

CS101

# Problem Solving and Object-Oriented Programming

## L16: Classes, Declarations, Scope

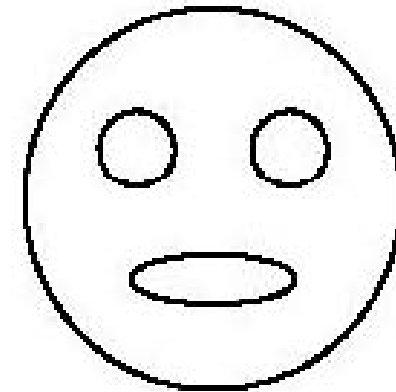


# Creating our own classes

Consider making a funny face:

- Physical Characteristics:

- Head
- Mouth
- Two Eyes

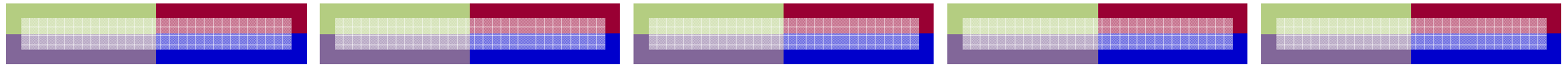


- Behaviors

- Can check contains
- Movable (by dragging)

- A FunnyFace class allows us to create new objects and lets us extend WindowController in a straight-forward way.





# A Draggable Face

```
public class RevFaceDrag extends WindowController {
    :
    :

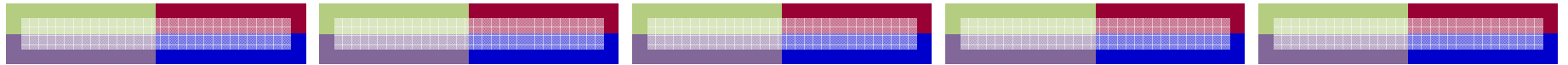
    private FunnyFace happy;           // FunnyFace to be dragged
    private Location lastPoint;

    private boolean happyGrabbed = false; // Whether happy has been grabbed by the mouse

    public void begin() { // Make the FunnyFace
        happy = new FunnyFace( FACE_LEFT, FACE_TOP, canvas );
    }
    public void onMousePress( Location point ){
        lastPoint = point;
        happyGrabbed = happy.contains( point );
    }
    public void onMouseDrag( Location point ) {
        if (happyGrabbed ) {
            happy.move( point.getX() - lastPoint.getX(),
                       point.getY() - lastPoint.getY() );
            lastPoint = point;
        }
    }
}
```

Looks just like our  
box dragging code.





# Defining a Class

- What do we want objects created from that class to do?
  - Example:
  - A train is a collection of shapes consisting of a car, wheels and a smokestack.
  - A train can move on the track. The train can produce a puff of smoke.





# Defining a Class

● What do we want objects created from that class to do?

● Example:

Data

● A train is a **collection of shapes** consisting of a **car**, **wheels** and a **smokestack**.

● A train can **move on the track**. The train can **produce a puff of smoke**.

Methods





# The Train Class

```
public class Train{
```

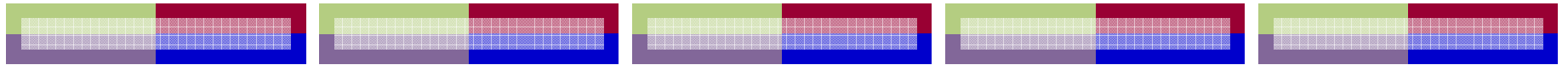
```
    private FilledRect car;  
    private FilledOval rearWheel;  
    private FilledOval frontWheel;  
    private FramedRect smokeStack;
```

Data

```
    public Train (...){  
    }  
    public void move (...){  
    ...  
    }  
    public boolean contains (Location point){  
    ...  
    }  
    public FilledOval produceSmoke (...){  
    ...  
    }  
}
```

Methods (to be defined)





# Constructing a Train

- Recall that constructor methods
  - create an instance of an object
  - Have the same name as the object
  - We have created new objects such as FilledOvals, etc
    - New FilledOval(LEFT, TOP, WIDTH, HEIGHT, canvas);
  - We've written our own constructors in the stick figures lab

```
public StickFigure (DrawingCanvas figureCanvas) {  
    // Draw the head  
    new FramedOval (HEAD_LEFT, HEAD_TOP, HEAD_SIZE, HEAD_SIZE,figureCanvas);  
    // Draw the body  
    new Line (BODY_LEFT, BODY_TOP, BODY_LEFT, BODY_TOP + BODY_SIZE,  
    figureCanvas);  
    ...  
}
```





# The Train Class

```
public class Train{
```

```
    private FilledRect car;  
    private FilledOval rearWheel;  
    private FilledOval frontWheel;  
    private FramedRect smokeStack;
```

Data

```
    public Train (...){  
    }
```

```
    public void move (...){
```

```
        ...
```

```
    }
```

```
    public boolean contains (Location point){
```

```
        ...
```

```
    }
```

```
    public FilledOval produceSmoke (...){
```

```
        ...
```

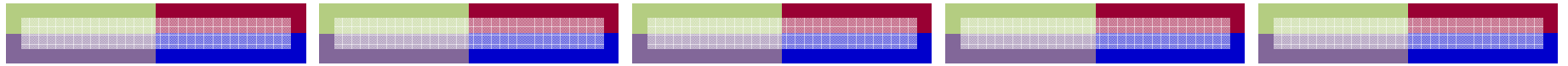
```
    }
```

```
}
```



Constructor



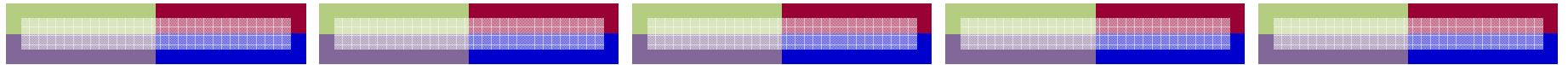


# Constructing a Train

- Initialize the instance variables
- Get it on the display

```
public class Train {  
    private FilledRect car;  
    private FilledOval rearWheel;  
    private FilledOval frontWheel;  
    private FramedRect smokeStack;  
  
    public Train (...) {  
        car = new FilledRect (...);  
        rearWheel = new FilledOval (...);  
        frontWheel = new FilledOval (...);  
        smokeStack = new FramedRect (...);  
    }  
    ...  
}
```





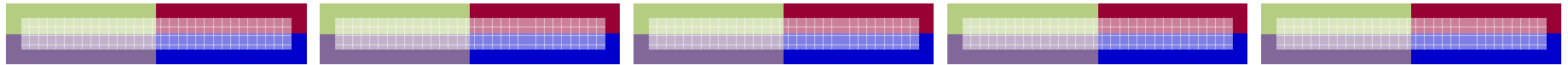
# Train Size

## ● Let Train determine it's own size

- Define constants within Train class

```
public class Train {  
    private static final int CAR_WIDTH = ...;  
    private static final int CAR_HEIGHT = ...;  
    private static final int WHEEL_DIAMETER = ...;  
    private static final int SMOKESTACK_HEIGHT = ...;  
    private static final int SMOKESTACK_WIDTH = ...;  
    ...  
    public Train (...) {  
        car = new FilledRect (... , CAR_WIDTH, CAR_HEIGHT, ...);  
        rearWheel = new FilledOval (... , WHEEL_DIAMETER, WHEEL_DIAMETER, ...);  
        frontWheel = new FilledOval (... , WHEEL_DIAMETER, WHEEL_DIAMETER, ...);  
        smokeStack = new FilledOval (... SMOKESTACK_WIDTH SMOKESTACK_HEIGHT, ...);  
    }  
    ...  
}
```



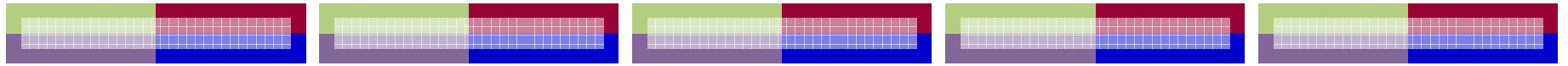


# Train Location

- Suppose we want to let the method that calls “new Train” decide where to place the train
  - Add parameters to the constructor

```
public Train(double left, double top, DrawingCanvas trainCanvas) {  
    car = new FilledRect (left, top, CAR_WIDTH, CAR_HEIGHT, trainCanvas);  
    rearWheel = new FilledOval (left + ..., top + ..., WHEEL_DIAMETER,  
        WHEEL_DIAMETER, trainCanvas);  
    frontWheel = new FilledOval (left + ..., top + ..., WHEEL_DIAMETER,  
        WHEEL_DIAMETER, trainCanvas);  
    smokeStack = new FilledOval (left + ..., top - ..., SMOKESTACK_WIDTH,  
        SMOKESTACK_HEIGHT, trainCanvas);  
}
```



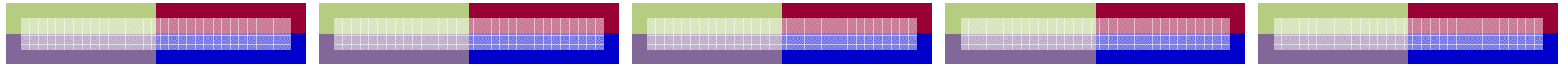


# Moving a train

- How do we move it?
  - Move it's pieces
  - This technique is called **delegation**

```
public void move (...){  
    car.move(...);  
    rearWheel.move (...);  
    frontWheel.move (...);  
    smokeStack.move (...);  
}
```



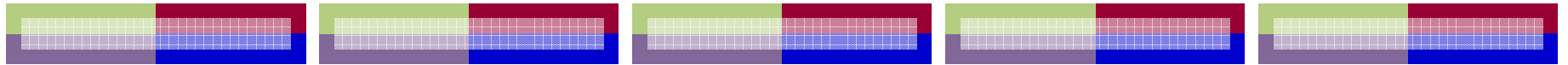


# Moving a Train

- How do we know where to move it to?
  - Model after things like FramedOval
  - Let the caller tell the train how far to move
  - Add parameters to the move method
  - All the parts move the same amount

```
public void move (double dx, double dy) {  
    car.move (dx, dy);  
    rearWheel.move (dx, dy);  
    frontWheel.move (dx, dy);  
    smokeStack.move (dx, dy);  
}
```

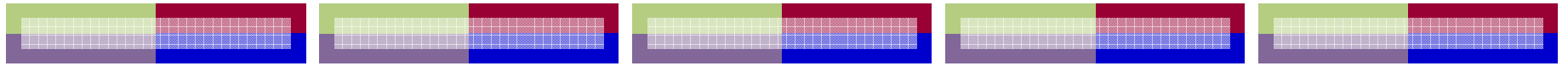




# The Train Class

```
public class Train {  
    ...  
    public Train (int left, int top, DrawingCanvas trainCanvas)  
    {  
    }  
    public void move (int dx, int dy) {  
    ...  
    }  
    public boolean contains (Location point) {  
    ...  
    }  
    public void produceSmoke () {  
    ...  
    }  
}
```

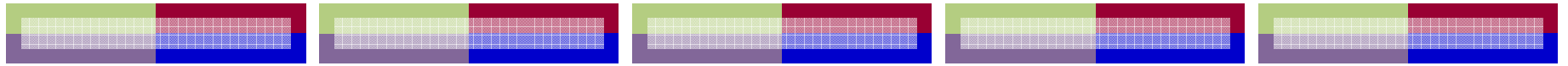




# TrainController

```
public class TrainController extends WindowController {  
  
    private Train train;  
  
    public void begin() {  
        train = new Train (10, 150, canvas);  
    }  
}
```

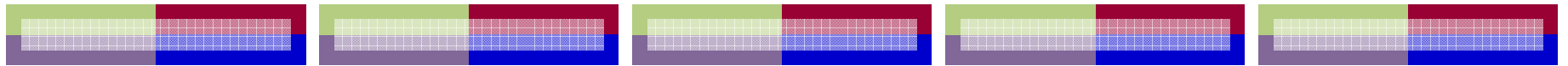




## Ignoring the User!

- What if we want to limit what the user can do?
- For example, don't allow the user to drag the train off the track
- How do we change the dragging code to do that?
- How do we change the move method to only move horizontally?





# Staying on track!

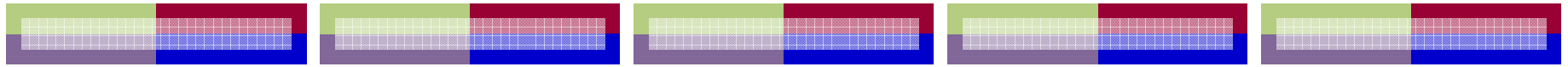
## Normal drag moves in 2 dimensions:

```
public void onMouseDrag (Location point) {  
    if (trainGrabbed) {  
        double dx = point.getX() - lastPoint.getX();  
        double dy = point.getY() - lastPoint.getY();  
        train.move (dx, dy);  
        lastPoint = point;  
    }  
}
```

## Staying on track ignores vertical motion:

```
public void onMouseDrag (Location point) {  
    if (trainGrabbed) {  
        double dx = point.getX() - lastPoint.getX();  
        train.move (dx);  
        lastPoint = point;  
    }  
}
```





# Staying on track!

**Normal drag moves in 2 dimensions:**

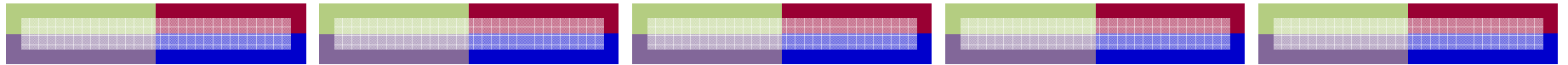
```
public void move (double dx, double dy) {  
    car.move (dx, dy);  
    rearWheel.move (dx, dy);  
    frontWheel.move (dx, dy);  
    smokeStack.move (dx, dy);  
}
```

---

**Staying on track moves 0 pixels vertical ly:**

```
public void move (double dx) {  
    car.move (dx, 0);  
    rearWheel.move (dx, 0);  
    frontWheel.move (dx, 0);  
    smokeStack.move (dx, 0);  
}
```





# Scope

- Scope determines the part of a program that knows about a declaration.
- Depends on:
  - Where the declaration is:
    - Instance variable
    - Parameter
    - Local variable
  - Use of words “public” and “private”





# Instance Variables

- Declared inside a class, but outside all methods
- Should always be private
- Scope: the entire class
- Should be given a value either where they are declared or in the constructor
- Key differentiator:
  - They remember their value between method calls

```
public class Train {  
    private FilledRect car;  
  
    public Train (...) {  
        car = new FilledRect (...);  
        ...  
    }  
  
    public void move (int dx, int dy){  
        car.move (dx, dy);  
        ...  
    }  
}
```





# Parameters


## ● Parameter

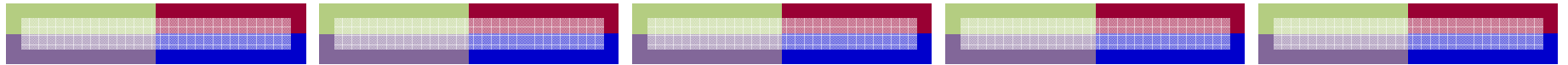
- Declared in the signature of a method or constructor
- Say neither public nor private
- Scope: the entire method or constructor
- Key differentiator: They are given a value based on an argument passed in when the method is called

```
public Train (int left, int trackHeight, DrawingCanvas canvas) {  
    car = new FilledRect (left, trackHeight - ..., ..., ..., canvas);  
    ...  
}
```

Somewhere else:

```
new Train (10, 50, canvas);  
new Train (100, 30, canvas);
```





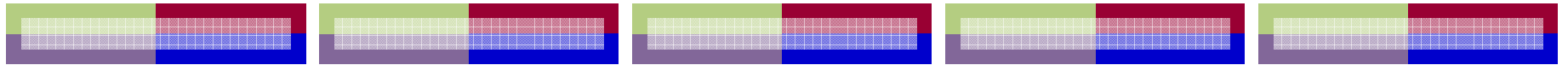
# Local variables

## ● Local variable

- Declared inside a method or constructor
- Say neither public nor private
- Scope: (part of) that method or constructor
- They are given a value with an assignment statement inside the method or constructor
- Go out of existence when the block in which they are declared ends
- Key differentiator: Good for temporary computations

```
public void onMouseDrag (Location point) {  
    Line line = new Line (start, point, canvas);  
    line.setColor (Color.RED);  
}
```





# Blocks

```
public void onMouseDrag (Location point) {  
    if (hiderGrabbed) {  
        double dx = point.getX() - lastPoint.getX();  
        double dy = point.getY() - lastPoint.getY();  
        hider.move (dx, dy);  
        lastPoint = point;  
    }  
}
```

scope of dy

A local variable's scope is from the point at which it is declared to the end of the enclosing block.





# Reusing Names within a Class

```
public class SomeController extends WindowController {  
    public void onMousePress (Location point) {  
        double x = point.getX();  
        ...  
    }  
  
    public void onMouseRelease (Location point) {  
        double y = point.getY();  
        ...  
    }  
}
```

Scopes do not overlap. No problem





# Reusing Names within a Class

```
public class SomeController extends WindowController {  
    private Location point;  
  
    public void onMousePress (Location point) {  
        double x = point.getX();  
        ...  
    }  
  
    public void begin () {  
        point = new Location (30, 50);  
        ...  
    }  
}
```

Scopes overlap. Inner declaration hides outer one.





# Reusing Names within a Class

```
public class SomeController extends WindowController {  
    private Location point;  
  
    public void onMousePress (Location point) {  
        this.point = point;  
        double x = point.getX();  
        ...  
    }  
}
```

“this.” always gives access to the instance variable





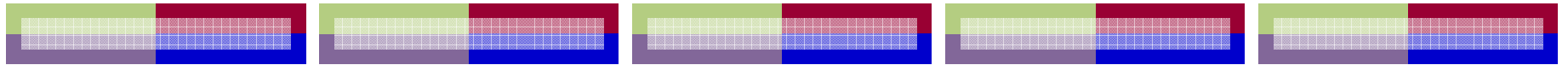
# Reusing Names in 2 Classes

```
public class SomeController extends WindowController {
    public void begin () {
        SmileyFace face = new SmileyFace (canvas);
        ...
    }
}

public class SmileyFace {
    public SmileyFace (DrawingCanvas canvas) {
        FramedOval face = new FramedOval(..., canvas);
        ...
    }
}
```

2 distinct variables that have distinct values



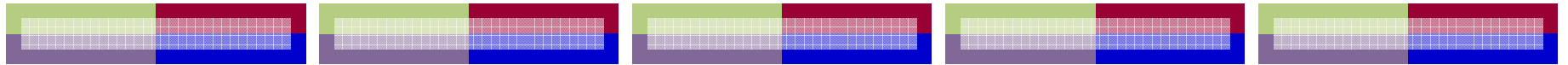


# Reusing Names in 2 Classes

```
public class SomeController extends WindowController {  
    public void begin () {  
        FilledRect grass = new FilledRect (... , canvas);  
        Train train = new Train (canvas);  
        ...  
    }  
}  
  
public class Train {  
    public Train (DrawingCanvas canvas) {  
        FilledRect car = new FilledRect(..., canvas);  
        ...  
    }  
}
```

2 distinct variables that will have the same value at runtime





# Summary

- Classes give us a way to treat a collection of objects as a single object
- When we define a class, we are defining an abstraction - a way of stepping back from implementation details to define a new thing
- A train is an abstraction of multiple shapes. We can use it thinking of it as its own type of thing, with its own interesting behavior, like move
- Abstraction is essential to build complex programs (or other things!)

