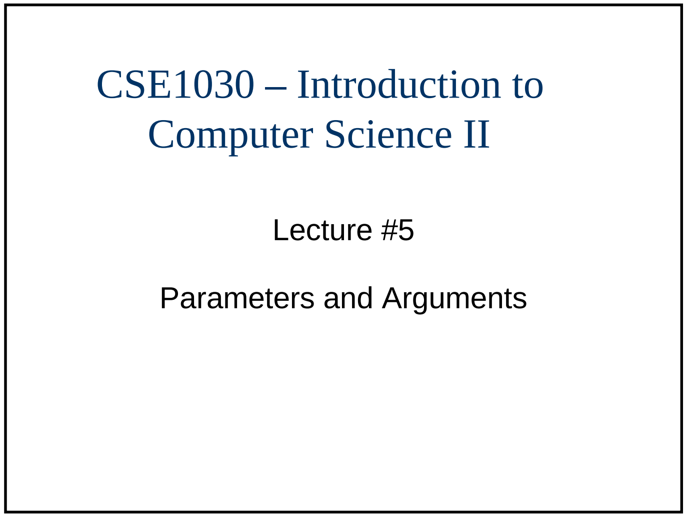
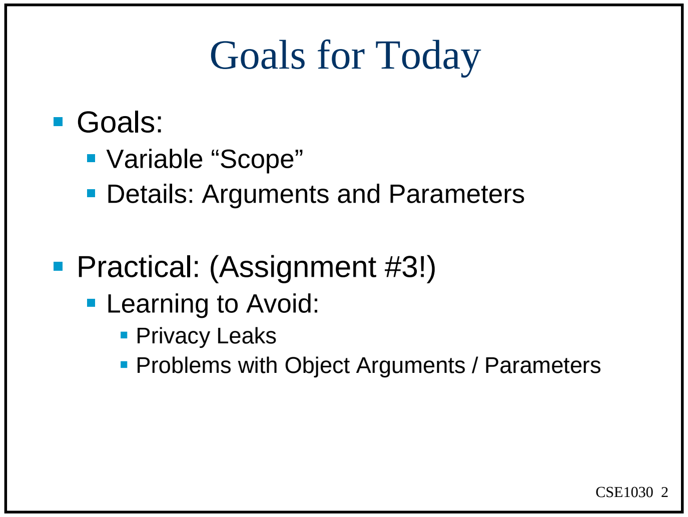


CSE1030 – Introduction to Computer Science II

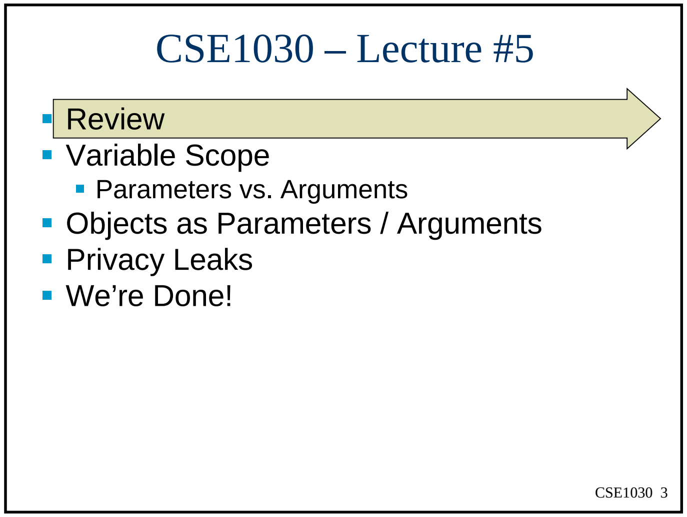
Lecture #5

Parameters and Arguments



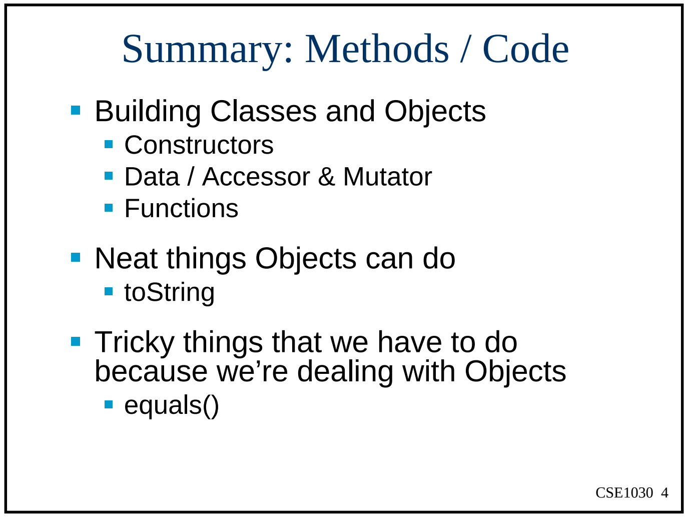
Goals for Today

- Goals:
  - Variable “Scope”
  - Details: Arguments and Parameters
- Practical: (Assignment #3!)
  - Learning to Avoid:
    - Privacy Leaks
    - Problems with Object Arguments / Parameters



CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Objects as Parameters / Arguments
- Privacy Leaks
- We’re Done!



Summary: Methods / Code

- Building Classes and Objects
  - Constructors
  - Data / Accessor & Mutator
  - Functions
- Neat things Objects can do
  - toString
- Tricky things that we have to do because we’re dealing with Objects
  - equals()

CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Objects as Parameters / Arguments
- Privacy Leaks
- We're Done!

Variable Scope

- What is "Scope"?
  - Variable Scope refers to the areas within your program in which a variable is available
- Why do we care?
  - So we don't write confusing code
  - So we control access to our data

Variable Scope

- The Basic Rule is that variables are only visible within the curly brackets that they are defined inside of
- For example, the body of a function…

```
public void method1()
{
   ...

   int C = 20;

   ...
}
```

Variable Scope

- … or, the body of a class …

```
public class scope1()
{
   ...

   int A = 10;
   static int B = 20;

   ...
}
```

Variable Scope

- … or, the inside a block of code:

```
public void method1()
{
   ...

   while(...)
   {
      int C = 20;
   }

   ...
}
```

But it’s Not Quite That Simple

```
public class scope1
{
  public static void method1()
  {
     System.out.println("A10 = " + A10);
  }

  public static int A10 = 10;

  public static void main(String[] args)
  {
     System.out.println("A10 = " + A10);

     System.out.println("B20 = " + B20);
     int B20 = 50;
     System.out.println("B20 = " + B20);

     method1();
  }
}
```

Ok!

Not Ok!

Function Parameters

- Function Parameters exist anywhere within the curly brackets

```
public void method2(int D)
{
   ...
}
```

Aside: Parameters versus Arguments

- A **Parameter** is the variable: `x`
- An **Argument** is the value: `10`

```
double calc(double x)
{
   return x * Slope + Offset;
}

System.out.println("the answer is: " + calc(10));
```

So, the Basic Scope Rules really are:

- Classes
  - Scope starts from the top {
  - Goes all the way to the bottom }
- Functions
  - Scope starts at the variable definition
  - Goes all the way to the bottom }
  - But parameters start at the top {

Advanced Scoping Rules

- ... But there are ways to break or circumvent the Basic Scoping Rules, like:
  - Declaring variables (or functions) **`public`**
  - Passing Objects into functions as Arguments
  - Example…

Useful Example Class

```
public class LinearEquationSolver
{
   double Slope;
   double Offset;

   LinearEquationSolver(double slope, double offset)
   {
      Slope  = slope;
      Offset = offset;
   }

   double calc(double x)
   {
      return x * Slope + Offset;
   }
}
```

Calculates:

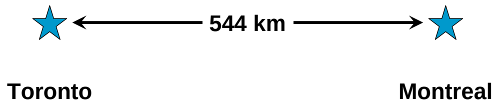
y = m*x + b

or in this case:

y = Slope*x + Offset

Question: How far am I from Toronto?

- Driving (100 km/h) from Montreal to Toronto for ***t*** hours:



- Distance = -***t*** * speed + distance
  = -100****t*** + 544

```
solver = new LinearEquationSolver(-100, 544);
```

Distance Example client

```
public class client
{
   public static void main(String[] args)
   {
      LinearEquationSolver solver
               = new LinearEquationSolver(-100, 544);

      double t = 2.0;   // driving for 2 hours

      System.out.println("Driving for " + t + " hours");
      System.out.println("Average Speed is "
                  + -solver.Slope + " km/h");
      System.out.println("Remaining Distance "
                       + solver.calc(t) + " km");
   }
}
```

Q: Why can I access the **`Slope`** variable? It is not in my scope.

A: Because it is not **`private`**!

Output of client program

```
Driving for 2.0 hours
Average Speed is 100.0 km/h
Remaining Distance 344.0 km
```

Advanced Scoping Rules

- ... But there are ways to break or circumvent the Basic Scoping Rules, like:
  - Declaring variables (or functions) **`public`**
  - Passing Objects into functions as Arguments
  - Example…

Passing Objects into a Function

```
public class client
{
   static void printSpeed(LinearSolver solver)
   {
      System.out.println("Ave. Speed is " + speed + " km/h");
      System.out.println("Ave. Speed is " + -solver.Slope
                 + " km/h");
   }

   public static void main(String[] args)
   {
      double speed = 100;
      LinearEquationSolver solver
               = new LinearEquationSolver(-speed, 544);
      double t = 2.0;   // driving for 2 hours
      System.out.println("Driving for " + t + " hours");
      printSpeed(solver);
      System.out.println("Remaining Distance "
             + solver.calc(t) + " km");
   }
}
```

CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Objects as Parameters / Arguments
- Privacy Leaks
- We're Done!

Objects as Parameters, Arguments, and Return Values

- **When an object is passed to a function's Parameter as an Argument**, the object is not copied! Instead, the **arrow (pointer) is passed**, yielding access to the original object.
- The same thing happens when an object is returned from a function.

Non Object Parameter Passing

```
public class example
{
   static int FUNCTION(int i2)
   {
      System.out.println("i2 (before) = " + i2 + "  == 100?");
      i2 = 200;
      System.out.println("i2 (after)  = " + i2 + "  == 200?");
      return i2;
   }

   public static void main(String[] args)
   {
      int i1 = 100;
      System.out.println("i1 = " + i1 + "  == 100?");

      System.out.println("Calling FUNCTION!");
      int i3 = FUNCTION(i1);

      System.out.println("i1 = " + i1 + "  == 100?");
      i1 = 400;
      System.out.println("i3 = " + i3 + "  == 200?");
   }
}
```

Results

```
i1 = 100  == 100?
Calling FUNCTION!
i2 (before) = 100  == 100?
i2 (after)  = 200  == 200?
i1 = 100  == 100?
i3 = 200  == 200?
```

Primitive data types (integers, floats, doubles) are **passed by value** meaning their values get copied on the way into (parameter/argument) and out of (return) a function.

Object Parameter Passing

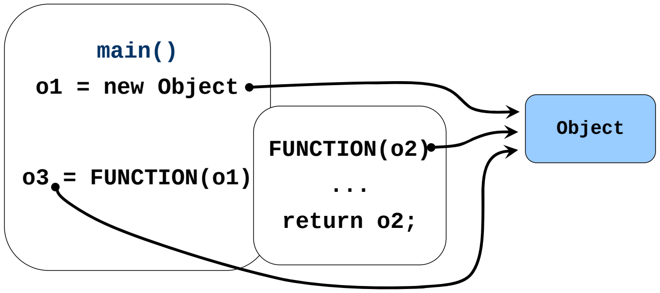
```
public class Int
{
   // data
   public int I;

   // Constructors
   public Int(int i) { I = i;   }  // regular
   public Int(Int i) { I = i.I; }  // copy

   // toString
   public String toString() { return Integer.toString(I); }

   // an example function
   static Int FUNCTION(Int i2)
   {
      System.out.println("i2 (before) = " + i2 + "  == 100?");
      i2.I = 200;
      System.out.println("i2 (after)  = " + i2 + "  == 200?");
      return i2;
   }
```

```
   public static void main(String[] args)
   {
      Int i1 = new Int(100);
      System.out.println("i1 = " + i1 + "  == 100?");

      System.out.println("Calling FUNCTION!");
      Int i3 = FUNCTION(i1);

      System.out.println("i1 = " + i1 + "  == 100?");
      i1.I = 400;
      System.out.println("i3 = " + i3 + "  == 200?");
   }
}
```

Results

```
i1 = 100  == 100?
Calling FUNCTION!
i2 (before) = 100  == 100?
i2 (after)  = 200  == 200?
i1 = 200  == 100?
i3 = 400  == 200?
```

Objects as Parameters and Arguments



The **arrows (pointers)** to the objects are what get copied on the way into (parameter/argument) and out of (return) a function.

So, are Strings Primitives, or Objects?

```
public class stringexample
{
   static String FUNCTION(String i2)
   {
      System.out.println("i2 (before) = " + i2 + "  == 100?");
      i2 = "200";
      System.out.println("i2 (after)  = " + i2 + "  == 200?");
      return i2;
   }


   public static void main(String[] args)
   {
      String i1 = "100";
      System.out.println("i1 = " + i1 + "  == 100?");

      System.out.println("Calling FUNCTION!");
      String i3 = FUNCTION(i1);
      System.out.println("i1 = " + i1 + "  == 100?");
      i1 = "400";
      System.out.println("i3 = " + i3 + "  == 200?");
   }
}
```

Results

```
i1 = 100  == 100?
Calling FUNCTION!
i2 (before) = 100  == 100?
i2 (after)  = 200  == 200?
i1 = 100  == 100?
i3 = 200  == 200?
```

Although Strings are objects they behave like primitive types, because they get special treatment by Java (whenever they are changed or assigned, a new String object is automatically created).

CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Objects as Parameters / Arguments
- Privacy Leaks
- We're Done!

2 Examples of Privacy Leaks

- Privacy Leaks are accidental access to private data members caused by incorrect treatment of parameters that are objects
- The following code looks like it's doing everything correctly (**`private`** data and accessor / mutator methods)
- But something is wrong…

Aside 1: Temperature Conversion

- Fahrenheit → Celsius

  C = (F – 32) * 5 / 9

- Celsius → Fahrenheit

  F = C * 9 / 5 + 32

- Celsius → kelvin
  - K = C + 273.15

Aside 2: Recall LinearEquationSolver

```
public class LinearEquationSolver
{
   double Slope;
   double Offset;

   LinearEquationSolver(double slope, double offset)
   {
      Slope  = slope;
      Offset = offset;
   }

   double calc(double x)
   {
      return x * Slope + Offset;
   }
}
```

TemperatureSolver Utility Class

```
public class TemperatureSolver
{
   // Solver for Fahrenheit → Celsius
   private static LinearEquationSolver Fahrenheit2Celsius
      = new LinearEquationSolver(5.0/9.0, -32.0*5.0/9.0);


   // Solver for Fahrenheit → Celsius
   private static LinearEquationSolver Celsius2Fahrenheit
      = new LinearEquationSolver(9.0/5.0, 32.0);


   // Solver for Custom Conversion
   private static LinearEquationSolver CustomSolver = null;
```

```
// Fahrenheit → Celsius functions

public static double calcFahrenheit2Celsius(double f)
{
   return Fahrenheit2Celsius.calc(f);
}

public static LinearEquationSolver getFahrenheit2CelsiusSolver()
{
   return Fahrenheit2Celsius;
}


// Celsius → Fahrenheit functions

public static double calcCelsius2Fahrenheit(double c)
{
   return Celsius2Fahrenheit.calc(c);
}

public static LinearEquationSolver getCelsius2FahrenheitSolver()
{
   return Celsius2Fahrenheit;
}
```

```
   // Custom Converter Functions

   public static
      void setCustomConversion(LinearEquationSolver customsolver)
   {
      CustomSolver = customsolver;
   }


   public static LinearEquationSolver getCustomConverter()
   {
      return CustomSolver;
   }


   public static double calcCustomConversion(double x)
   {
      return CustomSolver.calc(x);
   }
}
```

Client1

```
public class client1
{
   public static void main(String[] args)
   {
      // want Celsius to kelvin custom converter
      LinearEquationSolver C2K
               = new LinearEquationSolver(1, 273.15);

      // setup custom Celsius to kelvin converter
      TemperatureSolver.setCustomConversion(C2K);

      // do some conversions
      System.out.println("Freezing Point of Water:");
      System.out.println("0 Celsius = "
            + TemperatureSolver.calcCustomConversion(0)
            + " kelvin");

      System.out.println("Boiling Point of Water:");
      System.out.println("100 Celsius = "
            + TemperatureSolver.calcCustomConversion(100)
            + " kelvin");
```

```
      // now we're going to use the converter for something else
      C2K.Slope  = -1000;
      C2K.Offset = -1000;

      System.out.println("CONVERTER HAS BEEN CHANGED!");


      // do those same conversions again
      System.out.println("Freezing Point of Water:");
      System.out.println("0 Celsius = "
            + TemperatureSolver.calcCustomConversion(0)
            + " kelvin");


      System.out.println("Boiling Point of Water:");
      System.out.println("100 Celsius = "
            + TemperatureSolver.calcCustomConversion(100)
            + " kelvin");
   }
}
```

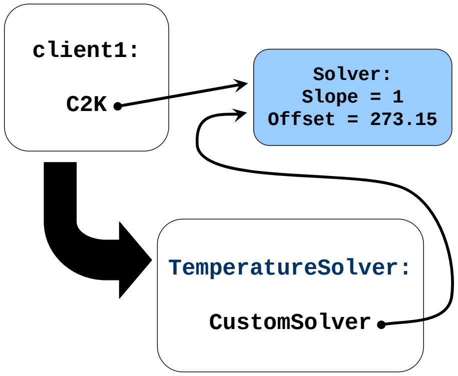
Results

```
Freezing Point of Water:
0 Celsius = 273.15 kelvin
Boiling Point of Water:
100 Celsius = 373.15 kelvin
CONVERTER HAS BEEN CHANGED!
Freezing Point of Water:
0 Celsius = -1000.0 kelvin
Boiling Point of Water:
100 Celsius = -101000.0 kelvin
```

What Happened?

Client1 created the Solver. When it passed it to the Temperature-Solver, the **arrow** was copied, not the Solver object!

So, client1 still had access to the Solver object.



client1:

C2K

Solver:
Slope = 1
Offset = 273.15

TemperatureSolver:

CustomSolver

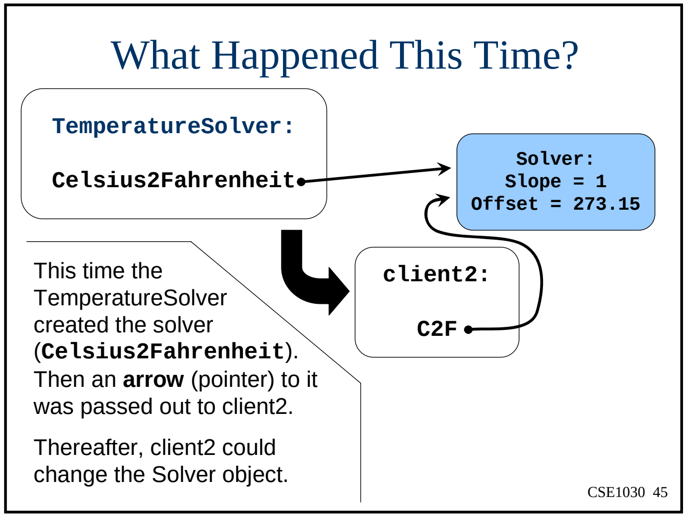
Another Example: client2

```
public class client2
{
   public static void main(String[] args)
   {
      // do some conversions

      System.out.println("Freezing Point of Water:");
      System.out.println("0 Celsius = "
            + TemperatureSolver.calcCelsius2Fahrenheit(0)
            + " Fahrenheit");

      System.out.println("Boiling Point of Water:");
      System.out.println("100 Celsius = "
            + TemperatureSolver.calcCelsius2Fahrenheit(100)
            + " Fahrenheit");
```

```
      // grab the Celsius to Fahrenheit converter
      // and change it to something else

      LinearEquationSolver C2F
            = TemperatureSolver.getCelsius2FahrenheitSolver();
      C2F.Slope  = -1000;
      C2F.Offset = -1000;

      System.out.println("CONVERTER HAS BEEN CHANGED!");


      // do those same conversions again

      System.out.println("Freezing Point of Water:");
      System.out.println("0 Celsius = "
            + TemperatureSolver.calcCelsius2Fahrenheit(0)
            + " Fahrenheit");

      System.out.println("Boiling Point of Water:");
      System.out.println("100 Celsius = "
            + TemperatureSolver.calcCelsius2Fahrenheit(100)
            + " Fahrenheit");
   }
}
```

Results

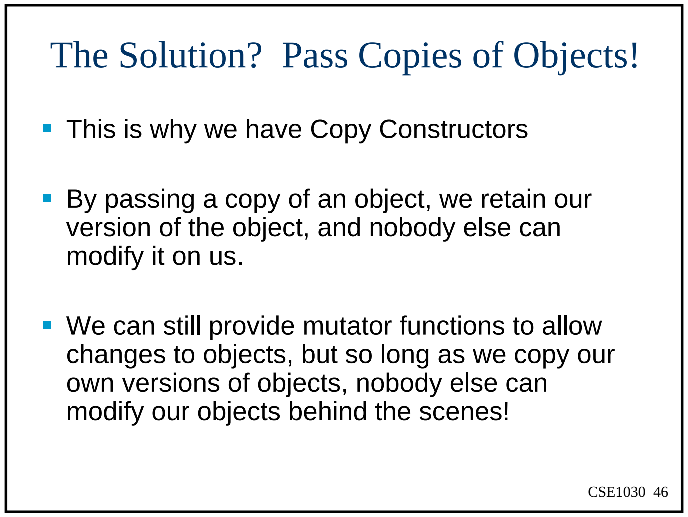
```
Freezing Point of Water:
0 Celsius = 32.0 Fahrenheit
Boiling Point of Water:
100 Celsius = 212.0 Fahrenheit
CONVERTER HAS BEEN CHANGED!
Freezing Point of Water:
0 Celsius = -1000.0 Fahrenheit
Boiling Point of Water:
100 Celsius = -101000.0 Fahrenheit
```



What Happened This Time?

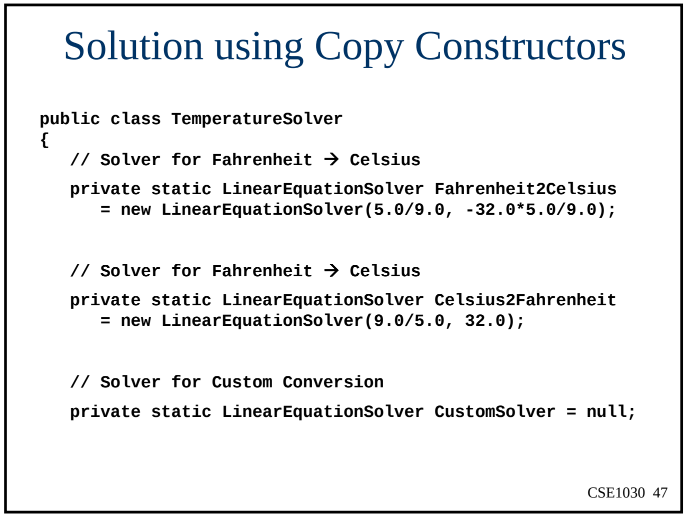
This time the TemperatureSolver created the solver (**`Celsius2Fahrenheit`**). Then an **arrow** (pointer) to it was passed out to client2.

Thereafter, client2 could change the Solver object.



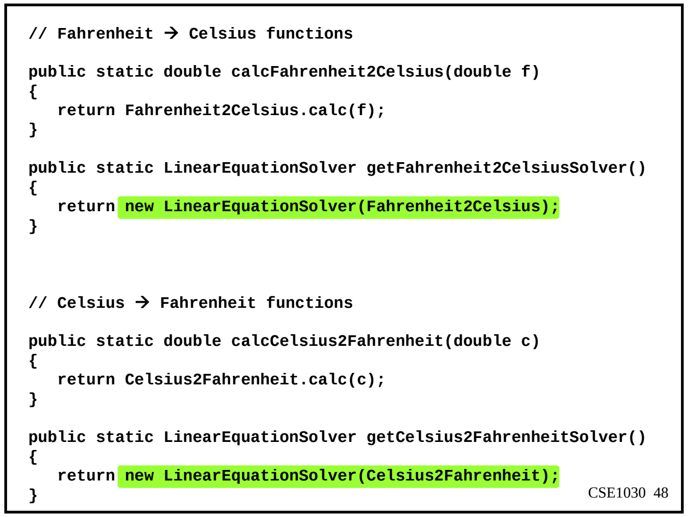
The Solution? Pass Copies of Objects!

- This is why we have Copy Constructors
- By passing a copy of an object, we retain our version of the object, and nobody else can modify it on us.
- We can still provide mutator functions to allow changes to objects, but so long as we copy our own versions of objects, nobody else can modify our objects behind the scenes!

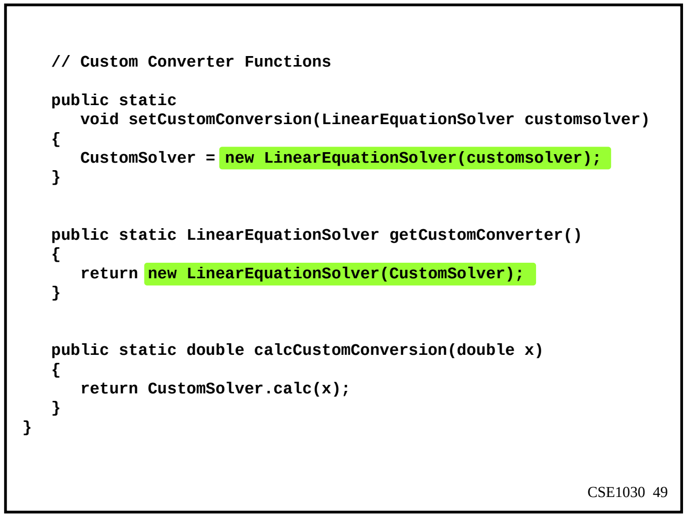


Solution using Copy Constructors

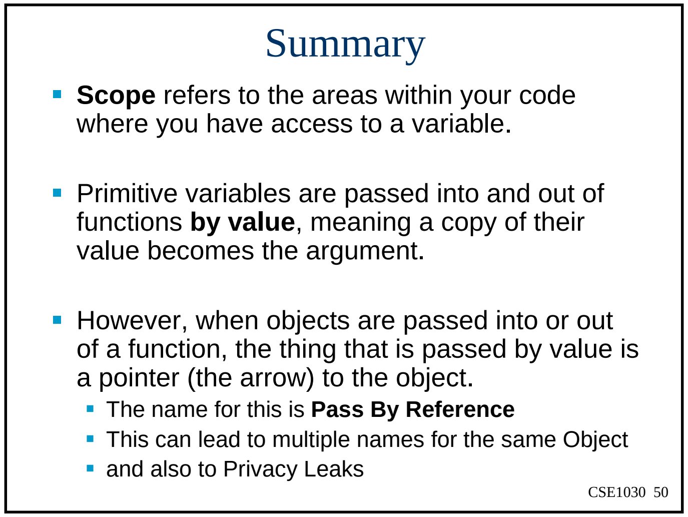
```
public class TemperatureSolver
{
   // Solver for Fahrenheit → Celsius
   private static LinearEquationSolver Fahrenheit2Celsius
      = new LinearEquationSolver(5.0/9.0, -32.0*5.0/9.0);


   // Solver for Fahrenheit → Celsius
   private static LinearEquationSolver Celsius2Fahrenheit
      = new LinearEquationSolver(9.0/5.0, 32.0);


   // Solver for Custom Conversion
   private static LinearEquationSolver CustomSolver = null;
```



```
// Fahrenheit → Celsius functions

public static double calcFahrenheit2Celsius(double f)
{
   return Fahrenheit2Celsius.calc(f);
}

public static LinearEquationSolver getFahrenheit2CelsiusSolver()
{
   return new LinearEquationSolver(Fahrenheit2Celsius);
}



// Celsius → Fahrenheit functions

public static double calcCelsius2Fahrenheit(double c)
{
   return Celsius2Fahrenheit.calc(c);
}

public static LinearEquationSolver getCelsius2FahrenheitSolver()
{
   return new LinearEquationSolver(Celsius2Fahrenheit);
}
```

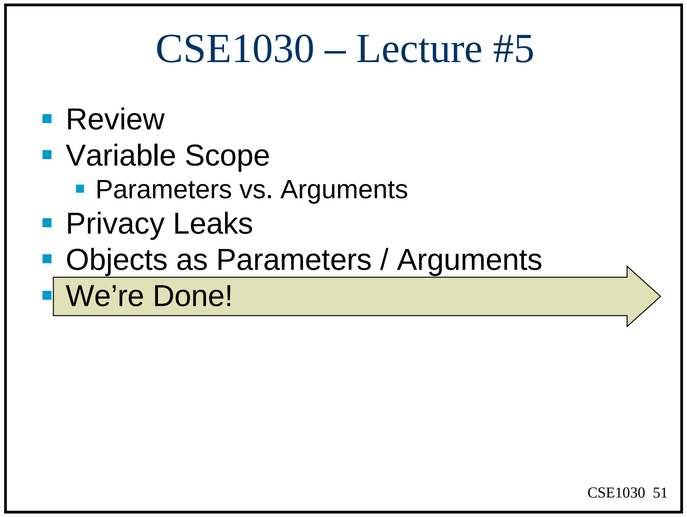


```
   // Custom Converter Functions

   public static
      void setCustomConversion(LinearEquationSolver customsolver)
   {
      CustomSolver = new LinearEquationSolver(customsolver);
   }


   public static LinearEquationSolver getCustomConverter()
   {
      return new LinearEquationSolver(CustomSolver);
   }


   public static double calcCustomConversion(double x)
   {
      return CustomSolver.calc(x);
   }
}
```



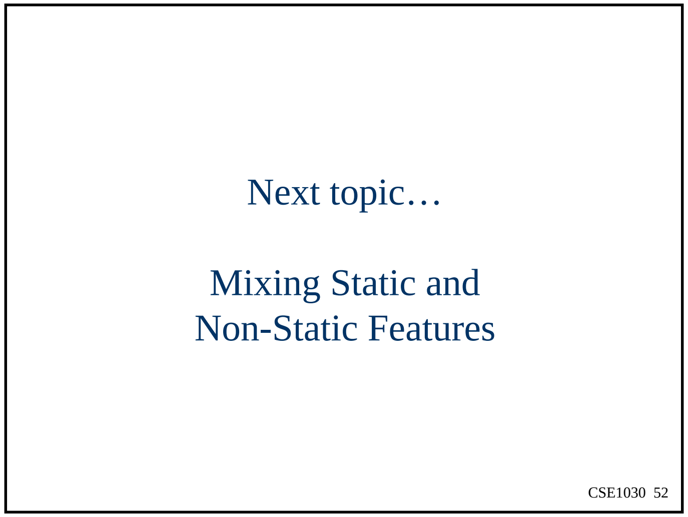
Summary

- **Scope** refers to the areas within your code where you have access to a variable.
- Primitive variables are passed into and out of functions **by value**, meaning a copy of their value becomes the argument.
- However, when objects are passed into or out of a function, the thing that is passed by value is a pointer (the arrow) to the object.
  - The name for this is **Pass By Reference**
  - This can lead to multiple names for the same Object
  - and also to Privacy Leaks



CSE1030 – Lecture #5

- Review
- Variable Scope
  - Parameters vs. Arguments
- Privacy Leaks
- Objects as Parameters / Arguments
- We’re Done!



Next topic…

Mixing Static and Non-Static Features