CSC 245: Objects and Data Abstraction

Chapter 2
Abstract Data Types & Java Classes

Outline

- Classes & Members
  - Defining a New Class & its Members
  - Constructors
  - Accessor Methods
- Using a Class
  - Creating & Using Objects
  - Object References
- Packages
- Parameters, Equals Methods, Clones

Objects

- Object Oriented Programming (OOP)
  - An approach in which data occurs in tidy packages called *Objects*.
  - Objects are manipulated using functions called *methods* which are part of their objects
- Class
  - Definition or template for objects (called instances)
    - Data members of an object.
    - Methods of an object.

Abstract Data Types

- Information Hiding
  - An example of the separation of specification from implementation
- Abstract Data Type (ADT)
  - Specification of a class's interface.
  - Class is an implementation of an ADT
- We will present throughout this chapter two examples of ADTs

Class Members

- A *class* is a new kind of data type
- A class includes
  - Data
    - Integers, characters, floats, etc.
  - Methods
    - Operations on objects.
  - Constructors
    - Initialize data of newly created objects.
  - View class as an ADT
    - Can specify which members are visible to the outside.
- Taken all together, the above elements constitute the class **members**

Example: Mechanical Throttle

- Description
  - A class that is used to store and manipulate the status of mechanical throttle
  - A throttle is a lever that can be moved to control fuel flow.
    - Similar to a gas pedal.
- Throttle Positions
  - Shutoff position
    - Allows no fuel flow.
  - *On* positions
    - Flow proportional to lever location.
    - Topmost position—*maximum* flow.
- Initialization
  - Number of positions
  - Throttle @ shutoff position.

Example: Mechanical Throttle…

- Constructor
  - Creates new throttle with:
    - One shutoff position &
    - A specified number of on positions.
- Methods
  - What is fuel flow?
    - Proportion of maximum flow.
  - Is throttle on?
    - Return true/false based on state of throttle.
  - Shift throttle by a given amount.
  - Move throttle back to shutoff position.

Example: Mechanical Throttle…

- Defining New Class

```
public class Throttle {
  private int top;    // topmost position of lever
  private int position;     // current position of
  lever

  // Implementations of constructors & methods
  // go here

}
```

  - Class header
  - Instance variables

Constructors

- Instance variables initialized to Java's default values.
  - 0 for numbers, `false` for booleans, *etc.*
- Declaration w/initialization overrides default values.

```
int step = 10;
```

- Constructor's name must be class's name.
- A constructor does not have a return value.

Example: Throttle Example…

- Constructor—Specification

`public Throttle(int size)`
Construct a `Throttle` with a specified number of positions.

  - **Parameters**
    `size`—number of on positions for this new `Throttle`.
  - **Precondition**
    `size > 0`
  - **Postcondition**
    This `Throttle` has been initialized with the specified number of on positions above the shutoff position, and is currently shut off.
  - **Throws:** `IllegalArgumentException`
    Indicates that `size` is not positive.

Example: Mechanical Throttle…

- Constructor—Implementation

```
public Throttle(int size)
{
  if (size <= 0)
    throw new
      IllegalArgumentException("Size <= 0: " + size);
  top = size;

  // No assignment needed for position.
  //   default value is zero.
}
```


No-Arguments Constructor

- Does not need information to initialize data members.
- Automatically created by Java.
  - When no other constructors defined!
- Can be overridden by implementer.

Methods

- Accessor
  - Gives info about object without altering it.
  - Also called *get*-methods or *getters*.
- Modifier
  - May change object's state.
- Methods implement operations on objects.
  - Inspect & modify object's data members.

Example: Mechanical Throttle…

- Accessor Method—getFlow

`public double getFlow()`
Get the current flow of this `Throttle`.

- **Returns**
  the current flow rate (always in the range [0.0 … 1.0]) as a proportion of the maximum flow.

```
public double getFlow()
{
  return (double) position / (double) top;
}
```


Example: Mechanical Throttle…

- Accessor Method—isOn

`public boolean isOn()`
Check whether this `Throttle` is on.

- **Returns**
  `true` if this `Throttle`'s flow is above zero. Otherwise, return `false`.

```
public boolean isOn()
{
  return (getFlow() > 0);
         // Equivalent to (position > 0)
}
```


Example: Mechanical Throttle…

- Modification Method—shutOff

`public void shutOff()`
Turn off this `Throttle`.

Postcondition:
This `Throttle`'s flow has been shut off.

```
public void shutOff()
{
  position = 0;
}
```

Example: Mechanical Throttle…

- Modification Method—shift

  public *void* shift(*int* amount)

  Move this Throttle's position up or down.

  - **Parameters**

    amount—amount to move position up or down (+ve for up, -ve for down)

  - **Postcondition**

    This Throttle's position has been moved by specified amount. Position always between zero & top position.

Example: Mechanical Throttle…

- Modification Method—shift…

```
public void shift(int amount)
{
  if (amount > top – position)
    // Adding amount puts position above top.
    position = top;
  else if (position + amount < 0)
    // Adding amount puts position below zero.
    position = 0;
  else
    // Adding amount puts position in range [0…top]
    position += amount;
}
```


Using a Class

- Creating Objects

  ```
  Throttle control;
  ```

  - control refers to an instance of class Throttle.
    - Initialized to null.
    - Cannot invoke any method on control yet.

  ```
  new Throttle(100);
  ```

  - Create a new Throttle object.
  - Instance variable top initialized to 100.

Using a Class

- Creating Objects…

  - ```
    Throttle control = new Throttle(100);
    ```
  - ```
    Throttle control;
    control = new Throttle(100);
    ```

  - Equivalent sets of statements
  - control refers to instance of Throttle.

Using a Class…

- Using Objects
  - `control.shift(3);`
  - `control.isOn();`
  - Invoke methods `shift` and `isOn` on object that `control` refers to.
- Method Call Components
  - Object reference (*e.g.*, `control`).
  - Field selector operator (`.`)
  - Method name (*e.g.*, `shift`)
  - Parameter list
    - May be empty.

Using a Class…

- Example

```
final int SIZE = 8;  // Size of the Throttle.
final int SPOT = 3;  // Target of Throttle's lever.

Throttle small = new Throttle(SIZE);

small.shift(SPOT);
System.out.print
    ("My small throttle is now at position");
System.out.println(SPOT + " out of " + SIZE + ".");
System.out.println("The flow is now: " +
                   small.getFlow());

My small throttle is now at position 3 out of 8.
The flow is now: 0.375.
```


Using a Class…

- Example—Multiple Instances of Same Class

```
Throttle tiny = new Throttle(4);
Throttle huge = new Throttle(10000);

tiny.shift(2);
huge.shift(2500);
```

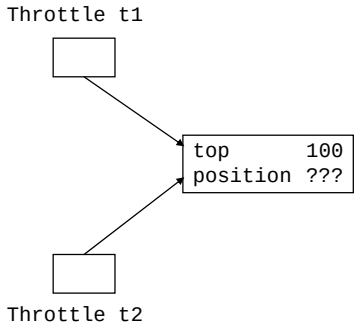
  - Objects `tiny` & `huge` are instances of the class `Throttle`.
    - Same methods.
    - Different copies of instance variables.

Reference Variable Assignment

- Code Example 1

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = t1;
t2.shift(-5);
```

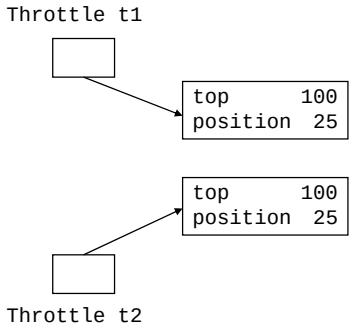


- Aliases
  - Refer to the same object.

Reference Variable Assignment…

- Code Example 2

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t1.shift(25);
t2 = new Throttle(100);
t2.shift(25);
```

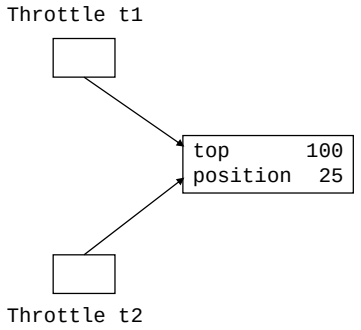


- `true` Expressions
  - `t1 != t2`
  - `t1.equals(t2)`

Equality Test

- `(t1 == t2)` is `true`

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t1.shift(25);
t2 = t1;
```

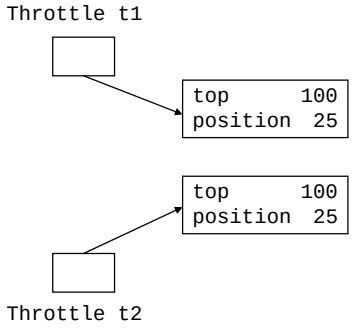


- Variables refer to SAME object.

Equality Test…

- `(t1 == t2)` is `false`

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t1.shift(25);
t2 = new Throttle(100);
t2.shift(25);
```



- Variables do NOT refer to SAME object.

Class `Location`—Specification

- Constructor
  `public` `Location(double xInitial, double yInitial)`
  - Constructs a `Location` with specified coordinates.
  - **Parameters**
    `xInitial`—the initial x coordinate of this `Location`.
    `yInitial`—the initial y coordinate of this `Location`.
  - **Postcondition**
    This `Location` has been initialized at the given coordinates.
- `clone` Method
  `public` `Object clone()`
  - Generate a copy of this `Location`.
  - **Returns**
    A copy of this `Location`. Changes to copy do not affect this `Location`.

Class `Location`—Specification…

- `distance` Method

  `public static` *`double`* `distance (Location p1, Location p2)`
  - Compute the distance between two `Locations`.
  - **Parameters**
    `p1`—the first `Location`.
    `p2`—the second `Location`.
  - **Returns**
    the distance between `p1` and `p2`.
  - **Note**
    The answer is `Double.POSITIVE_INFINITY` if the distance calculations overflows. The answer is `Double.NaN` if either `Location` is null.

Class `Location`—Specification…

- `equals` Method

  `public` *`boolean`* `equals(Object obj)`
  - Compare this `Location` with another object.
  - **Parameters**
    `obj`—an object with which this `Location` is compared.
  - **Returns**
    `true` if `obj` refers to a `Location` with same value. Otherwise, `false`.
  - **Note**
    The answer is false if `obj` is `null` or is not a `Location`.

Class `Location`—Specification…

- `midPoint` Method

  `public static` `Location midPoint (Location p1, Location p2)`
  - Generates & returns a `Location` halfway between two others.
  - **Parameters**
    `p1`—the first `Location`.
    `p2`—the second `Location`.
  - **Returns**
    a `Location` that is halfway between `p1` and `p2`.
  - **Note**
    The answer is `null` if `p1` or `p2` is `null`.

Class `Location`—Specification…

- `getX` & `getY` Methods

  `public` *`double`* `getX()` -and- **`public`** *`double`* `getY()`
  - Get the x or y coordinate of this Location.
  - **Returns**
    the x or y coordinate of this `Location`.

- `rotate90` Method

  `public` *`void`* `rotate90()`
  - Rotate this `Location` 90º in a clockwise direction.
  - **Postcondition**
    This `Location` has been rotated clockwise 90º around the origin.

Class `Location`—Specification…

- `shift` Method
  `public` *`void`* `shift(`*`double`* `xAmount,` *`double`* `yAmount)`
  - Move this `Location` by given amounts along x & y axes.
  - **Postcondition**
    This `Location` has been moved by given amounts along the two axes.
  - **Note**
    shift may cause a coordinate to go above `Double.MAX_VALUE` or below –`Double.MAX_VALUE`. Subsequent calls to accessor return `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`.

- `toString` **Method**
  `public` *`String`* `toString()`
  - Generate a string representation of this `Location`.
  - **Returns**
    - a string representation of this `Location`.

Location Class

```
package edu.colorado.geometry;

/*********************************************************************
 * A Location object keeps track of a location on a two-dimensional
 * plane.
 *********************************************************************/
public class Location implements Cloneable
{
    private double x; // The x coordinate of this Location
    private double y; // The y coordinate of this Location

    /*
     * Construct a Location with specified coordinates.
     * Parameters
     *  xInitial
     *    the initial x coordinate of this Location
     *  yInitial
     *    the initial y coordinate of this Location
     * Postcondition:
     *   This Location has been initialized at the given coordinates.
     */
    public Location(double xInitial, double yInitial)
    {
    x = xInitial;
    y = yInitial;
    }
```



```
/*
 * Generate a copy of this Location.
 * Parameters - none
 * Returns
 *   The return value is a copy of this Location. Subsequent
 *   changes to the copy  will not affect the original, nor vice versa.
 *   Note that the return value must be typecast to a
 *   Location before it can be used.
 */
public Object clone( )
{  // Clone a Location object.
Location answer;

try {
    answer = (Location) super.clone( );
}
catch (CloneNotSupportedException e) {
    // Exception should not occur. But if it does, it would
    // probably indicate a programming error that made
    // super.clone unavailable.  Most common error would be
    // forgetting the "Implements Cloneable" clause at the
    // start of this class.
    throw new RuntimeException
    ("This class does not implement Cloneable.");
}

return answer;
}
```



```
/*
 * Compute the distance between two Locations.
 * Parameters
 *  p1 - the first Location
 *  p2 - the second Location
 * Returns
 *   the distance between p1 and p2
 * Note
 *   The answer is Double.POSITIVE_INFINITY if the distance calculation
 *   overflows. The answer is Double.NaN if either Location is null.
 */
public static double distance(Location p1, Location p2)
{
double a, b, c_squared;

// Check whether one of the locations is null.
if ((p1 == null)  ||  (p2 == null))
    return Double.NaN;

// Calculate differences in x and y coordinates.
a = p1.x - p2.x;
b = p1.y - p2.y;

// Use Pythagorean Theorem to calculate the square of the distance.
// between the locations.
c_squared = a*a + b*b;

return Math.sqrt(c_squared);
}
```

```
/*
 * Compare this Location to another object for equality.
 * Parameters
 *  obj
 *    an object with which this Location will be compared
 * Returns
 *    A return value of true indicates that
 *    obj refers to a
 *    Location object with the same value as this
 *    Location. Otherwise the return value is
 *    false.
 * Note
 *    If obj is null or does not refer to a
 *    Location object, then the answer is false.
 */
public boolean equals(Object obj)
{
if (obj instanceof Location)
    {
    Location candidate = (Location) obj;
    return (candidate.x == x) && (candidate.y == y);
}
else
    return false;
}
```



```
/*
 * Get the x coordinate of this Location.
 * Parameters - none
 * Returns
 *    the x coordinate of this Location.
 */
public double getX( )
{
return x;
}

/*
 * Get the y coordinate of this Location.
 * Parameters - none
 * Returns
 *    the y coordinate of this Location.
 */
public double getY( )
{
return y;
}
```



```
/*
 * Generate and return a Location halfway between two others.
 * Parameters
 *  p1 - the first Location
 *  p2 - the second Location
 * Returns
 *    a Location that is halfway between p1
 *    and p2.
 * Note
 *    The answer is null if either p1 or p2 is null.
 */
public static Location midpoint(Location p1, Location p2)
{
double xMid, yMid;

//Check whether one of the locations is null.
if ((p1 == null)  ||  (p2 == null))
    return null;

// Compute the x and y midpoints.
xMid = (p1.x/2) + (p2.x/2);
yMid = (p1.y/2) + (p2.y/2);

// Create a new location and return it.
Location answer = new Location(xMid, yMid);
return answer;
}
```



```
/*
 * Rotate this Location 90 degrees in a clockwise direction.
 * Parameters
 *  - none
 * Postcondition
 *    This Location has been rotated clockwise 90 degrees around
 *    the origin.
 */
public void rotate90( )
{
double xNew;
double yNew;

// For a 90 degree clockwise rotations, the new x is the
// original y and the new y is -1 times the original x.
xNew = y;
yNew = -x;
x = xNew;
y = yNew;
}
```

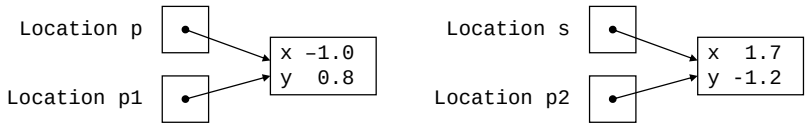
```
/*
 * Move this Location by given amounts along the x and y axes.
 * Parameters
 *  xAmount
 *    the amount to move this Location along the x axis
 *  yAmount
 *    the amount to move this Location along the y axis
 * Postcondition
 *    This Location has been moved by the given amounts along the
 *    two axes.
 * Note
 *    The shift may cause a coordinate to go above
 *    Double.MAX_VALUE or below -Double.MAX_VALUE.
 *    In these cases, subsequent activations of getX or
 *    getY will return Double.POSITIVE_INFINITY or
 *    Double.NEGATIVE_INFINITY.
 */
public void shift(double xAmount, double yAmount)
{
x += xAmount;
y += yAmount;
}
```



```
    /*
     * Generate a String representation of this Location.
     * Parameters
     *  - none
     * Returns
     *   a String representation of this Location
     */
     public String toString( )
     {
     return "(x=" + x + "  y=" + y + ")";
     }
}
```


Objects as Parameters

- Parameters
  - *Formal*—names of parameters as defined in method header.
  - *Actual*—names of parameters in method invocation.
- Formal parameters refer to same objects as actual parameters.
  - Changes to object in method are visible to invoking method.
- Example
  - `Location.midPoint(p, s);`
  - `public static Location midPoint(Location p1, Location p2)`



Objects as Return Values

- `midPoint` Method—Specification

  `public static` `Location midPoint(Location p1, Location p2)`

  Generates & returns a `Location` halfway between two others.
  - **Parameters**

    `p1`—the first location.

    `p2`—the second location.
  - **Returns**

    a `Location` halfway between two others.
  - **Note**

    The answer is `null` if either `Location` is null.

Objects as Return Values…

- `midPoint` Method—Implementation

```
public static Location midPoint(Location p1, Location p2)
{
  double xMid, yMid;

  // Check whether one of the Locations is null.
  if ((p1 == null) || (p2 == null))
    return null;

  // Compute the x & y midpoints.
  xMid = (p1.x / 2) + (p2.x / 2);
  yMid = (p1.y / 2) + (p2.y / 2);

  // Create a new Location & return it.
   return new Location(xMid, yMid);
}
```


Equals Method

== vs. Equals

- Operator "==" compares primitive types.
  - Object references are equal when they refer to the same object.
- Method Equals compares objects.
  - Instances of the same class are equal when their instance variables have the same values.

- `(p == s)` is true
  - `Location p = new Location(10,2);`
  - `Location s = p;`
- `(p != s) && p.equals(s)`
  - `Location p = new Location(10,2);`
  - `Location s = new Location(10,0);`
  - `s.shift(0,2);`

Equals Method—Implementation

- Template

```
public boolean
  equals(Object obj)
{
  if (obj is actually
      a Location)
  {
    // Compare contents of
  location
    // referred to by obj to this
    // location & return value.
  }
  else
    return false;
}
```

- Class `Location`

```
public boolean
  equals(Object obj)
{
  if (obj instanceOf
        Location)
  {
    Location candidate =
      (Location obj);
    return
    (candidate.x == x) &&
    (candidate.y == y);
  }
  else
    return false;
}
```


Clone Method

- Creates a copy of object.
- Returns reference to `Object`.
  - Must be typecast before used.
- Class must implement `Cloneable` interface.
  - `public class Location implements Cloneable`
- Should invoke clone method of superclass.
  - Needed for classes that are specialized.

Clone Method…

- Template

```
public Object clone() {
  Location ans;

  try {
    ans = (Location) super.clone();
  }
  catch (CloneNotSupportedException e) {
    throw new RuntimeException
      ("This class does not implement Cloneable");
  }

  // Make necessary changes.
  return ans;
}
```


Clone Method…

- Class `Location`

```
public Object clone() {
  Location answer;

  try {
    answer = (Location) super.clone();
  }
  catch (CloneNotSupportedException e) {
    // Exception should not occur.  "implements Cloneable" may be absent
    // from class header.
    throw new RuntimeException
      ("This class does not implement Cloneable");
  }

  return answer;
}
```


Class `Location`—Demo

- Description
  - Creates two locations
  - Rotates one twice 90º.
- Output

```
The still location is at:  (x=-2.0 y=-1.5)
The mobile location is at: (x=-2.0 y=-1.5)
Distance between them:  0.0
These two locations have equal coordinates.

I will rotate one location by two 90 degree turns.
The still location is at:  (x=-2.0 y=-1.5)
The mobile location is at: (x=2.0 y=1.5)
Distance between them:  5.0
These two locations have different coordinates.
```


Class `Location`—Demo…

```
import edu.colorado.geometry.Location;
class LocationDemonstration
{
   public static void main(String[ ] args)
   {
      final double STILL_X = -2.0;
      final double STILL_Y = -1.5;
      final int ROTATIONS = 2;
      Location still  = new Location(STILL_X, STILL_Y);
      Location mobile = (Location) still.clone( );
      printData(still, mobile);
      System.out.println("I will rotate one location by two 90 degree turns.");
      specifiedRotation(mobile, ROTATIONS);
      printData(still, mobile);
   }
   // Other methods...
}
```

Class `Location`—Demo…

```
// Rotate a Location p by a specified number of
// 90 degree clockwise turns.
public static void specifiedRotation(Location p, int n)
{
  while (n > 0)
  {
      p.rotate90( );
      n--;
  }
}
```


Class `Location`—Demo…

```
// Print some information about two locations:
// s (a "still" location) and m (a "mobile" location).

public static void printData(Location s, Location m)
{
 System.out.println ("The still location is at: " + s.toString( ));
 System.out.println ("The mobile location is at: " + m.toString( ));
 System.out.println ("Distance between them: " + Location.distance(s, m));
 if (s.equals(m))
   System.out.println ("These two locations have equal coordinates.");
 else
   System.out.println("These two locations have different coordinates.");
 System.out.println( );
}
```