Identifying Classes and Objects

• The core activity of object-oriented design is determining the classes and objects that will make up the solution

• The classes may be part of a class library, reused from a previous project, or newly written

• One way to identify potential classes is to identify the objects discussed in the requirements

• Objects are generally nouns, and the services that an object provides are generally verbs
Identifying Classes and Objects

• A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

• Of course, not all nouns will correspond to a class or object in the final solution.
Identifying Classes and Objects

• Remember that a class represents a group (classification) of objects with the same behaviors

• Generally, classes that represent objects should be given names that are singular nouns

• Examples: Coin, Student, Message

• A class represents the concept of one such object

• We are free to instantiate as many of each object as needed
Identifying Classes and Objects

• Sometimes it is challenging to decide whether something should be represented as a class

• For example, should an employee's address be represented as a set of instance variables or as an Address object

• The more you examine the problem and its details the more clear these issues become

• When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities
Identifying Classes and Objects

• We want to define classes with the proper amount of detail

• For example, it may be unnecessary to create separate classes for each type of appliance in a house

• It may be sufficient to define a more general Appliance class with appropriate instance data

• It all depends on the details of the problem being solved
Identifying Classes and Objects

• Part of identifying the classes we need is the process of assigning responsibilities to each class.

• Every activity that a program must accomplish must be represented by one or more methods in one or more classes.

• We generally use verbs for the names of methods.

• In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design.
Static Class Members

• Recall that a static method is one that can be invoked through its class name.

• For example, the methods of the Math class are static:

\[
\text{result} = \text{Math.sqrt}(25)
\]

• Variables can be static as well.

• Determining if a method or variable should be static is an important design decision.
The static Modifier

• We declare static methods and variables using the static modifier

• It associates the method or variable with the class rather than with an object of that class

• Static methods are sometimes called class methods and static variables are sometimes called class variables

• Let's carefully consider the implications of each
Static Variables

• Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

  private static float price;

• Memory space for a static variable is created when the class is first referenced

• All objects instantiated from the class share its static variables

• Changing the value of a static variable in one object changes it for all others
Static Methods

```java
public class Helper {
    public static int cube (int num) {
        return num * num * num;
    }
}
```

- Because it is declared as static, the `cube` method can be invoked through the class name:

  ```java
  value = Helper.cube(4);
  ```
Static Class Members

• The order of the modifiers can be interchanged, but by convention visibility modifiers come first.

• Recall that the main method is static – it is invoked by the Java interpreter without creating an object.

• Static methods cannot reference instance variables because instance variables don't exist until an object exists.

• However, a static method can reference static variables or local variables.
Static Class Members

• Static methods and static variables often work together

• The following example keeps track of how many Slogan objects have been created using a static variable, and makes that information available using a static method

• See SloganCounter.java
• See Slogan.java
public class SloganCounter
{
    //-------------------------------------------------------------------------------
    // Creates several Slogan objects and prints the number of
    // objects that were created.
    //-------------------------------------------------------------------------------
    public static void main (String[] args)
    {
        Slogan obj;

        obj = new Slogan ("Remember the Alamo.");
        System.out.println (obj);

        obj = new Slogan ("Don't Worry. Be Happy.");
        System.out.println (obj);

        continue
    }
}
obj = new Slogan ("Live Free or Die.");
System.out.println (obj);

obj = new Slogan ("Talk is Cheap.");
System.out.println (obj);

obj = new Slogan ("Write Once, Run Anywhere.");
System.out.println (obj);

System.out.println();
System.out.println ("Slogans created: " + Slogan.getCount());
}
obj = new Slogan("Live Free or Die.");
System.out.println(obj);

obj = new Slogan("Talk is Cheap.");
System.out.println(obj);

obj = new Slogan("Write Once, Run Anywhere.");
System.out.println(obj);

System.out.println();
System.out.println("Slogans created: " + Slogan.getCount());
}
public class Slogan {
    private String phrase;
    private static int count = 0;

    // Constructor: Sets up the slogan and counts the number of instances created.
    public Slogan (String str) {
        phrase = str;
        count++;
    }
}

continue
public String toString()
{
    return phrase;
}

public static int getCount ()
{
    return count;
}
Class Relationships

• Classes in a software system can have various types of relationships to each other

• Three of the most common relationships:
  – Dependency: A uses B
  – Aggregation: A has-a B
  – Inheritance: A is-a B

• Let's discuss dependency and aggregation further

• Inheritance is discussed in detail in Chapter 9
Aggregation

• An aggregate is an object that is made up of other objects

• Therefore aggregation is a has-a relationship
  – A car has a chassis

• An aggregate object contains references to other objects as instance data

• This is a special kind of dependency; the aggregate relies on the objects that compose it
Aggregation

• In the following example, a Student object is composed, in part, of Address objects

• A student has an address (in fact each student has two addresses)

• See StudentBody.java
• See Student.java
• See Address.java
public class StudentBody
{
    public static void main (String[] args)
    {
        Address school = new Address ("800 Lancaster Ave.", "Villanova", "PA", 19085);
        Address jHome = new Address ("21 Jump Street", "Lynchburg", "VA", 24551);
        Student john = new Student ("John", "Smith", jHome, school);
        Address mHome = new Address ("123 Main Street", "Euclid", "OH", 44132);
        Student marsha = new Student ("Marsha", "Jones", mHome, school);
        System.out.println (john);
        System.out.println ();
        System.out.println (marsha);
    }
}
public class StudentBody
{
    public static void main (String[] args)
    {
        Address school = new Address ("800 Lancaster Ave.", "Villanova", "PA", 19085);
        Address jHome = new Address ("21 Jump Street", "Lynchburg", "VA", 24551);
        Student john = new Student ("John", "Smith", jHome, school);
        Address mHome = new Address ("123 Main Street", "Euclid", "OH", 44132);
        Student marsha = new Student ("Marsha", "Jones", mHome, school);
        System.out.println (john);
        System.out.println (marsha);
    }
}
public class Student
{
    private String firstName, lastName;
    private Address homeAddress, schoolAddress;

    //--- Constructor: Sets up this student with the specified values. --
    public Student (String first, String last, Address home, Address school)
    {
        firstName = first;
        lastName = last;
        homeAddress = home;
        schoolAddress = school;
    }

public String toString()
{
    String result;

    result = firstName + " " + lastName + "\n";
    result += "Home Address: \n" + homeAddress + "\n";
    result += "School Address: \n" + schoolAddress;

    return result;
}
public class Address {
    private String streetAddress, city, state;
    private long zipCode;

    // Constructor: Sets up this address with the specified data.
    public Address (String street, String town, String st, long zip) {
        streetAddress = street;
        city = town;
        state = st;
        zipCode = zip;
    }
}

continue
// Returns a description of this Address object.
public String toString() {
    String result;

    result = streetAddress + "\n";
    result += city + ", " + state + " " + zipCode;

    return result;
}
Aggregation in UML

StudentBody

+ main (args : String[]) : void

Student

- firstName : String
- lastName : String
- homeAddress : Address
- schoolAddress : Address

+ toString() : String

Address

- streetAddress : String
- city : String
- state : String
- zipCode : long

+ toString() : String
Interfaces

- A Java interface is a collection of abstract methods and constants
- An abstract method is a method header without a method body
- An abstract method can be declared using the modifier abstract, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement
Interfaces

interface is a reserved word

```java
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2 (double value, char ch);
    public boolean doTheOther (int num);
}
```

None of the methods in an interface are given a definition (body)

A semicolon immediately follows each method header
Interfaces

• An interface cannot be instantiated

• Methods in an interface have public visibility by default

• A class formally implements an interface by:
  – stating so in the class header
  – providing implementations for every abstract method in the interface

• If a class declares that it implements an interface, it must define all methods in the interface
public class CanDo implements Doable
{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
Interfaces

- **ActionListener** is an example of an interface
  
  - [http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/event/ActionListener.html](http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/event/ActionListener.html)
Interfaces

• In addition to (or instead of) abstract methods, an interface can contain constants

• When a class implements an interface, it gains access to all its constants

• A class that implements an interface can implement other methods as well
public interface Animal {

    public void eat();
    public void travel();
}

Credits: http://www.tutorialspoint.com/java/java_interfaces.htm
public class Mammal implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        Mammal m = new Mammal();
        m.eat();
        m.travel();
    }
}
Another Interface Example

- See Complexity.java
- See Question.java
- See MiniQuiz.java
public interface Complexity
{
    public void setComplexity (int complexity);
    public int getComplexity();
}
public class Question implements Complexity {
    private String question, answer;
    private int complexityLevel;

    // Constructor: Sets up the question with a default complexity.
    public Question (String query, String result) {
        question = query;
        answer = result;
        complexityLevel = 1;
    }
}

continue
public void setComplexity (int level) {
    complexityLevel = level;
}

public int getComplexity() {
    return complexityLevel;
}

public String getQuestion() {
    return question;
}
continue

 //-----------------------------------------------------------------
 //  Returns the answer to this question.
 //-----------------------------------------------------------------
 public String getAnswer()
 {
     return answer;
 }

 //-----------------------------------------------------------------
 //  Returns true if the candidate answer matches the answer.
 //-----------------------------------------------------------------
 public boolean answerCorrect (String candidateAnswer)
 {
     return answer.equals(candidateAnswer);
 }

 //-----------------------------------------------------------------
 //  Returns this question (and its answer) as a string.
 //-----------------------------------------------------------------
 public String toString()
 {
     return question + "\n" + answer;
 }

import java.util.Scanner;

public class MiniQuiz
{
   public static void main (String[] args)
   {
      Question q1, q2;
      String possible;

      Scanner scan = new Scanner (System.in);

      q1 = new Question ("What is the capital of Jamaica?", "Kingston");
      q1.setComplexity (4);

      q2 = new Question ("Which is worse, ignorance or apathy?", "I don't know and I don't care");
      q2.setComplexity (10);
   }
}
continue

System.out.print (q1.getQuestion());
System.out.println (" (Level: " + q1.getComplexity() + ")");
possible = scan.nextLine();
if (q1.answerCorrect(possible))
    System.out.println ("Correct");
else
    System.out.println ("No, the answer is " + q1.getAnswer());

System.out.println();
System.out.print (q2.getQuestion());
System.out.println (" (Level: " + q2.getComplexity() + ")");
possible = scan.nextLine();
if (q2.answerCorrect(possible))
    System.out.println ("Correct");
else
    System.out.println ("No, the answer is " + q2.getAnswer());
Sample Run

What is the capital of Jamaica? (Level: 4)
Kingston
Correct

Which is worse, ignorance or apathy? (Level: 10)
apathy
No, the answer is I don't know and I don't care
Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```java
class ManyThings implements interface1, interface2 {
    // all methods of both interfaces
}
```
Interfaces

• The Java API contains many helpful interfaces

• The \textit{Comparable} interface contains one abstract method called \texttt{compareTo}, which is used to compare two objects
Interfaces

• We discussed the `compareTo` method of the `String` class in Chapter 5

• The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order
The Comparable Interface

• Any class can implement Comparable to provide a mechanism for comparing objects of that type

```java
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```
The Comparable Interface

• The value returned from `compareTo` should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`.

• It's up to the programmer to determine what makes one object less than another.
public class MyRect implements Comparable {

    int width;
    int height;
    point position;

    public int getHeight(){
        return this.height;
    }

    @Override public int compareTo(Object otherRect){
        return this.height - (MyRect)otherRect.getHeight();
    }
}

Credits: http://stackoverflow.com/questions/13243622/comparable-java
The Iterator Interface

• As we discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time.
The Iterator Interface

• An iterator is created formally by implementing the `Iterator` interface, which contains three methods
  – The `hasNext` method returns a boolean result
    – true if there are items left to process
  – The `next` method returns the next object in the iteration
  – The `remove` method removes the object most recently returned by the `next` method
Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other
Passing Objects to Methods

• What a method does with a parameter may or may not have a permanent effect (outside the method)

• Note the difference between changing the internal state of an object versus changing which object a reference points to

• See ParameterTester.java
• See ParameterModifier.java
• See Num.java
```java
public class ParameterTester
{
    public static void main (String[] args)
    {
        ParameterModifier modifier = new ParameterModifier();

        int a1 = 111;
        Num a2 = new Num(222);
        Num a3 = new Num(333);
    }
}
```
continue

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    modifier.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

}
System.out.println("Before calling changeValues:");
System.out.println("a1  a2  a3");
System.out.println(a1 + "  " + a2 + "  " + a3 + 
"\n");
modifier.changeValues(a1, a2, a3);
System.out.println("After calling changeValues:");
System.out.println("a1  a2  a3");
System.out.println(a1 + "  " + a2 + "  " + a3 + 
"\n");

Output

Before calling changeValues:
a1  a2  a3
111  222  333

Before changing the values:
f1  f2  f3
111  222  333

After changing the values:
f1  f2  f3
999  888  777

After calling changeValues:
a1  a2  a3
111  888  333
public class ParameterModifier
{
    // Modifies the parameters, printing their values before and after making the changes.
    public void changeValues (int f1, Num f2, Num f3)
    {
        System.out.println ("Before changing the values:");
        System.out.println ("f1\t\n" + f1 + \t\" + f2 + \"\t\" + f3 + \"\n");

        f1 = 999;
        f2.setValue(888);
        f3 = new Num (777);

        System.out.println ("After changing the values:");
        System.out.println ("f1\t\n" + f1 + \t\" + f2 + \"\t\" + f3 + \"\n");
    }
}
public class Num
{
    private int value;

    // Sets up the new Num object, storing an initial value.
    public Num (int update)
    {
        value = update;
    }
    continue
continue

//-----------------------------------------------------------------
//  Sets the stored value to the newly specified value.
//-----------------------------------------------------------------
public void setValue (int update)
{
    value = update;
}

//-----------------------------------------------------------------
//  Returns the stored integer value as a string.
//-----------------------------------------------------------------
public String toString ()
{
    return value + "";
}
}
**STEP 1**
Before invoking `changeValues`

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>222</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
</tbody>
</table>
```

**STEP 2**

```
tester.changeValues (a1, a2, a3);
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>222</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
</tbody>
</table>
```

**STEP 3**

```
f1 = 999;
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>222</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**STEP 4**

```
f2.setValue (888);
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>888</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**STEP 5**

```
f3 = new Num (777);
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>888</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>999</td>
<td>777</td>
<td></td>
</tr>
</tbody>
</table>
```

**STEP 6**

```
After returning from changeValues
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>888</td>
<td>333</td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>999</td>
<td>777</td>
<td></td>
</tr>
</tbody>
</table>
Method Overloading

• Let's look at one more important method design issue: method overloading

• *Method overloading* is the process of giving a single method name multiple definitions in a class

• If a method is overloaded, the method name is not sufficient to determine which method is being called

• The *signature* of each overloaded method must be unique

• The signature includes the number, type, and order of the parameters
Method Overloading

• The compiler determines which method is being invoked by analyzing the parameters.

```java
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

```java
result = tryMe(25, 4.32)
```
Method Overloading

- The `println` method is overloaded:

  ```java
  println (String s)
  println (int i)
  println (double d)
  ```

  and so on...

- The following lines invoke different versions of the `println` method:

  ```java
  System.out.println ("The total is:");
  System.out.println (total);
  ```
Overloading Methods

• The return type of the method is **not** part of the signature

• That is, overloaded methods cannot differ only by their return type

• Constructors can be overloaded

• Overloaded constructors provide multiple ways to initialize a new object
Testing

• Testing can mean many different things
• It certainly includes running a completed program with various inputs
• It also includes any evaluation performed by human or computer to assess quality
• Some evaluations should occur before coding even begins
• The earlier we find an problem, the easier and cheaper it is to fix
Testing

• The goal of testing is to find errors
• As we find and fix errors, we raise our confidence that a program will perform as intended
• We can never really be sure that all errors have been eliminated
• So when do we stop testing?
  – Conceptual answer: Never
  – Cynical answer: When we run out of time
  – Better answer: When we are willing to risk that an undiscovered error still exists
Reviews

- A review is a meeting in which several people examine a design document or section of code.
- It is a common and effective form of human-based testing.
- Presenting a design or code to others:
  - makes us think more carefully about it
  - provides an outside perspective
- Reviews are sometimes called inspections or walkthroughs.
Test Cases

• A test case is a set of input and user actions, coupled with the expected results.

• Often test cases are organized formally into test suites which are stored and reused as needed.

• For medium and large systems, testing must be a carefully managed process.

• Many organizations have a separate Quality Assurance (QA) department to lead testing efforts.
Defect and Regression Testing

- Defect testing is the execution of test cases to uncover errors
- The act of fixing an error may introduce new errors
- After fixing a set of errors we should perform regression testing – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems
Black-Box Testing

• In black-box testing, test cases are developed without considering the internal logic

• They are based on the input and expected output

• Input can be organized into equivalence categories

• Two input values in the same equivalence category would produce similar results

• Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories
White-Box Testing

• *White-box testing* focuses on the internal structure of the code

• The goal is to ensure that every path through the code is tested

• Paths through the code are governed by any conditional or looping statements in a program

• A good testing effort will include both black-box and white-box tests
GUI Design

• We must remember that the goal of software is to help the user solve the problem.

• To that end, the GUI designer should:
  – Know the user
  – Prevent user errors
  – Optimize user abilities
  – Be consistent

• Let's discuss each of these in more detail.
Know the User

• Knowing the user implies an understanding of:
  – the user's true needs
  – the user's common activities
  – the user's level of expertise in the problem domain and in computer processing

• We should also realize these issues may differ for different users

• Remember, to the user, the interface is the program
Prevent User Errors

• Whenever possible, we should design user interfaces that minimize possible user mistakes

• We should choose the best GUI components for each task

• For example, in a situation where there are only a few valid options, using a menu or radio buttons would be better than an open text field

• Error messages should guide the user appropriately
Optimize User Abilities

• Not all users are alike – some may be more familiar with the system than others

• Knowledgeable users are sometimes called *power users*

• We should provide multiple ways to accomplish a task whenever reasonable
  - "wizards" to walk a user through a process
  - short cuts for power users

• Help facilities should be available but not intrusive
Be Consistent

• Consistency is important – users get used to things appearing and working in certain ways

• Colors should be used consistently to indicate similar types of information or processing

• Screen layout should be consistent from one part of a system to another

• For example, error messages should appear in consistent locations
Layout Managers

- A *layout manager* is an object that determines the way that components are arranged in a container.

- There are several predefined layout managers defined in the Java API:
  
  - Defined in the AWT:
    - Flow Layout
    - Border Layout
    - Card Layout
    - Grid Layout
    - GridBag Layout
  
  - Defined in Swing:
    - Box Layout
    - Overlay Layout
Layout Managers

• Every container has a default layout manager, but we can explicitly set the layout manager as well.

• Each layout manager has its own particular rules governing how the components will be arranged.
Layout Managers

• Some layout managers pay attention to a component's preferred size or alignment, while others do not

• A layout manager adjusts the layout as components are added and as containers are resized
Layout Managers

• We can use the `setLayout` method of a container to change its layout manager:

```java
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

• The following example uses a *tabbed pane*, a container which permits one of several panes to be selected

• See `LayoutDemo.java`
• See `IntroPanel.java`
import javax.swing.*;

public class LayoutDemo
{
  // Sets up a frame containing a tabbed pane. The panel on each
  // tab demonstrates a different layout manager.

  public static void main (String[] args)
  {
    JFrame frame = new JFrame ("Layout Manager Demo");
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
  
    continue
continue

JTabbedPane tp = new JTabbedPane();
tp.addTab ("Intro", new IntroPanel());
tp.addTab ("Flow", new FlowPanel());
tp.addTab ("Border", new BorderPanel());
tp.addTab ("Grid", new GridPanel());
tp.addTab ("Box", new BoxPanel());

frame.getContentPane().add(tp);
frame.pack();
frame.setVisible(true);
import java.awt.*;
import javax.swing.*;

class IntroPanel extends JPanel{
  setBackground (Color.green);
  JLabel l1 = new JLabel ("Layout Manager Demonstration");
  JLabel l2 = new JLabel ("Choose a tab to see an example of " + "a layout manager.");

  add (l1);
  add (l2);
}
import java.awt.;
import javax.swing.;

public class IntroPanel extends JPanel
{
    public IntroPanel()
    {
        setBackground (Color.green);

        JLabel l1 = new JLabel ("Layout Manager Demonstration");
        JLabel l2 = new JLabel ("Choose a tab to see an example of a layout manager.");

        add (l1);
        add (l2);
    }
}
Flow Layout

• *Flow layout* puts as many components as possible on a row, then moves to the next row

• Components are displayed in the order they are added to the container

• Each row of components is centered horizontally by default, but could also be aligned left or right

• The horizontal and vertical gaps between the components can be explicitly set

• See `FlowPanel.java`
import java.awt.*;
import javax.swing.*;

public class FlowPanel extends JPanel {

    // Sets up this panel with some buttons to show how flow layout affects their position.
    public FlowPanel () {
        super();
        setLayout (new FlowLayout());
        setForeground (Color.green);
    }

    public static void main(String[] args) {
        // Create a new FlowPanel instance
        FlowPanel panel = new FlowPanel();
        // Add the panel to the main window
        // (This code is not provided in the example)

        // Run the event loop
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                // Add panel to main window
                // (This code is not provided in the example)
            }
        });
    }
}

continue

    JButton b1 = new JButton ("BUTTON 1");
    JButton b2 = new JButton ("BUTTON 2");
    JButton b3 = new JButton ("BUTTON 3");
    JButton b4 = new JButton ("BUTTON 4");
    JButton b5 = new JButton ("BUTTON 5");

    add (b1);
    add (b2);
    add (b3);
    add (b4);
    add (b5);
}
```
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
```
Border Layout

- A *border layout* defines five areas into which components can be added
Border Layout

• Each area displays one component (which could be a container such as a JPanel)

• Each of the four outer areas enlarges as needed to accommodate the component added to it

• If nothing is added to the outer areas, they take up no space and other areas expand to fill the void

• The center area expands to fill space as needed

• See BorderPanel.java
import java.awt.*;
import javax.swing.*;

public class BorderPanel extends JPanel {
    public BorderPanel() {
        // Sets up this panel with a button in each area of a border layout to show how it affects their position, shape, and size.
        setLayout (new BorderLayout());
        setBackground (Color.green);
    }
}

// BorderPanel.java   Authors: Lewis/Loftus
//
// Represents the panel in the LayoutDemo program that demonstrates the border layout manager.

//********************************************************************
//  BorderPanel.java       Authors: Lewis/Loftus
//  Represents the panel in the LayoutDemo program that demonstrates the border layout manager.
//********************************************************************
continue

    JButton b1 = new JButton ("BUTTON 1");
    JButton b2 = new JButton ("BUTTON 2");
    JButton b3 = new JButton ("BUTTON 3");
    JButton b4 = new JButton ("BUTTON 4");
    JButton b5 = new JButton ("BUTTON 5");

    add (b1, BorderLayout.CENTER);
    add (b2, BorderLayout.NORTH);
    add (b3, BorderLayout.SOUTH);
    add (b4, BorderLayout.EAST);
    add (b5, BorderLayout.WEST);
}
continue

```java
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1, BorderLayout.CENTER);
add(b2, BorderLayout.NORTH);
add(b3, BorderLayout.SOUTH);
add(b4, BorderLayout.EAST);
add(b5, BorderLayout.WEST);
}
```
Grid Layout

• A grid layout presents a container’s components in a rectangular grid of rows and columns

• One component is placed in each cell of the grid, and all cells have the same size

• Components fill the grid from left-to-right and top-to-bottom (by default)

• The size of each cell is determined by the overall size of the container

• See GridPanel.java
import java.awt.*;
import javax.swing.*;

public class GridPanel extends JPanel
{
    public GridPanel()
    {
        setBackground (Color.green);
        setLayout (new GridLayout (2, 3));
    }
}

//********************************************************************
// GridPanel.java       Authors: Lewis/Loftus
//@
//@ Represents the panel in the LayoutDemo program that demonstrates
//@ the grid layout manager.
//@********************************************************************

public class GridPanel extends JPanel
{
    public GridPanel()
    {
        setBackground (Color.green);
        setLayout (new GridLayout (2, 3));
    }
}

//**********************************************************
//@ GridPanel.java       Authors: Lewis/Loftus
//@ Represents the panel in the LayoutDemo program that demonstrates
//@ the grid layout manager.
//@**********************************************************
continue

    JButton b1 = new JButton ("BUTTON 1");
    JButton b2 = new JButton ("BUTTON 2");
    JButton b3 = new JButton ("BUTTON 3");
    JButton b4 = new JButton ("BUTTON 4");
    JButton b5 = new JButton ("BUTTON 5");

    add (b1);
    add (b2);
    add (b3);
    add (b4);
    add (b5);

}
```java
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
```
Box Layout

• A box layout organizes components horizontally (in one row) or vertically (in one column)

• Components are placed top-to-bottom or left-to-right in the order in which they are added to the container

• By combining multiple containers using box layout, many different configurations can be created

• Multiple containers with box layouts are often preferred to one container that uses the more complicated gridbag layout manager
Box Layout

- *Invisible components* can be added to a box layout container to take up space between components
  - *Rigid areas* have a fixed size
  - *Glue* specifies where excess space should go

- Invisible components are created using these methods of the *Box* class:
  
  createRigidArea(Dimension d)
  createHorizontalGlue()
  createVerticalGlue()

- See *BoxPanel.java*
import java.awt.*;
import javax.swing.*;

public class BoxPanel extends JPanel
{
    // Sets up this panel with some buttons to show how a vertical
    // box layout (and invisible components) affects their position.
    public BoxPanel()
    {
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));

        setBackground (Color.green);
    }
}

.getSelectionModel
continue

```java
    JButton b1 = new JButton("BUTTON 1");
    JButton b2 = new JButton("BUTTON 2");
    JButton b3 = new JButton("BUTTON 3");
    JButton b4 = new JButton("BUTTON 4");
    JButton b5 = new JButton("BUTTON 5");

    add (b1);
    add (Box.createRigidArea (new Dimension (0, 10)));
    add (b2);
    add (Box.createVerticalGlue());
    add (b3);
    add (b4);
    add (Box.createRigidArea (new Dimension (0, 20)));
    add (b5);
```
continue

```java
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1);
add(Box.createRigidArea(new Dimension(0, 10)));
add(b2);
add(Box.createVerticalGlue());
add(b3);
add(b4);
add(Box.createRigidArea(new Dimension(0, 20)));
add(b5);
```