CS 106A, Lecture 22
More Classes

suggested reading:

Java Ch. 6

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Learning Goals

- Know how to define our own variable types
- Know how to define variable types that inherit from other types
- Be able to write programs consisting of multiple classes

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Announcements

- Assignment 5 due/Assignment 6 out Monday
- Reminder: the 106A website's "Schedule" page has lots of neat stuff for each lecture!
  - Slides and suggested reading sections
  - Starter code and polished solutions for live-coded programs
  - CodeStepByStep practice problems
- Midterm regrade requests can be made on Gradescope until 1PM on Monday

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

What Is A Class?

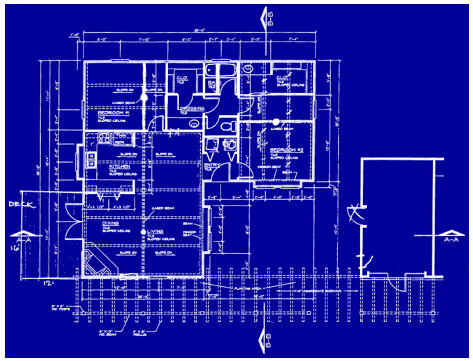
A class defines a new variable type.

Classes Are Like Blueprints

iPod blueprint (class)

state:
- current song
- volume
- battery life

behavior:
- power on/off
- change station/song
- change volume
- choose random song



constructs

iPod (variable) #1

state:
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod (variable) #2

state:
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod (variable) #3

state:
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

Creating A New Class

1. **What information is inside this new variable type?**
   - These are its instance variables.
2. **How do you create a variable of this type?**
   - This is the constructor.
3. **What can this new variable type do?**
   - These are its public methods.

Example: BankAccount

Let’s see the code!

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Printing Variables

- By default, Java doesn't know how to print objects.

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1); // ba1 is BankAccount@9e8c34

// better, but cumbersome to write
println("ba1 is " + ba1.getName() + " with $"
        + ba1.getBalance()); // ba1 is Marty with $1.25


// desired behavior
println("ba1 is " + ba1);   // ba1 is Marty with $1.25
```

The toString Method

A special method in a class that tells Java how to convert an object into a string.

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// the above code is really calling the following:
println("ba1 is " + ba1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (base 16)

```
BankAccount@9e8c34
```

The toString Method

```
public String toString() {
    code that returns a String
    representing this object;
}
```

– Method name, return, and parameters must match exactly.

– Example:

```
// Returns a String representing this account.
public String toString() {
    return name + " has $" + balance;
}
```

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

The “this” Keyword

`this`: Refers to the object on which a method is currently being called

```
BankAccount ba1 = new BankAccount();
ba1.deposit(5);

// in BankAccount.java
public void deposit(double amount) {
      // for code above, “this” -> ba1
      ...
}
```

Using "this"

Sometimes we want to name parameters the same as instance variables.

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String newName) {
        name = newName;
    }
}
```

– Here, the parameter to `setName` is named `newName` to be distinct from the object's field `name` .

Using “this”

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        name = name;
    }
}
```

Using "this"

We can use "this" to specify which one is the instance variable and which one is the local variable.

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        this.name = name;
    }
}
```

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Practice: Employee

Let's define a new variable type called **Employee** that represents a single Employee.

What information would an Employee store?

What could an Employee do?

How would you create a new Employee variable?

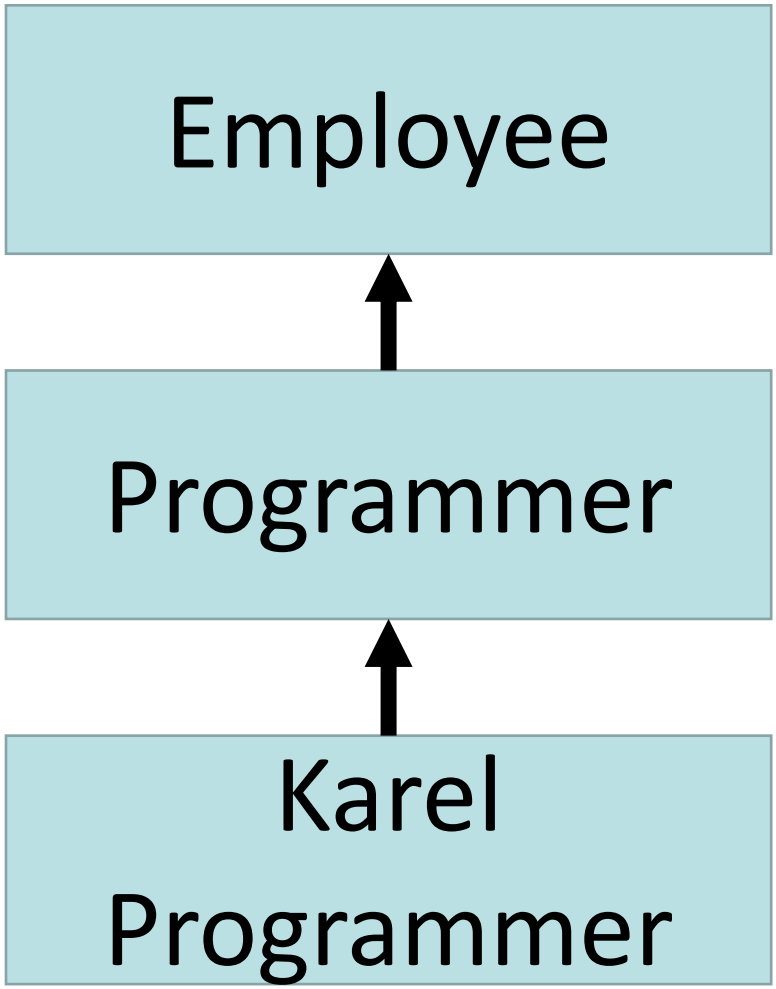
Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Inheritance

Inheritance lets us relate our variable types to one another.

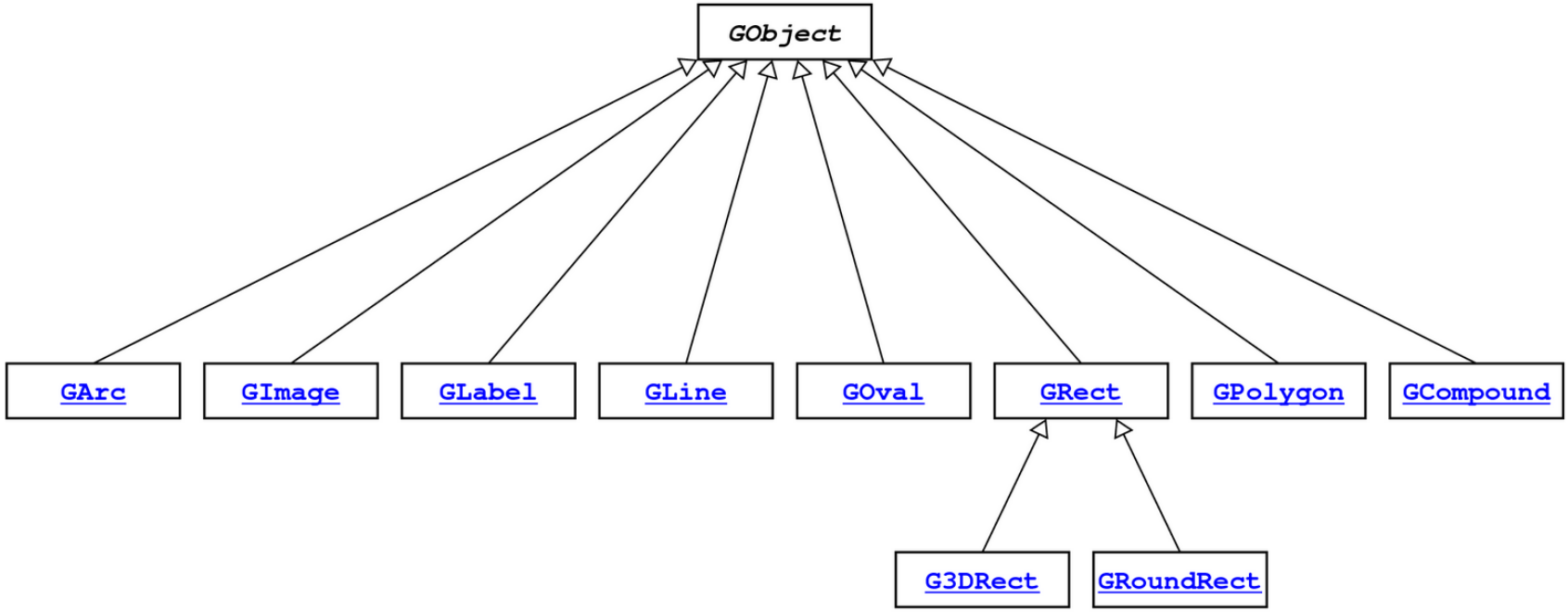
Inheritance



Variable types can seem to "inherit" from one other. We don't want to have to duplicate code for each one!

Example: GObjects

- The Stanford library uses an inheritance hierarchy of graphical objects based on a common superclass named **`GObject`**.



Example: GObjects

- **`GObject`** defines the state and behavior common to all shapes:

```
contains(x, y)
getColor(), setColor(color)
getHeight(), getWidth(), getLocation(), setLocation(x, y)
getX(), getY(), setX(x), setY(y), move(dx, dy)
setVisible(visible), sendForward(), sendBackward()
toString()
```

- The subclasses add state and behavior unique to them:

`GLabel`	**`GLine`**	**`GPolygon`**
`get/setFont`	`get/setStartPoint`	`addEdge`
`get/setLabel`	`get/setEndPoint`	`addVertex`
		`get/setFillColor`
`...`	`...`	`..`

Using Inheritance

```
public class Name extends Superclass {
```

– Example:

```
public class Programmer extends Employee {
    ...
}
```

- By extending Employee, this tells Java that `Programmer` can do **everything an Employee can do, plus more**.
- Programmer automatically inherits all of the code from Employee!
- The **superclass** is Employee, the **subclass** is Programmer.

Example: Programmer

```
public class Programmer extends Employee {
      private int timeCoding;
      ...
      public void code() {
            timeCoding += 10;
      }
}

...

Programmer annie = new Programmer("Annie");
annie.code();           // from Programmer
annie.promote();        // from Employee!
```

Example: KarelProgrammer

```
public class KarelProgrammer extends Programmer {
      private int numBeepersPicked;
      ...
      public void pickBeepers() {
            numBeepersPicked += 2;
      }
}

...
KarelProgrammer colin = new KarelProgrammer("Colin");
colin.pickBeepers();          // from KarelProgrammer
colin.code();                 // from Programmer!
colin.promote();              // From Employee!
```

Advanced: Overriding

```
public class KarelProgrammer extends Programmer {
      ...

      @Override
      public boolean promote() {
            salary *= 3;
            return true;
      }
}

...
KarelProgrammer colin = new KarelProgrammer("Colin");
colin.promote();     // From KarelProgrammer, not Employee!
```

Advanced: Overriding

```
public class Clicker extends GraphicsProgram {
	...

	@Override
	public void mouseClicked(MouseEvent e) {
		// do some stuff
	}
}
```

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

GCanvas

- A **GCanvas** is the canvas area that displays all graphical objects in a **GraphicsProgram**.
- When you create a **GraphicsProgram**, it automatically creates a **GCanvas** for itself, puts it on the screen, and uses it to add all graphical shapes.

- **GCanvas** is the one that contains methods like:
  - getElementAt
  - add
  - remove
  - getWidth
  - getHeight
  - …

GCanvas

```
public class Graphics extends GraphicsProgram {
	public void run() {
		// A GCanvas has been created for us!
		GRect rect = new GRect(50, 50);
		add(rect); // adds to the GCanvas!


		...
		// Checks our GCanvas for elements!
		GObject obj = getElementAt(25, 25);
	}
}
```

Extending GCanvas

```
public class Graphics extends Program {
      public void run() {
            // We have to make our own GCanvas now
            MyCanvas canvas = new MyCanvas();
            add(canvas);

            // Can't do this anymore, because we are
            // not using GraphicsProgram's provided
            // canvas
            // GObject obj = getElementAt(…);
      }
}
```

Extending GCanvas

```
public class Graphics extends Program {
      public void run() {
            // We have to make our own GCanvas now
            MyCanvas canvas = new MyCanvas();
            add(canvas);

            // Operate on this canvas
            GObject obj = canvas.getElementAt(…);
      }
}
```

Extending GCanvas

```
public class MyCanvas extends GCanvas {
      public void addCenteredSquare(int size) {
            GRect rect = new GRect(size, size);
            int x = getWidth() / 2.0 –
                  rect.getWidth() / 2.0;
            int y = getHeight() / 2.0 –
                  rect.getHeight() / 2.0;
            add(rect, x, y);
      }
}
```

Extending GCanvas

```
public class Graphics extends Program {
	public void run() {
		// We have to make our own GCanvas now
		MyCanvas canvas = new MyCanvas();
		add(canvas);

		canvas.addCenteredSquare(20);
	}
}
```

Extending GCanvas

- Sometimes, we want to be able to have all of our graphics-related code in a separate file.
- To do this, instead of using the provided **GraphicsProgram** canvas, we **define our own subclass of GCanvas,** have our program **extend Program**, and add our own canvas ourselves.
- Then, all graphics-related code can go in our **GCanvas** subclass.

The init method

- **`init`** is a special public method, like **`run`**, that is called when your program is being initialized.
- Unlike **`run`**, however, it is called *before* your program launches, letting you do any initialization you need.

```
public class MyProgram extends GraphicsProgram {
	public void init() {
		// executed before program launches
	}

	public void run() {
		// executed after program launches
	}
}
```

The init method

- **`init`** is typically used to initialize graphical components, such as adding a custom **`GCanvas`** to the screen.

```
public class MyProgram extends Program {
	private MyCanvas canvas;
	public void init() {
		canvas = new MyCanvas();
		add(canvas);
	}

	public void run() {
		canvas.addCenteredSquare(20);
	}
}
```

Common Bugs

- When you are using a custom canvas, make sure to not call **`getWidth`** or **`getHeight`** on the canvas until it is shown onscreen!

```
public class MyProgram extends Program {
        private MyCanvas canvas;
        public void init() {
                // canvas not created yet!
                canvas = new MyCanvas();
                // canvas not added yet!
                add(canvas);
                // window not showing yet!
        }
        public void run() {
                // good to go
        }
}
```

Preview: Aquarium

- Let's write a graphical program called **Aquarium** that simulates fish swimming around.
- To decompose our code, we can make our own **GCanvas** subclass.

Plan for today

- Announcements
- Review: Classes
- toString
- this
- Practice: Employee
- Inheritance
- Preview: Extending GCanvas
- Recap

Recap

- Classes let us define our own variable types, with their own instance variables, methods and constructors.
- We can **relate** our variable types to one another by using **inheritance**. One class can **extend** another to inherit its behavior.
- We can **extend GCanvas** in a graphical program to decompose all of our graphics-related code in one place.

Next time: Interactors and GUIs