінимо конструктор класу `SimpleWindow` (див. приклад вище) наступним чином:

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());
    panel.add(new JButton("Кнопка"));
    panel.add(new JButton("+"));
    panel.add(new JButton("-"));
    panel.add(new JButton("Кнопка з довгим написом"));
    setContentPane(panel);
    setSize(250, 100);
}
```

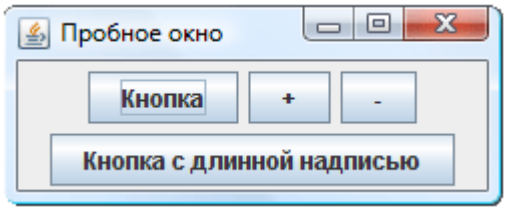

Рис. 6.3. Демонстрація FlowLayout

У вікні, що з'являється після запуску програми (рис. 6.3), чотири кнопки розташовані як слова у тексті із вирівнюванням по центру. Ефект буде краще помітний, якщо змінювати розміри вікна під час роботи програми.

Проаналізуємо текст прикладу. Об’єкт `FlowLayout` створюється конструктором без параметрів. Слід звернути увагу, що цей об’єкт не зберігається у змінну. Це виправдано в тих випадках, коли в подальшому не буде потреби звертатися до створюваного об'єкту. В нашому випадку після зв'язування об’єкта `FlowLayout` із панеллю вони надалі взаємодіятимуть між собою автоматично. Так само додаємо на панель нові кнопки. Подібна конструкція, коли об’єкт

створюється тільки для одноразового використання, дуже поширена в програмах із GUI. Якщо потрібно буде звертатися до створених кнопок в інших методах програми, то їх потрібно зберегти в змінні в класі `SimpleWindow`.

Менеджер граничного розміщення BorderLayout

Цей менеджер розділяє панель на п'ять областей: центральну, верхню, нижню, праву і ліву. У кожну з цих областей можна додати рівно по одному компоненту, причому компонент буде займати всю відведену для нього область. Верхні і нижні компоненти будуть розтягнуті по ширині, правий і лівий – по висоті, а компонент, для якого відведено центр, буде розтягнутий до повного заповнення простору, що залишився.

При додаванні елемента на панель з менеджером розміщення `BorderLayout` необхідно в методі `add()` додатково вказувати область, в якій він розміститься (за замовчуванням елемент буде поміщено до центральної області). П'ять областей `BorderLayout` умовно позначаються назвами сторін світу: "North", "South", "East", "West" і "Center"; їм відповідають константи, визначені в класі `BorderLayout`: `NORTH`, `SOUTH`, `EAST`, `WEST` і `CENTER`.

Панель вмісту в першому прикладі має саме таке розміщення. Кнопка за замовчуванням була додана в центральну область, саме тому вона займала весь робочий простір вікна. Додамо тепер кнопки в кожну з п'яти областей (рис. 6.4):

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    getContentPane().add(new JButton("Кнопка"), BorderLayout.NORTH);
    getContentPane().add(new JButton("+"), BorderLayout.EAST);
    getContentPane().add(new JButton("-"), BorderLayout.WEST);
    getContentPane().add(
           new JButton("Кнопка з довгим написом"), BorderLayout.SOUTH);
    getContentPane().add(new JButton("В ЦЕНТР!"));
    setSize(250, 100);
}
```

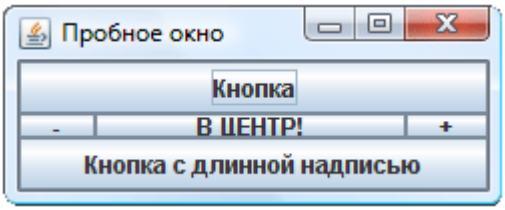

Рис. 6.4. Демонстрація BorderLayout

Принцип роботи `BorderLayout` – добре видно, якщо змінювати розміри вікна. Дане розміщення не випадково використовується в панелі вмісту за замовчанням. Більшість програм користуються областями по краях вікна, щоб розташувати в них панелі інструментів, рядок стану і т. п. При цьому обмеження на один компонент в центральній області абсолютно не суттєве, адже цим компонентом може бути панель з багатьма дочірніми компонентами і з будь-яким власним менеджером розташування.

Менеджер табличного розміщення GridLayout

`GridLayout` розбиває контейнер на комірки однакового розміру і контейнер стає схожим на таблицю. Кожен дочірній компонент займає одну комірку. Комірки заповнюються по черзі, починаючи з верхньої лівої (рис. 6.5).

Такий менеджер, на відміну від розглянутих вище, створюється конструктором з параметрами (чотири цілих числа). Необхідно вказати кількість стовпців, рядків, а також відстань між комірками по горизонталі і по вертикалі, наприклад:

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(2, 3, 5, 10));
    panel.add(new JButton("Кнопка"));
    panel.add(new JButton("+"));
    panel.add(new JButton("-"));
    panel.add(new JButton("Кнопка з довгим написом"));
    panel.add(new JButton("ще кнопка"));
    setContentPane(panel);
    setSize(250, 100);
}
```

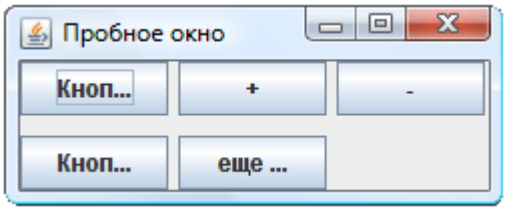

Рис. 6.5. Демонстрація GridLayout

Менеджер блочного розміщення BoxLayout і клас Box

Менеджер `BoxLayout` розміщує елементи в рядок або в стовпчик. Зазвичай для роботи з даним сервісом використовують допоміжний клас `Box`, що представляє собою панель, для якої вже налаштоване блочне розміщення. Створюється така панель не конструктором, а одним з двох статичних методів, визначених в класі `Box`: `createHorizontalBox()` і `createVerticalBox()`.

Елементи, додані на панель з блоковим розміщенням, розташовуються один за іншим. Відстань між елементами за замовчанням нульова. Однак замість компонента можна додати невидимий «буфер», єдине завдання якого – розсовувати сусідні елементи, забезпечуючи між ними задану відстань. Горизонтальний буфер створюється статичним методом `createHorizontalStrut(int width)`, а вертикальний – методом `createVerticalStrut(int height)`. Обидва названі методи визначені в класі `Box`, а цілочисловий параметр в кожному з них визначає розмір буфера.

Крім того, до `BoxLayout` можна додати ще один спеціальний елемент – своєрідну «пружину». Якщо розмір контейнера буде більшим, ніж необхідно для оптимального розміщення всіх елементів, ті з них, які здатні розтягуватися, будуть намагатися заповнити додатковий простір собою. Якщо ж розмістити серед елементів одну або кілька «пружин», додатковий вільний простір буде розподілятися на ці проміжки між елементами. Горизонтальна і вертикальна пружини створюються відповідно методами `createHorizontalGlue()` і `createVerticalGlue()`.

Особливості роботи цього менеджера покажемо на наочному прикладі. Розташуємо чотири кнопки вертикально, поставивши між двома центральними «пружину», а між іншими – буфери в 10 пікселів (рис. 6.6).

```
SimpleWindow() {
    super("Пробне вікно");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Box box = Box.createVerticalBox();
    box.add(new JButton("Кнопка"));
    box.add(Box.createVerticalStrut(10));
    box.add(new JButton("+"));
    box.add(Box.createVerticalGlue());
    box.add(new JButton("-"));
    box.add(Box.createVerticalStrut(10));
    box.add(new JButton("Кнопка з довгим написом"));
    setContentPane(box);
    setSize(250, 100);
```

```
}
```

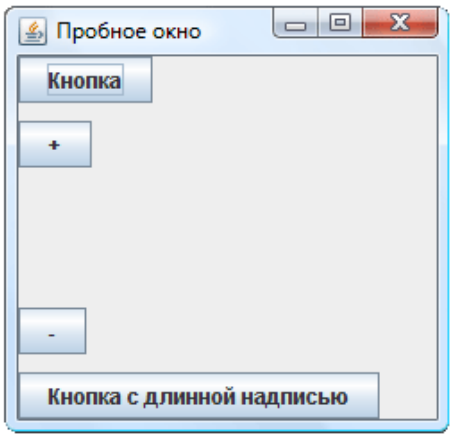

Рис. 6.6. Демонстрація `BoxLayout`

Особливості вирівнювання елементів

У попередньому прикладі з вертикальною панеллю всі кнопки виявилися вирівняні по лівому краю. Таке вирівнювання по горизонталі прийнято за замовчанням. Для того, щоб встановити інші правила вирівнювання компонентів, використовуються методи `setAlignmentX(float alignment)` – вирівнювання по горизонталі і `setAlignmentY(float alignment)` – вирівнювання по вертикалі. Як параметр найпростіше використовувати константи, визначені в класі `JComponent`. Для вирівнювання по горизонталі служать константи `LEFT_ALIGNMENT` (по лівому краю), `RIGHT_ALIGNMENT` (по правому краю) і `CENTER_ALIGNMENT` (по центру). Для вирівнювання по вертикалі – `BOTTOM_ALIGNMENT` (по нижньому краю), `TOP_ALIGNMENT` (по верхньому краю) і `CENTER_ALIGNMENT` (по центру).

Однак вирівнювання працює трохи інакше, ніж очікується. Щоб це виявити, змінимо попередній приклад, вирівнявши третю кнопку по правому краю. Для цього замінимо рядок

```
box.add(new JButton("-"));
```

на три інших:

```
JButton rightButton = new JButton("-");
rightButton.setAlignmentX(JComponent.RIGHT_ALIGNMENT);
box.add(rightButton);
```

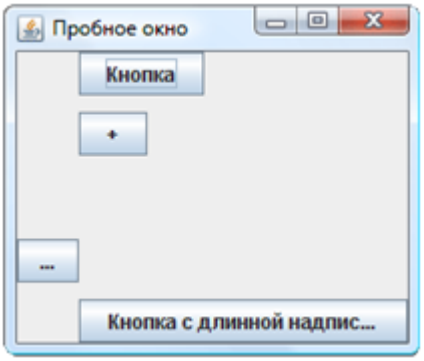

Рис. 6.7. Демонстрація вирівнювання компонентів

Після запуску програми побачимо вікно (рис. 6.7), в якому кнопки розташовані не так, як очікувалося. Ми звикли, що вирівнювання по правому краю притискає об'єкт до правого краю контейнера, але в даному випадку перемістилися всі елементи, причому кнопка з вирівнюванням

по правому краю виявилася самою лівою. Насправді при вирівнюванні по правому краю компонент не притискається до правого краю контейнера; замість цього він притискається правим краєм до невидимої лінії вирівнювання. Всі інші компоненти притискаються до цієї лінії своїм лівим краєм (за замовчуванням), тому і виходить спостережуваний ефект.

Єдина складність для розробника в тому, що не завжди легко зрозуміти, де саме пройде умовна лінія вирівнювання. Її положення залежить від розмірів і параметрів вирівнювання всіх елементів контейнера. Однак корисно запам'ятати просте правило: якщо всі елементи в контейнері вирівняні однаково, то лінія вирівнювання знаходиться біля краю контейнера (як це і було в попередньому прикладі).

Можна також поекспериментувати з вертикальною панеллю, задаючи різне вирівнювання для її елементів: логіка вирівнювання буде схожою.

Розміщення компонентів вручну

Якщо встановити для контейнера в якості розміщення порожній об'єкт викликом `setLayout(null)`, то положення дочірніх компонентів не буде обраховуватись автоматично і тоді можна вручну задати їхні розміри та положення. При цьому координати кожного компонента необхідно вказати явно, оскільки вони ніяк не залежать від розмірів панелі і від положення інших компонентів. За замовчанням координати компонента встановлюються рівними нулю (тобто він розташований в лівому верхньому кутку контейнера). Розмір елемента (ширина і висота) за замовчанням також дорівнюють нулю, тому якщо не задати розміри явно, елемент відображатися не буде.

Для задання положення і розмірів будь-який компонент має наступні методи:

```
setLocation(int x, int y);
setLocation(Point point);
setSize(int width, int height),
setSize(Dimension size)
```

Обидва метода `setLocation()` задають положення лівого верхнього кута компонента відносно лівого верхнього кута контейнера. Координати точки кута можна задати двома способами: двома цілими числами, або об'єктом класу **Point**. Об'єкт `Point` має дві властивості – `x` і `y` – і створюється конструктором `Point(int x, int y)`. Останній спосіб є найбільш вживаним, оскільки багато методів графічної бібліотеки видають координати точок у вигляді об'єктів `Point`. Наприклад, за допомогою методу `getLocation()` можна дізнатись поточне положення будь-якого компонента. Тоді наступний рядок помістить елемент b в точності в те місце, яке займає елемент a:

```
b.setLocation(a.getLocation());
```

В методах `setSize()` також маємо два способи задання розмірів. Об'єкт **Dimension**, аналогічно `Point`, зберігає два цілі числа, має дві властивості – `width` і `height`, – а також конструктор з двома параметрами `Dimension(int width, int height)`. Отримати поточний розмір компонента можна методом `getSize()`, що повертає об'єкт класу `Dimension`. Тоді наступний рядок зробить елемент b точно такого ж розміру, як і елемент a:

```
b.setSize(a.getSize());
```

Автоматичне визначення розмірів компонентів

Всі менеджери розміщення ігнорують явно задані координати і розміри дочірніх компонентів. Менеджер сам визначає їх координати і розміри. Але все ж деякі властивості компонентів можуть впливати на те, як саме менеджер буде їх розміщувати.

Ми бачили раніше, що в деяких випадках компоненти заповнюють весь доступний їм простір: в `BorderLayout` компонент заповнював всю центральну область, а в разі `GridLayout` – всю комірку таблиці. У разі `FlowLayout`, навпаки, компоненти не збільшувалися понад деякого

розміру, навіть якщо поруч залишався вільний простір. Подібними аспектами поведінки компонентів найлегше керувати, використовуючи їх властивості.

Кожен компонент має три пари розмірів: *мінімально допустимий, максимально допустимий і бажаний*. Вони доступні через властивості `minimumSize`, `preferredSize` і `maximumSize` типу `Dimension` (а значить є відповідні методи – «геттери» і «сеттери»). Наприклад, кнопка `JButton` за замовчуванням має мінімальний розмір – нульовий, максимальний розмір – не обмежений (що позначається у властивості `maximumSize` як `width = -1` і `height = -1`), а бажаний розмір залежить від напису на кнопці (обчислюється як розмір тексту напису плюс розміри полів).

Різні менеджери розміщення використовують ці властивості по-різному:

`FlowLayout` завжди використовує бажані розміри елементів (`preferredSize`);

`BorderLayout` використовує бажану ширину для правого і лівого, а також бажану висоту для верхнього і нижнього компонентів; інші розміри підганяються під розміри контейнера;

`GridLayout` намагається підігнати розміри всіх компонентів під розмір комірок;

`BoxLayout` орієнтується на бажані розміри.

У випадках, коли компонент повинен заповнити весь доступний простір, або навпаки, коли місця недостатньо, більшість (але не всі) менеджери розміщення враховують властивості `minimumSize` і `maximumSize`. Ці властивості за правильного налаштування обмежують стиснення компонентів при зменшенні розмірів контейнера, в якому вони знаходяться, а також обмежують їх розтягнення, коли розмір контейнера збільшується.

Багато компонентів, наприклад кнопки, взагалі не повинні збільшуватися і зменшуватися, а повинні зберігати свій бажаний розмір. Для них використовується наступний простий прийом:

```
component.setMinimumSize(component.getPreferredSize());
component.setMaximumSize(component.getPreferredSize());
```

Розрахунок ризміру вікна

У попередніх прикладах розмір вікна задавався явно методом `setSize()`. Однак на практиці важко визначити найбільш підходящі розміри вікна, особливо якщо вікно має багато дочірніх компонентів, і їх розміри визначаються різними менеджерами розміщення. Безумовно, найбільш підходящим буде варіант, за якого всі елементи вікна мають бажані розміри або близькі до них.

Всі контейнери-вікна мають метод `pack()`, який автоматично підбирає і встановлює розміри вікна в залежності від його вмісту. При цьому враховуються бажані розміри всіх дочірніх компонентів, вимоги менеджерів розміщення, і вікно приймає мінімальний розмір, достатній для розміщення всіх дочірніх компонентів.

Зауважте, якщо головна панель вікна (властивість `contentPane`) не має менеджера розміщення (`setLayoutManager(null)`), то вікно не має алгоритму для обчислення свого бажаного розміру і в цьому випадку метод `pack()` не працює.

Рамки

Зазвичай рамка використовується щоб візуально відокремити компонент від оточення. Рамку можуть мати панелі, написи, кнопки і багато інших компонентів. Деякі з них мають рамку за замовчуванням, інші – не мають, але в будь-якому випадку можна програмно керувати зовнішнім виглядом рамки. Рамка встановлюється методом `setBorder(Border border)`. Як параметр потрібно передати об'єкт класу `Border`, який описує створювану рамку і може мати безліч різних властивостей. `Border` – це абстрактний клас, тому для створення рамок використовуються його спадкоємці:

- EmptyBorder – порожня рамка, що не промальовується, але дозволяє створити відступи по краях компонента. Розміри відступів задаються в конструкторі чотирма цілими числами.
- TitledBorder – рамка з заголовком, для якої необхідно задати текст заголовка і інші необов'язкові параметри, які керують розташуванням заголовка і його стилем.
- EtchedBorder – рамка з ефектом вдавлення або опуклості.

- BevelBorder – об'ємна рамка (опукла або вдавлена). Можна налаштувати кольори, що використовуються для отримання об'ємних ефектів.
- SoftBevelBorder – те ж, що і BevelBorder, але дозволяє додатково округлити кути рамки.
- LineBorder – рамка у вигляді лінії. Можна вибирати колір, товщину і стиль лінії, а також округлити кути.
- MatteBorder – рамка у вигляді повторюваного малюнка.
- CompoundBorder – об'єкт цього класу об'єднує дві рамки, що передаються в якості параметрів конструктора, в одну складну рамку.

Всі перераховані класи описані в пакеті `javax.swing.border`.

Для прикладу створимо шість панелей з різними рамками і розмістимо їх у вигляді таблиці. Щоб не повторювати дії зі створення нової панелі, винесемо їх в окремий метод:

```
private JPanel createPanel(Border border, String text) {
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.add(new JButton(text));
    panel.setBorder(
              new CompoundBorder( border, new EmptyBorder(12, 12, 12, 12) ) );
    return panel;
}
```

Метод `createPanel()` створює панель, весь простір якої займає одна кнопка. В якості параметрів передається напис для кнопки і рамка, яку необхідно використовувати для панелі. Зверніть увагу, що передана рамка комбінується з «порожньою» рамкою `EmptyBorder` – цей прийом дозволяє забезпечити відступи між рамкою і внутрішнім вмістом панелі (в нашому випадку – кнопкою). Тепер створимо кілька схожих панелей з різними типами рамок (рис. 6.8).

```
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(2, 3, 5, 10));
    panel.setBorder(new EmptyBorder(12, 12, 12, 12));
    panel.add( createPanel(
              new TitledBorder("Рамка с заголовком"), "TitledBorder" ));
    panel.add( createPanel(
              new EtchedBorder(), "EtchedBorder" ));
    panel.add( createPanel(
              new BevelBorder(BevelBorder.LOWERED), "BevelBorder" ));
    panel.add( createPanel(
              new SoftBevelBorder(BevelBorder.RAISED), "SoftBevelBorder" ));
    panel.add( createPanel(
              new LineBorder(Color.ORANGE, 4), "LineBorder" ));
    panel.add( createPanel(
              new MatteBorder(new ImageIcon("1.png")), "MatteBorder" ));
```

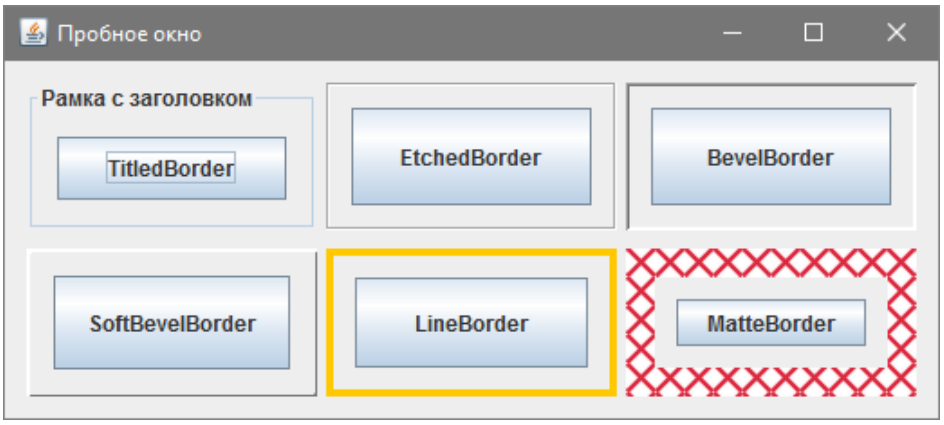

Рис. 6.8. Демонстрація різних видів рамок

У цьому прикладі використані два нових класи: `Color` і `ImageIcon`.

Об'єкт класу **Color** представляє колір. Такий об'єкт можна створити, вказавши параметри кольору в форматі RGB, наприклад, `new Color(128,255,128)` – це світло–зелений колір. У класі `Color` безліч стандартних кольорів визначено у вигляді констант, наприклад: `Color.RED`, `Color.BLACK`, `Color.ORANGE` та ін.

Об'єкт класу **ImageIcon** описує графічне зображення, яке можна використовувати в якості «іконки» в GUI. Такий об'єкт легко створити, вказавши шлях до файлу зображення. У прикладі використовується відносне ім'я файлу "1.png" – такий файл повинен знаходитися в директорії проекту. Крім `ImageIcon` є також споріднені класи `Image` і `BufferedImage`: – всі вони так чи інакше представляють зображення. Відмінність між ними полягає в тому, як саме зображення може створюватися і використовуватися в програмі.

Поняття і принципи usability. Розташування елементів екранної форми

У найширшому сенсі термін ***юзабіліті*** означає ступінь зручності користування тим чи іншим предметом. Стосовно ж до інтерфейсу користувача, під юзабіліті мають на увазі сукупність наступних факторів:

– логічність і простота розташування різних графічних елементів;

– простота і зручність навігації;

– продуманість розташування елементів управління;

– загальна легкість сприйняття інтерфейсу людиною.

Існує безліч правил і рекомендацій юзабіліті, які стосуються як візуального дизайну інтерфейсу, так і логіки роботи з ним. Ми ж торкнемося лише найбільш простих правил, що регулюють взаємне розташування візуальних компонентів. Для дотримання цих правил достатньо правильно налаштувати менеджери розміщення і параметри окремих компонентів.

1. Елементи управління в діалоговому вікні бувають двох видів – введення-виведення (редагування) даних і підтвердження такого введення. Елементи, пов’язані із редагуванням (або внесенням) інформації, повинні бути розташовані зверху, а кнопки підтвердження – знизу в горизонтальний ряд.

2. Розміщення редагуючих елементів управління слід виконувати зверху вниз в порядку важливості редагованої ними інформації. Обов'язково також задати ***порядок табуляції*** (tab order) для переходу між елементами за допомогою клавіатури. В порядку табуляції спочатку повинні йти елементи редагування, потім кнопка підтвердження, потім кнопка скасування дії, а потім – інші кнопки (додаткові дії).

3. Необхідно витримувати відстань між елементами. Тісно пов'язані елементи (такі як текстове поле і підпис до нього) повинні відстояти один від одного на 6 пікселів. Логічно згруповані елементи – на 12 пікселів (наприклад кнопки "Ok" і "Cancel", або поля для введення логіна і пароля). Всі інші елементи повинні перебувати на відстані 17 пікселів один від одного. Не слід забувати і про відступи між елементами управління і рамкою вікна.

Работа с GUI редактором IDEA

IDE IntelliJ IDEA має багатий набір засобів для швидкого і зручного створення графічного інтерфейсу користувача із мінімальною необхідністю написання програмного коду. Всі ці засоби об'єднані в редакторі GUI.

Робота з GUI редактором IDEA докладно описана в книзі:

Давыдов С., Ефимов А. IntelliJ IDEA. Профессиональное программирование на Java. – ***С. 398 – 428.***

Скачати цю книгу можна за посиланням: http://www.twirpx.com/file/142621/.

GUI редактор WindowBuilder для IDE Eclipse

Eclipse не містить вбудованого GUI-редактора. Його функції беруть на себе різні доповнення.

WindowBuilder – це найбільш популярний візуальний дизайнер інтерфейсів на Swing, GWT та SWT для середовища Eclipse. Він виконаний у вигляді плагіна Eclipse. На даний момент WindowsBuilder поширюється вільно.

Сторінка проекту: http://www.eclipse.org/windowbuilder/

Установка WindowsBuilder в Eclipse

1) Завантажуємо WindowBuilder з офіційного сайта:

http://www.eclipse.org/windowbuilder/download.php.

Для своєї версії Eclipse вибираємо "**Zipped Update Site**", вибираємо дзеркало для завантаження і зберігаємо .zip-файл, наприклад "WB_v1.9.1_UpdateSite.zip".

2) Розпаковуємо архів, наприклад в папку "D:\Eclipse\WB_v1.6.0"

3) В IDE Eclipse в меню вибираємо **Help® Install new Software® Add**.

4) У вікні "**Add Repository**" в полі "**Name**" задаємо "WindowsBuilder", тиснемо кнопку "**Local**" і вибираємо папку з установочними файлами – "D:\Eclipse\WB_v1.6.0". Тиснемо "**ОК**".

5) У вікні "Install" відмічаємо всі встановлювані компоненти: GroupLayout, InfraStructure, SwingDesigner, натискаємо "**Next**" і надалі використовуємо інструкції майстра установки.

Створення екранної форми

1) Створюємо Java-проект (File® New® Java Project).

2) Далі натискаємо правою кнопкою миші на створеному проекті, вибираємо **New® Other** (або ж натискаємо Ctrl+N) і у вікні шукаємо папку **WindowBuilder**, а в ній вибираємо "**Swing Designer**"® "**Application Window**".

В результаті буде створений клас, в тілі якого згенерується заголовок коду для створення стандартної екранної форми.

Для того, щоб переключитися в графічний режим редагування форми, необхідно натиснути кнопку "**Design**", розташовану знизу від робочої області Eclipse. У режимі Design можна перетягувати елементи на форму і задавати їх властивості.