CSE1030 – Introduction to Computer Science II

Lecture #6

Mixing Static and
Non-Static Features

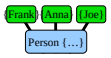
Goals for Today

- Goals:
  - Understanding static versus instance (non-static) data and code
- Practical: (Assignment #3!)
  - You will need to use both static and non-static data and code for the assignment

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

Important Concepts from Past Lectures

- In Java, everything is a Class
- Classes Define Objects
  - 

- An Object Variable is
  - A Name,
  - An Arrow (pointer) to memory, and,
  - A Block of Memory
  - 

- Static Utility Classes have no Objects

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **this**
- We're Done!

Recall the CreditCard Class (next 4)

```
public class CreditCard
{
   // instance variables/attributes/fields
   private String Name;
   private String Number;
   private double Balance;
   private double Limit;

   // constructor
   public CreditCard(String number, String name, double limit)
   {
      Name    = name;
      Number  = number;
      Balance = 0;
      Limit   = limit;
   }
```



```
// accessors
public String getName()    { return Name;    }
public String getNumber()  { return Number;  }
public double getBalance() { return Balance; }
public double getLimit()   { return Limit;   }

// mutator
public boolean setLimit(double limit)
{
   if(limit > 0)
   {
      Limit = limit;
      return true;
   }
   else
      return false;
}
```



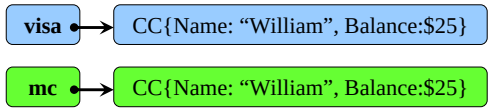
```
// charge the credit card
public boolean charge(double amount)
{
   if(amount < 0)
      return false;

   if(Balance + amount > Limit)
   {
      return false;
   }
   else
   {
      Balance += amount;
      return true;
   }
}
```

```
    // credit the credit card
    public boolean credit(double amount)
    {
       if(amount < 0)
          return false;

       Balance -= amount;

       return true;
    }
}
```


Problem? We Want the Total Balance

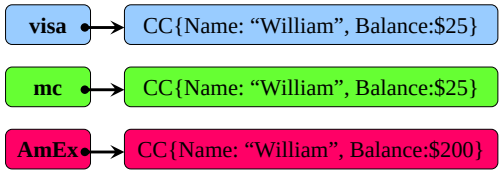
```
public class client
{
   public static void main(String[] args)
   {
      // first we create some credit-cards
      CreditCard visa = new CreditCard(
               "1234 5678 9012 3456", "William", 20000);
      CreditCard mc   = new CreditCard(
               "5678 9012 3456 7890", "William", 10000);

      // transactions
      visa.charge(100);
      visa.credit(75);   // $25 owing
      mc.charge(250);
      mc.credit(225);    // another $25 owing

      // what's the grand total?
      System.out.println("Total Owing: "
              + (visa.getBalance() + mc.getBalance()));
   }
}
```


The Big Picture



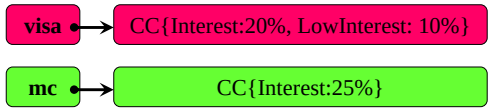
- Right now, there are two separate objects with no direct connection between them. So API user:
  - Must keep track of the cards
  - Must know details of the cards

Possible Problems?



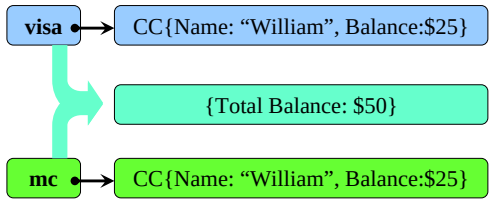
- What if we've forgotten a card?
  - or haven't been told about it?
  - or are being defrauded?

Another Problem?



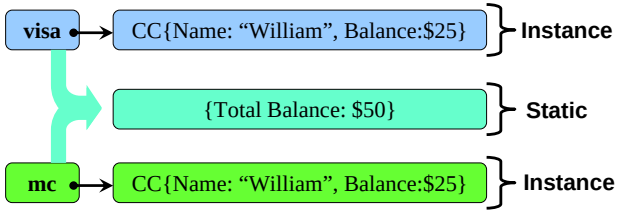
- Or, what if the question is to calculate the monthly interest?
  - We would need the client code to know details of the card's interest calculations, that really should be contained within the CreditCard class implementation

The Solution



- We need a single place to store information **common to both objects**
  - Easily **Accessible**
  - but still **Safe** from the Outside World

Static versus Instance Data



- Instance is the data and code in an object
- Static is data and code common to all objects

How does it look in the Code?

```
public class CreditCard
{
   // instance variables/attributes/fields
   private String Name;
   private String Number;
   private double Balance;
   private double Limit;

   // static data to hold the total balance
   private static double TotalBalance = 0;

   // constructor
   public CreditCard(String number, String name, double limit)
   {
      Name    = name;
      Number  = number;
      Balance = 0;
      Limit   = limit;
   }
```

```
// charge the credit card
public boolean charge(double amount)
{
   if(amount < 0)
      return false;

   if(Balance + amount > Limit)
   {
      return false;
   }
   else
   {
      Balance += amount;
      TotalBalance += amount;
      return true;
   }
}
```



```
// credit the credit card
public boolean credit(double amount)
{
   if(amount < 0)
      return false;

   Balance -= amount;
   TotalBalance -= amount;

   return true;
}

// TotalBalance accessor
public static double getTotalBalance()
{
   return TotalBalance;
}
}
```


Client Example: Secret Card (1/4)

```
public class client
{
   public static void main(String[] args)
   {
      // first we create some credit-cards
      CreditCard visa = new CreditCard(
                    "1234 5678 9012 3456", "William", 20000);
      CreditCard mc   = new CreditCard(
                    "5678 9012 3456 7890", "William", 10000);
```


Client Example: Secret Card (2/4)

```
      // transactions
      visa.charge(100);
      visa.credit(75);   // $25 owing
      mc.charge(250);
      mc.credit(225);    // another $25 owing

      // do something else...
      somethingelse.doit();

      // what's the grand total?
      System.out.println("Total Owing: "
                 + CreditCard.getTotalBalance());
   }
}
```

Client Example: Secret Card (3/4)

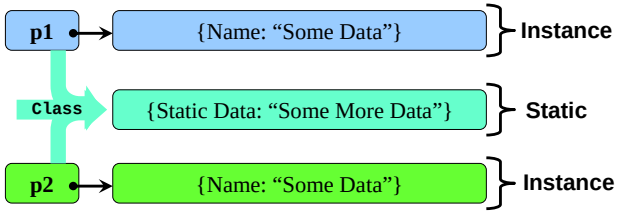
```
public class somethingelse
{
   static public void doit()
   {
      CreditCard AmEx = new CreditCard(
                "9012 3456 7890 1234", "William", 10000);
      AmEx.charge(100);
   }
}
```


Client Example: Secret Card (4/4)

- Output Without Static Data (old way) is wrong: `Total Owing: 50.0`
- Output With Static Data (old way) is correct: `Total Owing: 150.0`

Review: Regular Classes:

- Regular Classes have:
  - Instance Data (in the Objects)
  - Instance Code (does things with Objects)
  - Static Data (Shared by All Objects)
  - Static Code (Only does things with static data)



Inherent Relationships: Static versus Non-Static Data

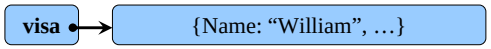
- Static Data is Best for
  - Summary Statistics
    - Counting, Serial Numbers, Profiling (Frequency, Time)
  - Class-wide **`final`**s (Constants)
- Static Code is Best for
  - Static Functions
    (Little Utilities that don't need an Object)
  - **`main()`**
- Why?
  - Pertain to a Class, Not Tied to an Object

CSE1030 – Lecture #6

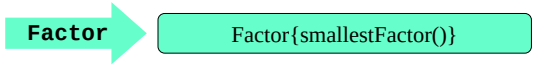
- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

Accessing Instance versus Static Data

- Instance Data and Code require an object! No object? No way to access them. Need the **Name**.

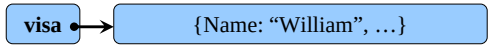


- Static Data and Code do not require an object! Can be accessed from the **Class Name**:



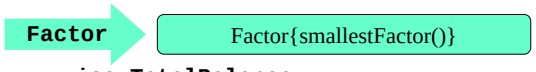
Java API Notation (Outside View)

- Instance Data and Code are accessed through an object variable:




```
visa.interestRate
visa.credit()
```

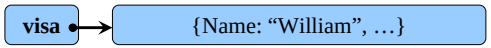
- Static Data and Code can be accessed through an object or directly from the class:




```
visa.TotalBalance
Factor.smallestFactor(int C)
```

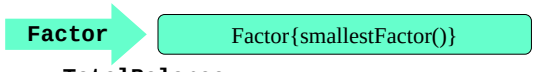

Java Notation (Inside View)

- Inside the class, instance and static data can be access directly – there is no required notational distinction:
- Instance Data and Code:




```
interestRate
credit()
```

- Static Data and Code:




```
TotalBalance
smallestFactor(int C)
```

Initialisation

- Initialise **`static`**s when they are defined (because the constructor is called once for each object created)

```
private static int Number = 42;
```

- Initialise instance variables when the object is constructed (i.e., in the Constructor)

```
class example {
   private int Number;

   example() { Number = 42; }
}
```


Initialising **`final`**s

- **`final`** denotes a constant within **a Class** or within **an Instance (Object)**
- Why?
  - Some constants pertain to the whole Class, whereas other only to an object
- Example…


```
class Number
{
   final int InstanceNumber;

   static final int ClassNumber = 101;

   public Number(int n) { InstanceNumber = n; }

   public static void main(String[] args)
   {
      // define some numbers:
      Number n7  = new Number(7);
      Number n42 = new Number(42);

      System.out.println("n7  = " + n7.InstanceNumber);
      System.out.println("n42 = " + n42.InstanceNumber);
      System.out.println("ClassNumber = " + ClassNumber);

//       n7.InstanceNumber = 45;
//       ClassNumber = 32;
   }
}
```


Output

```
> java Number
n7  = 7
n42 = 42
ClassNumber = 101
```

- Summary: Both Instances (Objects) and the entire Class can have constants.

The Implicit Parameter / Argument

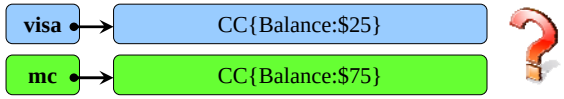
- Think about these two lines of code:

```
visa.credit(10)
mc.credit(10)
```

- They both call this function:

```
// credit the credit card
public boolean credit(double amount)
{
   if(amount < 0)
      return false;

   Balance -= amount;
   TotalBalance -= amount;

   return true;
}
```


How does Java know which Object?




```
// credit the credit card
public boolean credit(double amount)
{
   if(amount < 0)
      return false;

   Balance -= amount;
   TotalBalance -= amount;

   return true;
}
```

- In **`credit()`**, we just write "Balance", java implicitly figures-out which object (**`visa`** or **`mc`**) we are using

Implicit Parameter / Argument

- The idea is that the object by which an instance function is called is an **Implicit Parameter**, whereas our regular parameters are **Explicit**:

```
visa.credit(10)
```

Implicit Parameter

Explicit Parameter

You can Imagine the Code Automatically Becomes:

`visa.credit(10)`

```
// credit the credit card
public boolean
     credit(double amount)
{
   if(amount < 0)
      return false;

   visa.Balance -= amount;
   TotalBalance -= amount;

   return true;
}
```

`mc.credit(10)`

```
// credit the credit card
public boolean
     credit(double amount)
{
   if(amount < 0)
      return false;

   mc.Balance -= amount;
   TotalBalance -= amount;

   return true;
}
```

Nomenclature:

- Instance = “in an Object”
  - Has an Implicit Parameter / Argument
  - Instance Data = Data in an Object
  - Instance Code = Code that does things with an Object: “needs an object”
- Static = “Not in an Object”
  - Does Not have an Implicit Parameter / Argument
  - Static Data = Data in the Class (not an object), where the same copy of the data is accessible by all Code
  - Static Code = Code that doesn’t use an implicit parameter to access any Objects
- Example:


```
class Number
{
   final int InstanceNumber;

   static final int ClassNumber = 101;

   public Number(int n) { InstanceNumber = n; }

   public static void main(String[] args)
   {
      // define some numbers:
      Number n7  = new Number(7);
      Number n42 = new Number(42);

      System.out.println("n7  = " + n7.InstanceNumber);
      System.out.println("n42 = " + n42.InstanceNumber);
      System.out.println("ClassNumber = " + ClassNumber);
   }
}
```

Instance (Implicit Parameter)

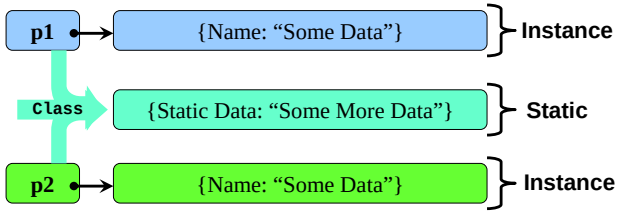
No Implicit Parameters

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We’re Done!

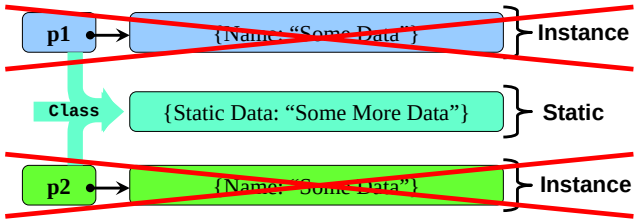
Regular Classes Look Like This:

- Classes have:
  - Instance Data (In the Objects)
  - Instance Code (Does things with Objects)
  - Static Data (Shared by All Objects)
  - Static Code (Only does things with static data)



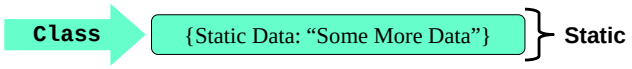
Static Utility Classes Revisited

- Utility Classes have:
  - Private Constructors
  - **No Objects**
  - Only Static Data and Code



Static Utility Classes Revisited

- Utility Classes have:
  - Private Constructors
  - **No Objects**
  - Only Static Data
  - Only Static Code

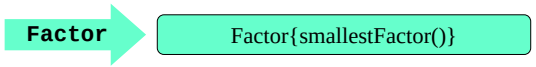


Java Notation for Utility Classes

- Because Utility Classes have No Objects, we have to access them through their class name

```
Factor.smallestFactor(int C)
```

- Which "looks" like this:



CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

Variable Hiding / Shadowing

- You can define a “Local Variable” or parameter to have the same name as a Class Data Member
- Why?
  - It’s confusing, so it’s a bad programming practice
- Example…


```
public class Hidden
{
   static int Variable = 10;

   public static void method1()
   {
      Variable = 100;
      System.out.println("in 1: " + Variable);
   }

   public static void method2()
   {
      int Variable = 200;
      System.out.println("in 2: " + Variable);
   }

   public static void method3(int Variable)
   {
      Variable = 300;
      System.out.println("in 3: " + Variable);
   }
```

Hidden Variable

Shadow Variables


```
public static void method4(int Variable)
{
   Variable = Variable;
   System.out.println("in 4: " + Variable);
}

public static void method5(int Variable)
{
   Hidden.Variable = Variable;
   System.out.println("in 5: " + Variable);
}
```



```
   public static void main(String[] args)
   {
      method1();
      System.out.println("main: " + Variable);
      method2();
      System.out.println("main: " + Variable);
      method3(1000);
      System.out.println("main: " + Variable);
      method4(2000);
      System.out.println("main: " + Variable);
      method5(3000);
      System.out.println("main: " + Variable);
   }
}
```

Output

```
in 1: 100
main: 100
in 2: 200
main: 100
in 3: 300
main: 100
in 4: 2000
main: 100
in 5: 3000
main: 3000
```

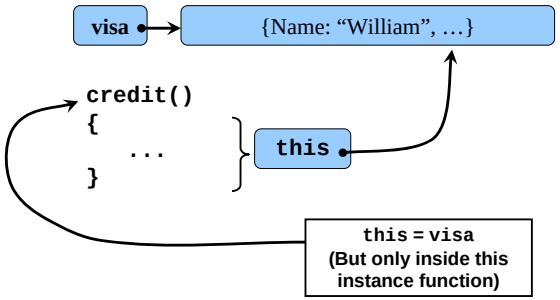
- Hidden variables are neat, but confusing, and can lead to hard-to-find bugs

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

`this`

- In instance code, the **`this`** variable is an alias for the name of our object



Why do we need **`this`**?

- Since we can easily directly refer to:
  - Instance Data (Data inside Objects)
  - Static Data (Data in the Class)

  why do we need **`this`**?

- **`this`** allows us to explicitly refer to Instance Data
  - Sometimes good for clarity
  - Solves Variable Hiding Problems
  - Solves Inheritance Problems

```
public class Hidden
{
   int Variable = 10;

   public void method1()
   {
      Variable = 100;
      System.out.println("in 1: " + Variable);
   }

   public void method2()
   {
      int Variable = 200;
      System.out.println("in 2: " + Variable);
   }

   public void method3(int Variable)
   {
      Variable = 300;
      System.out.println("in 3: " + Variable);
   }
```

This time it's an Instance Variable


```
public void method4(int Variable)
{
   Variable = Variable;
   System.out.println("in 4: " + Variable);
}

public void method5(int Variable)
{
   this.Variable = Variable;
   System.out.println("in 5: " + Variable);
}
```



```
   public static void main(String[] args)
     {
        Hidden h = new Hidden();
        h.method1();
        System.out.println("main: " + h.Variable);
        h.method2();
        System.out.println("main: " + h.Variable);
        h.method3(1000);
        System.out.println("main: " + h.Variable);
        h.method4(2000);
        System.out.println("main: " + h.Variable);
        h.method5(3000);
        System.out.println("main: " + h.Variable);
     }
  }
```


Output

```
in 1: 100
main: 100
in 2: 200
main: 100
in 3: 300
main: 100
in 4: 2000
main: 100
in 5: 3000
main: 3000
```

- Same output as before, same hiding of the variable **`Variable`**, even though it's an Instance variable this time.

`this` and Cool Variable Hiding?

```
public class Cool
{
   String Name;
   int    Age;

   public Cool(String Name, int Age)
   {
      this.Name = Name;
      this.Age  = Age;
   }

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

   ...  // rest of class
}
```


Annoying Overuse of `this`

```
public class NotCool
{
   String Name;
   static int CountNameChanges = 0;

   public NotCool(String Name, int Age)
   {
      this.Name = Name;
      NotCool.CountNameChanges++;
   }

   public void setName(String Name)
   {
      this.Name = Name;
      NotCool.CountNameChanges++;
   }

   // ...  rest of class
}
```


I Apologise if you like to code that way

- Some textbooks and profs recommend the explicit approach (this.var, class.var, for all references to Instance or Class variables)
- It makes explicitly clear which variables are instance or static
  - Although it is easier to accomplish this by variable name prefixing:
  - "Name" vs. "name", or "iName" vs. "sName"
- In the end, it takes a lot more typing to merely accomplish what Java does by default
  - (But it's great if you're getting paid by the character!)

CSE1030 – Lecture #6

- Review
- Static Data versus Instance Data
- Java Notation
- Static Utility Class Revisited
- Variable Hiding & Shadowing
- **`this`**
- We're Done!

Next topic…

Aggregation and Composition I