CS 106A, Lecture 22
More Classes

suggested reading:

Java Ch. 6

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

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

Plan for today

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

What Is A Class?

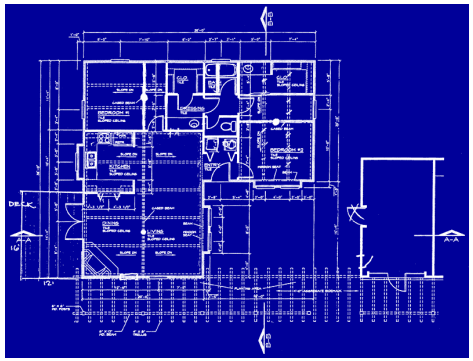
A class defines a new variable type.

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

What if…

What if we could write a program like this:

```
BankAccount nickAccount = new BankAccount();
nickAccount.setName("Nick");
nickAccount.deposit(50);

BankAccount rishiAccount = new BankAccount();
rishiAccount.setName("Rishi");
rishiAccount.deposit(50);
boolean success = rishiAccount.withdraw(10);
if (success) {
      println("Rishi withdrew $10.");
}
```

Creating A New Class

1. **What information is inside this new variable type?** These are its private instance variables.

Example: BankAccount

```
// In file BankAccount.java
public class BankAccount {
     // Step 1: the data inside a BankAccount
     private String name;
     private double balance;
}
```

Each BankAccount object has its *own copy* of all instance variables.

Creating A New Class

1. **What information is inside this new variable type?** These are its instance variables.
2. **What can this new variable type do?** These are its public methods.

Example: BankAccount

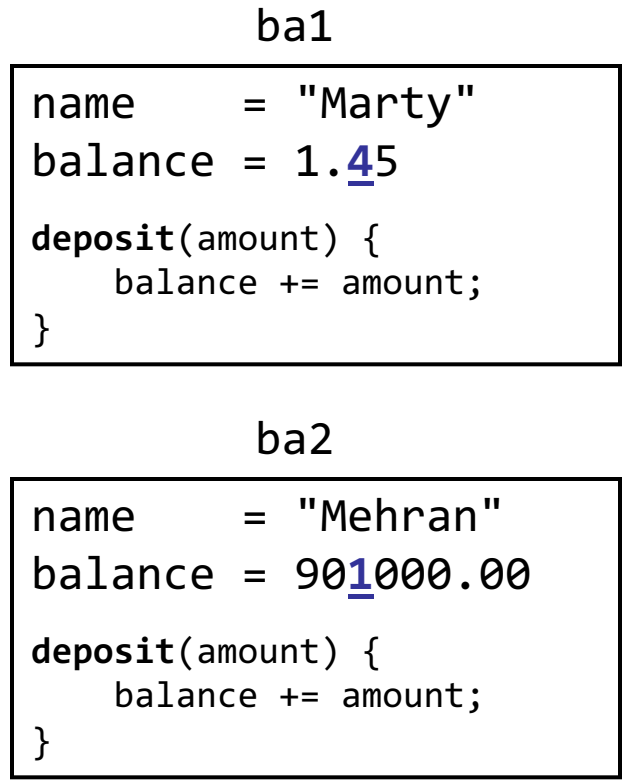
```
public class BankAccount {
	// Step 1: the data inside a BankAccount
	private String name;
	private double balance;

	// Step 2: the things a BankAccount can do
	public void deposit(double amount) {
		balance += amount;
	}
	public boolean withdraw(double amount) {
		if (balance >= amount) {
			balance -= amount;
			return true;
		}
		return false;
	}
}
```

Defining Methods In Classes

Methods defined in classes can be called **on an instance of that class**.

When one of these methods executes, it can reference **that object's copy** of instance variables.

```
ba1.deposit(0.20);
ba2.deposit(1000.00);
```



This means calling one of these methods on different objects has *different effects.*

Getters and Setters

Instance variables in a class should *always be private*. This is so only the object itself can modify them, and no-one else.

To allow the client to reference them, we define public methods in the class that **set** an instance variable's value and **get** (return) an instance variable's value. These are commonly known as **getters** and **setters**.

```
account.setName("Nick");
String accountName = account.getName();
```

Getters and setters prevent instance variables from being tampered with.

Example: BankAccount

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

	...
	public void setName(String newName) {
		if (newName.length() > 0) {
			name = newName;
		}
	}

	public String getName() {
		return name;
	}
}
```

Creating A New Class

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

Constructors

```
BankAccount ba1 = new BankAccount();

BankAccount ba2 = new BankAccount("Nick", 50);
```

The constructor is executed when a new object is created.

Example: BankAccount

```
public class BankAccount {
  // Step 1: the data inside a BankAccount
  private String name;
  private double balance;

  // Step 2: the things a BankAccount can do (omitted)
  // Step 3: how to create a BankAccount
  public BankAccount(String accountName, double startBalance) {
       name = accountName;
       balance = startBalance;
  }

  public BankAccount(String accountName) {
       name = accountName;
       balance = 0;
  }
}
```

Using Constructors

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

ba1

```
name    = "Marty"
balance = 1.25

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

```
BankAccount ba2 =
    new BankAccount("Mehran", 900000.00);
```

ba2

```
name    = "Mehran"
balance = 900000.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

- When you call a constructor (with **new**):
  - Java creates a new object of that class.
  - The constructor runs, on that new object.
  - The newly created object is returned to your program.

Plan for today

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

Printing Variables

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

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

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


// 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

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

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

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

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?

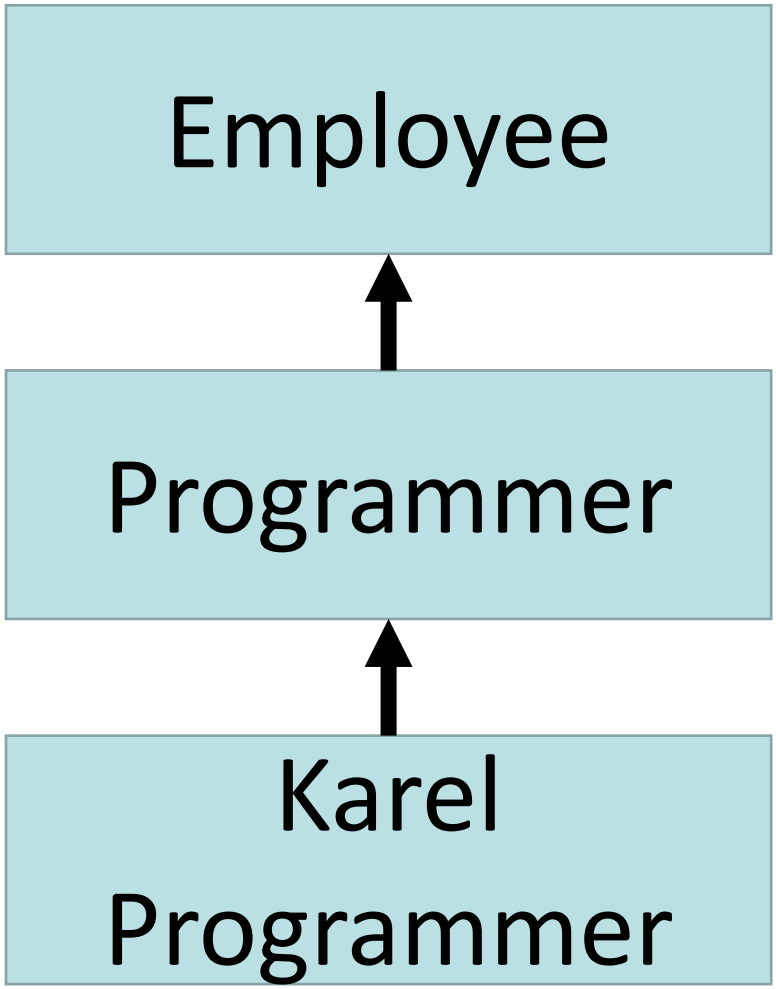
Plan for today

- Recap: Classes
- toString
- this
- *Example:* Employee
- Inheritance

Inheritance

Inheritance lets us relate our variable types to one another.

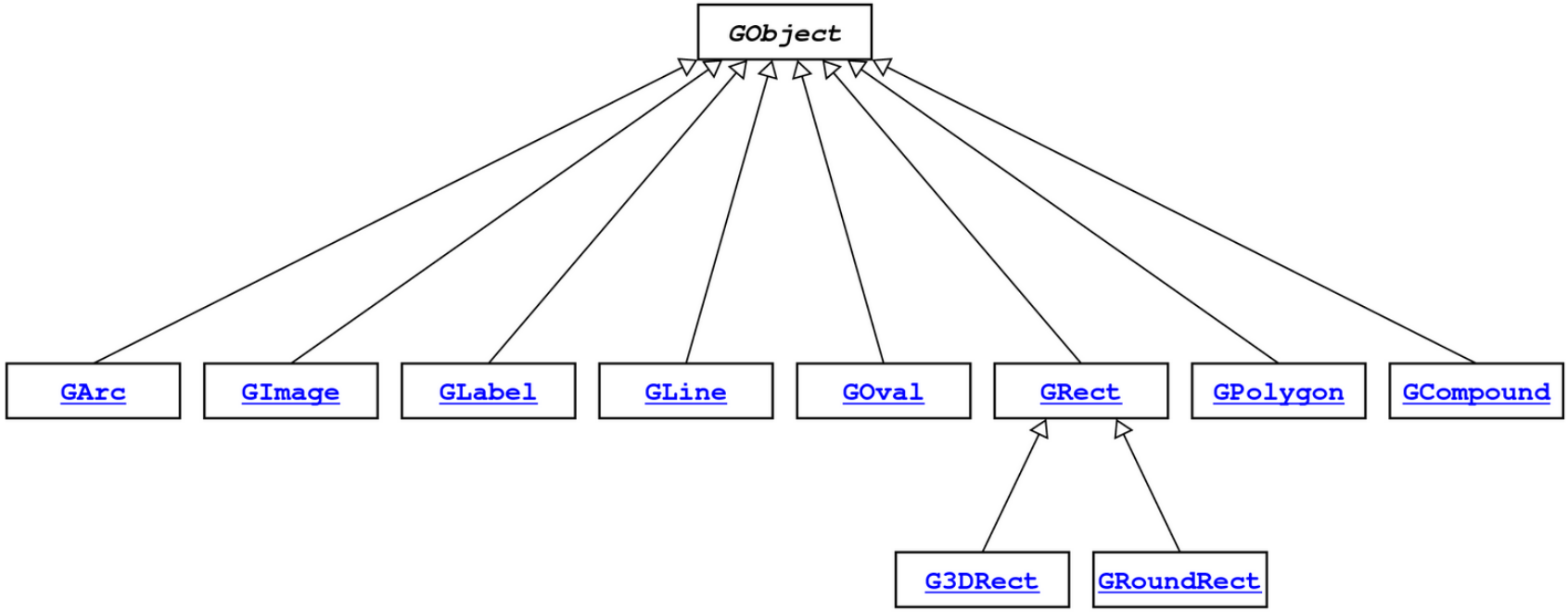
Inheritance



Variable types can seem to "inherit" from each 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 rishi = new Programmer("Rishi");
rishi.code();           // from Programmer
rishi.promote();        // from Employee!
```

Example: KarelProgrammer

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

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

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 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.

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