

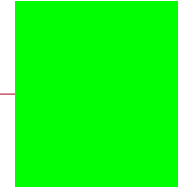
Objects

CS110: Introduction to Computer Science



Drawing

- To move an object what do we need?
 - X,Y, and ID
- How do we keep track of this?
 - **Let's do it right now**
- **Why is this problematic?**



Objects and Classes

- What we want is a way to create a "Rectangle" type that contains all the variables
- Moreover, we'd like it if we could tell objects of this type to perform actions, like "move"
- There are two parts to making such a type
 - Creating a **class**
 - Creating an **object**.



Classes

- A **class**, short for classification, is where we specify all of the variables and functions that we want to be associated with each other.
 - Classes are like blueprints.
 - They sketch out how the Object will work
- Think of classes as though they are real "classification of objects." This will help you when building



Class Syntax

```
public class Rectangle {  
    private int x;  
    private int y;  
    private int ID;  
  
    public Rectangle(int setX, int setY){  
        x = setX;  
        y = setY;  
        ID = Magic.drawRectangle(setX,setY,20, 20, "red");  
    }  
    public void moveBy(int deltaX, int deltaY){  
        x+=deltaX;  
        y+=deltaY;  
  
        Magic.moveObject(x,y,ID);  
    }  
}
```



Class Syntax

```
public class Rectangle {
```

```
    private int x;  
    private int y;  
    private int ID;
```

Member
Variables

```
    public Rectangle(int setX, int setY){  
        x = setX;  
        y = setY;  
        ID = Magic.drawRectangle(setX,setY,20, 20, "red");  
    }
```

```
    public void moveBy(int deltaX, int deltaY){  
        x+=deltaX;  
        y+=deltaY;
```

```
        Magic.moveObject(x,y,ID);
```

```
    }
```

```
}
```



Class Syntax

```
public class Rectangle {  
    private int x;  
    private int y;  
    private int ID;  
  
    public Rectangle(int setX, int setY){  
        x = setX;  
        y = setY;  
        ID = Magic.drawRectangle(setX,setY,20, 20, "red");  
    }  
    public void moveBy(int deltaX, int deltaY){  
        x+=deltaX;  
        y+=deltaY;  
  
        Magic.moveObject(x,y,ID);  
    }  
}
```

Methods



Class Syntax

```
public class Rectangle {  
    private int x;  
    private int y;  
    private int ID;  
  
    public Rectangle(int setX, int setY){  
        x = setX;  
        y = setY;  
        ID = Magic.drawRectangle(setX,setY,20, 20, "red");  
    }  
  
    public void moveBy(int deltaX, int deltaY){  
        x+=deltaX;  
        y+=deltaY;  
  
        Magic.moveObject(x,y,ID);  
    }  
}
```

Constructor



Member Variables

```
private int x;  
private int y;  
private int ID;
```

- Member Variables are the **properties** or **fields** associated with the class.
- **Every Function** has access to these variables
 - Differs from **local variables**, what we've done so far.
 - Local variables exist only in one functions
- The word **private** means that only methods in this class can access it.



Instance Functions

```
public void moveBy(int deltaX, int deltaY){  
    x+=deltaX;  
    y+=deltaY;  
    Magic.moveObject(x,y,ID);  
}
```

- Instance functions are the "verbs" associated with the class.
- They have access to **all** the member variables.
- The word **public** means that methods outside of this class can access it.
- There is **no static keyword**.
 - We'll talk about **static** in a bit.



Constructor

```
public Rectangle(int setX, int setY){  
    x = setX;  
    y = setY;  
    ID = Magic.drawRectangle(setX, setY, 20, 20, "red");  
}
```

- The constructor is a "function" that is the first thing called when the **object** of this class is made.
- It **has the same name** as the class.
- It **does not return any value**.
- ***You should initialize all member variables in the class***



Object

- Classes are concepts like "Student"
- Objects are things like "Omer" or "Paige"
- Classes describe the functions, objects use the functions



Object Syntax

```
Rectangle rect1;  
rect1 = new Rectangle(40,50);  
  
Rectangle rect2 = new Rectangle(40,50);  
  
rect1.moveBy(20,30);  
rect2.moveBy(10,10);
```



Object Syntax

```
Rectangle rect1;  
rect1 = new Rectangle(40,50);
```

```
Rectangle rect2 = new Rectangle(40,50);  
  
rect1.moveBy(20,30);  
rect2.moveBy(10,10);
```

**Making the
Object**



Object Syntax

```
Rectangle rect1;  
rect1 = new Rectangle(40,50);
```

```
Rectangle rect2 = new Rectangle(40,50);
```

```
rect1.moveBy(20,30);  
rect2.moveBy(10,10);
```

**Constructor
Shortcut**



Object Syntax

```
Rectangle rect1;  
rect1 = new Rectangle(40,50);  
  
Rectangle rect2 = new Rectangle(40,50);  
  
rect1.moveBy(20,30);  
rect2.moveBy(10,10);
```

Using
functions



Making an Object

```
Rectangle rect1;  
rect1 = new Rectangle(40,50);
```

- Three parts to making an object
 1. Declaration
 2. Construction
 3. Initialization



Making an Object

```
Rectangle rect1;
```

-
- **Declaration** is where the type of the object is "declared" but the object is not made



Making an Object

```
rect1 = new Rectangle(40, 50);
```

- **Construction** is where the memory for the object is allocated
 - **Note:** It's allocated in RAM. Not important for now. But good to know



Making an Object

```
rect1 = new Rectangle(40,50);
```

-
- **Initialization** is where all of the member variables for the object are specified.
 - This is when the **constructor** you created is run.



Keyword Fun: **public/private**

- **public** and **private** can be applied to both member variables and instance functions.
- **Generally**, all *member variables* should be **private** and all *instance functions* should be **public**.
- **Let's see public and private in action.**



Keyword Fun: **static**

- **static** is not widely used in Java.
 - We've used it as a cheat but we shouldn't really use it any more.
- **static** denotes a function or variable is a **class function** or **class variable**.
- Unlike **instance functions** or **member variables**,
 - you do not need to create an object to use them
 - There is only **one** "instance" of each class variable.
- **Let's see this in action**



Keyword Fun: **this**

- Why can't we just call `setX` and `setY` the names "x" and "y"

```
public Rectangle(int setX, int setY){  
    x = setX;  
    y = setY;  
    ID = Magic.drawRectangle(setX, setY, 20, 20, "red");  
}
```

- Because, it creates a scope conflict.
- The keyword **this** can be used to resolve this.



Keyword Fun: **this**

- The keyword **this** is used to refer to member variables in *this* current object

```
public Rectangle(int setX, int setY){  
    x = setX;  
    y = setY;  
    ID = Magic.drawRectangle(setX,setY,20, 20, "red");  
}
```

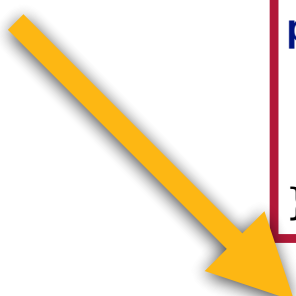
```
public Rectangle(int x, int y){  
    this.x = x;  
    this.y = y;  
    ID = Magic.drawRectangle(x,y,20, 20, "red");  
}
```



Keyword Fun: **this**

Set's the member variable "x" used to refer to member variables in *this* to be the parameter "x"

```
public Rectangle(int setX, int setY){  
    x = setX;  
    y = setY;  
    ID = Magic.drawRectangle(setX, setY, 20, 20, "red");  
}
```



```
public Rectangle(int x, int y){  
    this.x = x;  
    this.y = y;  
    ID = Magic.drawRectangle(x, y, 20, 20, "red");  
}
```



TL;DR

- Member variables: `private` or `protected`
- Instance Methods: `public`
- Don't use `static`
- The keyword `this` can force reference to *this* object's member variables.



Common Functions

- There are a functions that we frequently see and they are generally worth making
 - Getters
 - Setters
 - `toString()`



Getters

- Since member variables are generally private, what if we need to "get" their value?
- Well, we make a **getter**, a function that returns a member variable value.
- Always called `getMemberVarName()`
- Return type is same as variable
- Takes no parameters.

```
private int x;  
  
public int getX(){  
    return x;  
}
```



Setters

- What if we need to "set" a member variable?
- Well, we make a **setter**, a function that modifies a member variable.
- Always called `setMemberVarName()`
- Return type is `void`
- Takes one parameters, the same type as the variable.

```
private int x;  
  
public void setX(int x){  
    this.x = x;  
  
    Magic.moveObject(x, y, ID);  
}
```



Why Setters and Getters?

1. **Protection:** some values might be invalid and we don't want to allow that
 - e.g., Preventing setting denominator of fraction to 0
2. **Additional actions:** we might need to perform additional steps when setting a value
 - e.g., The code to the right.

```
private int x;  
  
public void setX(int x){  
    this.x = x;  
  
    Magic.moveObject(x, y, ID);  
}
```



toString()

- Sometimes it's helpful to convert an object to a string.
- Hence, toString().
- If we implement it in a given class, then we can use string concatenation with objects of that type

```
public String toString() {  
    return "(" + x + "," + y + ")";  
}
```

```
public static void main(String[] args){  
    Rectangle r = new Rectangle(20,30);  
    Magic.println("The rectangle is "+r);  
}
```



Example: Students

- Let's do an example where we make Student class
- To do this first we need to figure out what are the properties of students?
- Next we figure out what actions we want students to perform.
- Then let's make a few Student objects and see what we can do.



Example: Professor

- Let's do an example where we make Professor class.
- To do this first we need to figure out what are the properties of professors?
- Next we figure out what actions we want professors to perform.
- Then let's make a few Professor objects and see what we can do.
- Hmm... There is some overlap between professors and students It'd be real nice to type less... We'll get to that in the future.



No More Magic



No More Magic (Mostly)

- Let's kill magic
- Here is how to "actually" print

```
System.out.println("Hello World, again");
```



Input

- Put this at the top of the class

```
import java.util.Scanner;
```

- At the start of the function where you want to get input:

```
Scanner scan = new Scanner(System.in);
```

- Use these methods to get input

```
int a = scan.nextInt();  
double b = scan.nextDouble();  
String str = scan.nextLine();
```

- When done, close

```
scan.close();
```



Weirdness

- Let's try these two code snippets and see what happens here

```
Scanner scan = new Scanner(System.in);
System.out.println("Enter a string");
String s = scan.nextLine();
System.out.println("Enter an int");
int a = scan.nextInt();
System.out.println("Your integer is \""+a+"\"");
System.out.println("Your string is \""+s+"\"");
scan.close();
```

```
Scanner scan = new Scanner(System.in);
System.out.println("Enter an int");
int a = scan.nextInt();
System.out.println("Enter a string");
String s = scan.nextLine();
System.out.println("Your integer is \""+a+"\"");
System.out.println("Your string is \""+s+"\"");
scan.close();
```



Why?

- `nextInt()` gets the next integer entered
- `nextLine()` gets the next line entered
- What happens when the user types 42

4 2 ↵

- So, `nextLine()` gets just the return statement.



How to fix?

- Easiest way to fix is after every `nextInt()`, `nextDouble()`, etc... is call `nextLine()` again and just ignore the result.

```
Scanner scan = new Scanner(System.in);

System.out.println("Enter an int");
int a = scan.nextInt();
scan.nextLine();

System.out.println("Enter a string");
String s = scan.nextLine();

System.out.println("Your integer is \""+a+"\"");
System.out.println("Your string is \""+s+"\"");
scan.close();
```



No More Magic?

- Soon, Magic will disappear from this world.
- For now, you can just use it for drawing and files.



More Powerful Arrays



Data Structures

- We have focused a lot on syntax this semester.
- We've discussed algorithms and the techniques for creating solutions to problems.
- So, far we haven't really discussed how data is stored aside from variables and arrays. An area called **data structures**.



Limitations

- With only variables and arrays, we run into limitations very quickly: We can't increase the size of arrays after they are created



Changing the Size

- To change the size of an array, we are going to use a type of object called an `ArrayList`.
- `ArrayList` is a weird type because it used a concept we haven't covered so far called **generics**.
- Before we cover generics, lets give the basic syntax for arrays



```
import java.util.ArrayList;

public class Starter {

    public static void main(String[] args){
        ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
        rectangles.add(new Rectangle(20,30,40,50));
        rectangles.add(new Rectangle(80,100,20,500));
        rectangles.get(1);
        int size = rectangles.size();
        rectangles.remove(0);

    }
}
```



Notice that Rectangle is in the <>
This is because rectangles
is an array of Rectangle objects

```
import java.util.ArrayList;

public class Starter {

    public static void main(String[] args){
        ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
        rectangles.add(new Rectangle(20,30,40,50));
        rectangles.add(new Rectangle(80,100,20,500));
        rectangles.get(1);
        int size = rectangles.size();
        rectangles.remove(0);

    }
}
```



Generic

- **Generics** are classes that are written generically so that they can be used with different types
- Generics can work with any type of object. As long as we pass it via the `<>` tokens.
- Generics are very useful when creating advanced data structures.
- We won't talk about how to make a generic class in CS110, but we will in CS120.



Generic Limitation

- Generics only work with Classes.
- So, how do create an array list of `int`, `double`, or any of the other built in types?
- Java includes a "class" version of these.
- Their names are: `Integer`, `Double`, etc. (Notice the capital letter)
- You can use them mostly like normal `int` and `double`.
- **Let's do an example.**



Multidimensional ArrayLists

- What's the limitation of built-in multidimensional arrays that we have discussed?
- Well it can't grow in size.
- How do we fix this?
- Create a multidimensional `ArrayList`.
- Syntax for this going to get a little wonky



Let's talk through this

```
ArrayList<ArrayList<Integer>> weirdAL = new ArrayList<ArrayList<Integer>>();  
  
//Make Row 0  
ArrayList<Integer> row0 = new ArrayList<Integer>();  
row0.add(10);  
row0.add(20);  
row0.add(30);  
  
//Make Row 1  
ArrayList<Integer> row1 = new ArrayList<Integer>();  
row1.add(40);  
row1.add(50);  
row1.add(60);  
  
//Add Rows to Multidimensional Arrays.  
weirdAL.add(row0);  
weirdAL.add(row1);
```



```
int sum = 0;
for(int rowIndex = 0; rowIndex < weirdAL.size(); rowIndex++){
    ArrayList<Integer> row = weirdAL.get(rowIndex);
    for(int colIndex = 0; colIndex < row.size(); colIndex++){
        sum += row.get(colIndex);
    }
}
System.out.println(sum);
```

```
int sum = 0;
for(ArrayList<Integer> row: weirdAL){
    for(Integer element: row){
        sum += element;
    }
}
System.out.println(sum);
```



Larger Multidimensional ArrayLists

- Can we make larger multidimensional ArrayLists?
- Yes, we can. The syntax is the same, but a bit ugly.



More Limitations

- There are many more limitations we run into with just built in arrays and variables; however, studying these is outside of the scope of CS110.
- If you are interested, you should take CS120, where we start discussing these at length.

