ACM Student Package

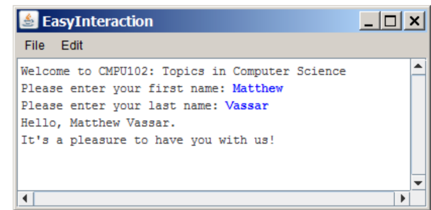
- Library of Java classes that simplify input, output and interaction with users.
- Intended for use by students learning Java programming.
- Available on class wiki.

EasyInteraction.java

```
import acm.program.*;

public class EasyInteraction extends ConsoleProgram {
   /* Displays a pop-up frame for program execution */
   public void run() {
      println("  Welcome to CMPU102: Data Structures and Java\n");
      String firstName = readLine("  Please enter your first name: ");
      String lastName = readLine("  Please enter your last name: \n");
      println("  Hello, " + firstName + " " + lastName + "." );
      println("  It's a pleasure to have you with us!");
   }

   /* Standard Java entry point */
   public static void main(String[] args) {
      /* Calling the start method of a ConsoleProgram (e.g., EasyInteraction)
       * invokes the system to call the run method. */
      EasyInteraction EI = new EasyInteraction();
      EI.setFont(new Font("sansserif", 0, 18));
      EI.start();
   }
}
```

The **main** method creates a new instance of **EasyInteraction** and invokes its **start** method. The **start** method calls the **run** method, which carries on with the program.

EasyInteraction.java



The `Stack` Class

- The `Stack` class is exported by the **`java.util`** package. It is a generic class that enforces LIFO (last-in, first-out) access to data. All operations are performed on the "top" element of the stack. The operations allowed on a Stack include:

  push: insert and return item on top of stack
  pop: remove and return item on top of stack
  peek: return item on top of stack
  isEmpty (or empty)

  Stacks are used in many applications and they are a central part of a programmer's toolkit.

  <u>Exercise:</u> Write a program that uses a Stack to match parentheses

The `Queue` ADT

- A queue is a data structure that enforces a FIFO (first-in, first-out) access to data. Elements are added at one end (usually called the "rear") and are removed from the other (usually called the "front"). Java has a Queue interface, but no Queue class.

  Queues are used in enough applications that they should be a familiar structure for all programmers. Operations that are allowed on a queue include:

  enqueue: add element at rear of queue
  dequeue: remove an element from front of queue
  isEmpty (or empty)

The `PriorityQueue` Class

- A priority queue is a data structure that isn't really a queue at all. It is implemented using a special type of binary tree known as a heap. Heaps are either min-heaps or max-heaps and are generally implemented using arrays.

  A min-heap is a binary tree that
  - is full at every level, but may be only left-filled at the bottom-most level such that
  - every child node contains a key that is >= the key at its parent node.
- A max-heap is similarly defined, except that every child node contains a key that is <= the key at its parent node
- Guarantees O(logN) operations on the tree. Used to implement the HashSort algorithm.

Implementing ADTs

- Both Stacks and Queues can be implemented with an ArrayList, using the ArrayList functions in controlled ways to enforce the requirements of these ADTs.
- For example, you could define a queue to store Strings by extending ArrayList<String>, then implement an enqueue method that adds elements to the end of the list, and a dequeue method that removes the item at position 0.

The `HashMap` Class

- The **`HashMap`** class is a generic class exported by the **`java.util`** package and is an implementation of the Dictionary ADT.
- The **`HashMap`** class implements the abstract idea of a **map (or dictionary)**, an associative relationship between *keys* and *values*. A **key** is an object that never appears more than once in a map and can therefore be used to identify a **value,** which is the object associated with a particular key.

The `HashMap` Class

- Although the **`HashMap`** class exports other methods as well, the essential operations on a **`HashMap`** are the ones listed in the following table:

map.**put**(*key*, *value*)	Sets the association for *key* in the map to *value*.
map.**get**(*key*)	Returns the value associated with *key*, or **null** if none.

Generic Types for Keys and Values

- A **`HashMap`** requires two type parameters in angle brackets: one for the key and one for the value.

  E.g., the type designation **`HashMap<String,Integer>`** indicates a **`HashMap`** that uses strings as keys to obtain integer values.

A Simple `HashMap` Application

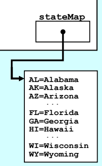
- Suppose that you want to write a program that displays the name of a state given its two-letter postal abbreviation.
- This program is an ideal application for the **`HashMap`** class because what you need is a map between two-letter codes and state names. Each two-letter code uniquely identifies a particular state and therefore serves as a key for the **`HashMap`**; the state names are the corresponding values.

A Simple `HashMap` Application

- To implement this program in Java, you need to perform the following steps, which are illustrated on the following slide:
  1. Create a **`HashMap`** containing all 50 key/value pairs.
  2. Read in the two-letter abbreviation to translate.
  3. Call **`get`** on the **`HashMap`** to find the state name.
  4. Print out the name of the state.

The `PostalLookup` Application

```
public void run() {
   HashMap<String,String> stateMap = new HashMap<String,String>();
   initStateMap(stateMap);
   while (true) {
      String code = readLine("Enter two-letter state abbreviation: ");
      if (code.length() == 0) break;
      String state = stateMap.get(code);
      if (state == null) {
         println(code + " is not a known state abbreviation");
      } else {
         println(code + " is " + state);
      }
   }
}
```



PostalLookup

```
Enter two-letter state abbreviation: HI
HI is Hawaii
Enter two-letter state abbreviation: WI
WI is Wisconsin
Enter two-letter state abbreviation: VE
VE is not a known state abbreviation
Enter two-letter state abbreviation:
```

skip simulation

The Idea of Hashing

- The goal of hashing is to do a search in O(1) time. To see how it works, it helps to think about how you find a word in a dictionary. You certainly don't start at the beginning and look at every word, but you probably don't use binary search either. Most dictionaries have thumb tabs that indicate where each letter first appears. Words starting with *A* are in the *A* section, and so on.
- The **`HashMap`** class uses a strategy called *hashing,* which is conceptually similar to the thumb tabs in a dictionary. The critical idea is that you can improve performance enormously if you use the key to figure out where to look.

Hash Codes

- To make it possible for the **`HashMap`** class to know where to look for a particular key, every object defines a method called **`hashCode`** that returns an integer (positive or negative) associated with that object. As you will see in a subsequent slide, this hash code value tells the **`HashMap`** implementation where it should look for a particular key.
- In general, clients of the **`HashMap`** class have no reason to know the actual value of the integer returned as a hash code for some key. The important things to remember are:
  1. Every object has a hash code, even if you don't know what it is.
  2. The hash code for any particular object is always the same.
  3. If two objects have equal values, they have the same hash code.

Hash Codes and Collisions

- For any Java object, the **`hashCode`** method returns an **`int`** that can be any one of the 4,294,967,296 (2^{32}) possible values for that type.
- While 4,294,967,296 seems huge, it is insignificant compared to the total number of objects that can be represented inside a machine, which would be infinite if there were no limits on the size of memory.
- The fact that there are more possible objects than hash codes means that there must be some distinct objects that have the same hash codes. For example, the strings **`"hierarch"`** and **`"crinolines"`** have the same hash code, which happens to be −1732884796.
- Because different keys can generate the same hash codes, any strategy for implementing a map using hash codes must take that possibility into account, even though it happens rarely.

The Bucket Hashing Strategy

- One common strategy for implementing a map is to use the hash code for an object to select an index into an array that will contain all the keys with that hash code. Each element of that array is conventionally called a **bucket**.
- In practice, the array of buckets is smaller than the number of hash codes, making it necessary to convert the hash code into a bucket index, typically by executing a statement like

```
int bucket = Math.abs(key.hashCode()) % N_BUCKETS;
```

- The value in each element of the bucket array cannot be a single key/value pair given the chance that different keys fall into the same bucket. Such situations are called **collisions**.
- To take account of the possibility of collisions, each element of the bucket array is usually a linked list of the keys that fall into that bucket, as shown in the simulation on the next slide.

Simulating Bucket Hashing

Suppose you call **`stateMap.get("HI")`**

`"HI".hashCode()` → 2305

`Math.abs`(2305) **`% 7`** → 2

The key **`"HI"`** must therefore be in bucket 2 and can be located by searching the chain.

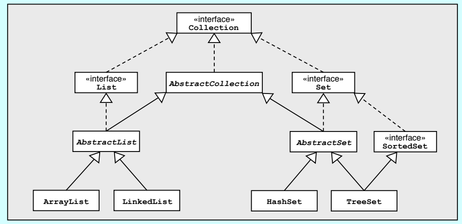
skip simulation

Achieving $O(1)$ Performance

- The simulation on the previous side uses only seven buckets to emphasize what happens when collisions occur: the smaller the number of buckets, the more likely collisions become.
- In practice, the real implementation of `HashMap` uses a much larger value for `N_BUCKETS` to minimize the opportunity for collisions. If the number of buckets is considerably larger than the number of keys, most of the bucket chains will either be empty or contain exactly one key/value pair.
- The ratio of the number of keys to the number of buckets is called the **load factor** of the `HashMap`. Because a `HashMap` achieves $O(1)$ performance only if the load factor is small, the library implementation of `HashMap` automatically increases the number of buckets when the table becomes too full.

The `Collection` Hierarchy

The following diagram shows the portion of the Java Collections Framework that implements the `Collection` interface. The dashed lines specify that a class implements a particular interface.



`ArrayList` *vs.* `LinkedList`

- If you look at the left side of the collections hierarchy on the preceding slide, you will discover that there are two classes in the Java Collections Framework that implement the `List` interface: `ArrayList` and `LinkedList`.
- Because these classes implement the same interface, it is generally possible to substitute one for the other.
- The fact that these classes have the same effect, however, does not imply that they have the same performance characteristics.
  - The `ArrayList` class is more efficient if you are selecting a particular element or searching for an element in a sorted array.
  - The `LinkedList` class is more efficient if you are adding or removing elements from a large list.
- Choosing which list implementation to use is therefore a matter of evaluating the performance tradeoffs.

Iteration in Collections

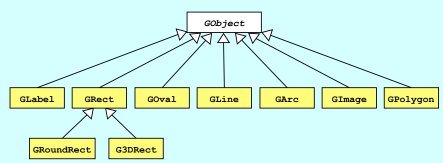
- One of the most useful operations for any collection is the ability to run through each of the elements in a loop. This process is called **iteration**.
- The `java.util` package includes a generic interface called `Iterator` that supports iteration over the elements of a collection. You can use a while loop to go through all the elements in a collection, or you can use the "for-each" loop.

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
   String element = iterator.next();
   . . . statements that process this particular element . . .
}
```

```
for (type element : collection) {
   . . . statements that process this particular element . . .
}
```

Using the Shape Classes

- The shape classes are the `GObject` subclasses that appear in yellow at the bottom of the hierarchy diagram.



- Each of the shape classes corresponds precisely to a method in the `Graphics` class in the `java.awt` package. Once you have learned to use the shape classes, you will easily be able to transfer that knowledge to Java's standard graphics tools.

The `GLabel` Class

You've been using the `GLabel` class ever since Chapter 2 and already know how to change the font and color, as shown in the most recent version of the "Hello World" program:

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world", 100, 75);
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      add(label);
   }
}
```

Centering Labels

The following update to the "Hello World" program centers the label in the window:

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() - label.getAscent()) / 2;
      add(label, x, y);
   }
}
```

The `GRect` Class

- The **`GRect`** class implements the **`GFillable`**, **`GResizable`**, and **`GScalable`** interfaces but does not otherwise extend the facilities of **`GObject`**.
- Like every other shape class, the **`GRect`** constructor comes in two forms. The first includes both the location and the size:

  `new GRect(`*x***`,`** *y***`,`** *width***`,`** *height***`)`**

  This form makes sense when you know in advance where the rectangle belongs.
- The second constructor defers setting the location:

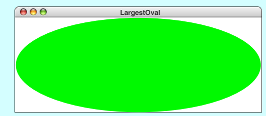
  `new GRect(`*width***`,`** *height***`)`**

  This form is more convenient when you want to create a rectangle and then decide where to put it later.

The `GOval` Class

- The **`GOval`** class represents an elliptical shape defined by the boundaries of its enclosing rectangle.
- As an example, the following **`run`** method creates the largest oval that fits within the canvas:

```
public void run() {
   GOval oval = new GOval(getWidth(), getHeight());
   oval.setFilled(true);
   oval.setColor(Color.GREEN);
   add(oval, 0, 0);
}
```



The `GLine` Class

- The **`GLine`** class represents a line segment that connects two points. The constructor call looks like this:

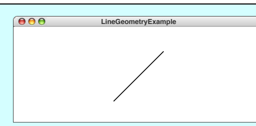
  `new GLine(`x_0**`,`** y_0**`,`** x_1**`,`** y_1**`)`**

  The points (x_0, y_0) and (x_1, y_1) are called the **start point** and the **end point**, respectively.
- The **`GLine`** class does not support filling or resizing but does implement the **`GScalable`** interface. When you scale a line, its start point remains fixed.
- Given a **`GLine`** object, you can get the coordinates of the two points by calling **`getStartPoint`** and **`getEndPoint`**. Both of these methods return a **`GPoint`** object.
- The **`GLine`** class also exports the methods **`setStartPoint`** and **`setEndPoint`**, which are illustrated on the next slide.

Setting Points in a `GLine`

The following **`run`** method illustrates the difference between the **`setLocation`** method (which moves both points together) and **`setStartPoint`**/**`setEndPoint`** (which move only one):

```
public void run() {
   GLine line = new GLine(0, 0, 100, 100);
   add(line);
   line.setLocation(200, 50);
   line.setStartPoint(200, 150);
   line.setEndPoint(300, 50);
}
```



The `GImage` Class

- The **`GImage`** class is used to display an image from a file. The constructor has the form

  `new GImage(`*image file***`,`** *x***`,`** *y***`)`**

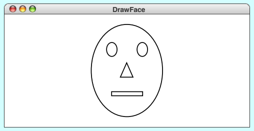
  where *image file* is the name of a file containing a stored image and *x* and *y* are the coordinates of the upper left corner of the image.
- When Java executes the constructor, it looks for the file in the current directory and then in a subdirectory named **`images`**.
- To make sure that your programs will run on a wide variety of platforms, it is best to use one of the two most common image formats: the Graphical Interchange Format (GIF) and the Joint Photographic Experts Group (JPEG) format. Typically, your image file name will end with the suffix **`.gif`** for GIF files and either **`.jpg`** or **`.jpeg`** for JPEG files.

Creating Compound Objects

- The **GCompound** class in the **acm.graphics** package makes it possible to combine several graphical objects so that the resulting structure behaves as a single **GObject**.
- The easiest way to think about the **GCompound** class is as a combination of a **GCanvas** and a **GObject**. A **GCompound** is like a **GCanvas** in that you can add objects to it, but it is also like a **GObject** in that you can add it to a canvas.
- As was true in the case of the **GPolygon** class, a **GCompound** object has its own coordinate system that is expressed relative to a **reference point**. When you add new objects to the **GCompound**, you use the local coordinate system based on the reference point. When you add the **GCompound** to the canvas as a whole, all you have to do is set the location of the reference point; the individual components will automatically appear in the right locations relative to that point.

Creating a Face Object

- The first example of the **GCompound** class is the **DrawFace** program, which is illustrated at the bottom of this slide.
- The figure consists of a **GOval** for the face and each of the eyes, a **GPolygon** for the nose, and a **GRect** for the mouth. These objects, however, are not added directly to the canvas but to a **GCompound** that represents the face as a whole.
- This primary advantage of using the **GCompound** strategy is that doing so allows you to manipulate the face as a unit.



The GFace Class

```
import acm.graphics.*;

/** Defines a compound GFace class */
public class GFace extends GCompound {

/** Creates a new GFace object with the specified dimensions */
   public GFace(double width, double height) {
      head = new GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
      mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
      add(head, 0, 0);
      add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
                   0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
                    0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0.50 * width, 0.50 * height);
      add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
                 0.75 * height - MOUTH_HEIGHT * height / 2);
   }
```

page 1 of 2 *skip code*

The GFace Class

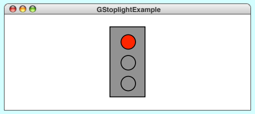
```
/* Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      poly.addVertex(0, -height / 2);
      poly.addVertex(width / 2, height / 2);
      poly.addVertex(-width / 2, height / 2);
      return poly;
   }

/* Constants specifying feature size as a fraction of the head size */
   private static final double EYE_WIDTH    = 0.15;
   private static final double EYE_HEIGHT   = 0.15;
   private static final double NOSE_WIDTH   = 0.15;
   private static final double NOSE_HEIGHT  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

/* Private instance variables */
   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

page 2 of 2 *skip code*

Specifying Behavior of a GCompound

- The **GCompound** class is useful for defining graphical objects that involve behavior beyond that common to all **GObject**s.
- The **GStoplight** on the next slide implements a stoplight object that exports methods to set and get which lamp is on. The following code illustrates its use:

```
public void run() {
   GStoplight stoplight = new GStoplight();
   add(stoplight, getWidth() / 2, getHeight() / 2);
   stoplight.setColor("RED");
}
```



The GStoplight Class

```
/**
 * Defines a GObject subclass that displays a stoplight.  The
 * state of the stoplight must be one of the Color values RED,
 * YELLOW, or GREEN.
 */
public class GStoplight extends GCompound {

/** Creates a new Stoplight object, which is initially GREEN */
   public GStoplight() {
      GRect frame = new GRect(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setFilled(true);
      frame.setFillColor(Color.GRAY);
      add(frame, -FRAME_WIDTH / 2, -FRAME_HEIGHT / 2);
      double dy = FRAME_HEIGHT / 4 + LAMP_RADIUS / 2;
      redLamp = createFilledCircle(0, -dy, LAMP_RADIUS);
      add(redLamp);
      yellowLamp = createFilledCircle(0, 0, LAMP_RADIUS);
      add(yellowLamp);
      greenLamp = createFilledCircle(0, dy, LAMP_RADIUS);
      add(greenLamp);
      setState(Color.GREEN);
   }
```

page 1 of 3 *skip code*

The **GStoplight** Class

```
/** Sets the state of the stoplight */
   public void setState(Color color) {
      if (color.equals(Color.RED)) {
         redLamp.setFillColor(Color.RED);
         yellowLamp.setFillColor(Color.GRAY);
         greenLamp.setFillColor(Color.GRAY);
      } else if (color.equals(Color.YELLOW)) {
         redLamp.setFillColor(Color.GRAY);
         yellowLamp.setFillColor(Color.YELLOW);
         greenLamp.setFillColor(Color.GRAY);
      } else if (color.equals(Color.GREEN)) {
         redLamp.setFillColor(Color.GRAY);
         yellowLamp.setFillColor(Color.GRAY);
         greenLamp.setFillColor(Color.GREEN);
      }
      state = color;
   }

/** Returns the current state of the stoplight */
   public Color getState() {
      return state;
   }
```

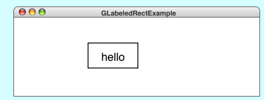

The **GStoplight** Class

```
/* Creates a filled circle centered at (x, y) with radius r */
   private GOval createFilledCircle(double x, double y, double r) {
      GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
      circle.setFilled(true);
      return circle;
   }

/* Private constants */
   private static final double FRAME_WIDTH = 50;
   private static final double FRAME_HEIGHT = 100;
   private static final double LAMP_RADIUS = 10;

/* Private instance variables */
   private Color state;
   private GOval redLamp;
   private GOval yellowLamp;
   private GOval greenLamp;
}
```


Exercise: Labeled Rectangles

Define a class **GLabeledRect** that consists of an outlined rectangle with a label centered inside. Your class should include constructors that are similar to those for **GRect** but include an extra argument for the label. It should also export **setLabel**, **getLabel**, and **setFont** methods. The following **run** method illustrates the use of the class:

```
public void run() {
   GLabeledRect rect = new GLabeledRect(100, 50, "hello");
   rect.setFont("SansSerif-18");
   add(rect, 150, 50);
}
```



Solution: The **GLabeledRect** Class

```
/** Defines a graphical object combining a rectangle and a label */
public class GLabeledRect extends GCompound {

/** Creates a new GLabeledRect object */
   public GLabeledRect(int width, int height, String text) {
      frame = new GRect(width, height);
      add(frame);
      label = new GLabel(text);
      add(label);
      recenterLabel();
   }

/** Creates a new GLabeledRect object at a given point */
   public GLabeledRect(int x, int y, int width, int height,
                       String text) {
      this(width, height, text);
      setLocation(x, y);
   }

/** Sets the label font */
   public void setFont(String font) {
      label.setFont(font);
      recenterLabel();
   }
```


Solution: The **GLabeledRect** Class

```
/** Sets the text of the label */
   public void setLabel(String text) {
      label.setLabel(text);
      recenterLabel();
   }

/** Gets the text of the label */
   public String getLabel() {
      return label.getLabel();
   }

/* Recenters the label in the window */
   private void recenterLabel() {
      double x = (frame.getWidth() - label.getWidth()) / 2;
      double y = (frame.getHeight() + label.getAscent()) / 2;
      label.setLocation(x, y);
   }

/* Private instance variables */
   private GRect frame;
   private GLabel label;
}
```