Chapter Four: Writing Classes

CTEC 150, Fall 2019

Jay Jain

Bossier Parish Community College

jjain@bpcc.edu

September 2019

4.1: Classes and Objects Revisited

Classes and Objects Revisited

- The goal of this chapter is to start designing and implementing our own classes to suit our specific needs

Classes and Objects Revisited

- The goal of this chapter is to start designing and implementing our own classes to suit our specific needs
- A class is a blueprint of an object

Classes and Objects Revisited

- The goal of this chapter is to start designing and implementing our own classes to suit our specific needs
- A class is a blueprint of an object
- An object is an instance of a class

Classes and Objects Revisited

- The goal of this chapter is to start designing and implementing our own classes to suit our specific needs
- A class is a blueprint of an object
- An object is an instance of a class
- An object has *state* which is defiend by values called attributes associated with that particular object

Classes and Objects Revisited

- The goal of this chapter is to start designing and implementing our own classes to suit our specific needs
- A class is a blueprint of an object
- An object is an instance of a class
- An object has *state* which is defiend by values called attributes associated with that particular object
- An object also has behaviors which are defined by the operations associated with that object.

4.2: Anatomy of a Class

Writing Classes

- The class that contains the `main` method is the starting point of the program

Writing Classes

- The class that contains the `main` method is the starting point of the program
- Remember that an object has *state* and *behavior*

Writing Classes

- The class that contains the `main` method is the starting point of the program
- Remember that an object has *state* and *behavior*
- Classes contain data declarations and method declarations

Writing Classes

- The class that contains the `main` method is the starting point of the program
- Remember that an object has *state* and *behavior*
- Classes contain data declarations and method declarations
- The data (attributes) define the state of the object

Writing Classes

- The class that contains the `main` method is the starting point of the program
- Remember that an object has *state* and *behavior*
- Classes contain data declarations and method declarations
- The data (attributes) define the state of the object
- The methods (functions) define the behavior of the object

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

toString method

- The toString method returns a character string that represents the object in some way

Example

```
String test = "hello";
System.out.println(test); // The toString() method from
// the String class is actually being called
```

toString method

- The `toString` method returns a character string that represents the object in some way
- Called automatically when an object is concatenated to a string or when an object is passed to the `println()` method

Example

```
String test = "hello";
System.out.println(test); // The toString() method from
// the String class is actually being called
```

toString method

- The `toString` method returns a character string that represents the object in some way
- Called automatically when an object is concatenated to a string or when an object is passed to the `println()` method

Example

```
String test = "hello";
System.out.println(test); // The toString() method from
// the String class is actually being called
```

```
//
public String toString()
{
    String result = Integer.toString(faceValue);

    return result;
}
```

Constructor

```
//*******************************************************************
//  Die.java        Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//*******************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

Data Scope

- The scope of the data is the area in a program in which data is referenced (used)

Data Scope

- The scope of the data is the area in a program in which data is referenced (used)
- Data declared at the class level can be referenced by all methods in that class

Data Scope

- The scope of the data is the area in a program in which data is referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can only be used in that method - this is called <u>local data</u>

Data Scope

- The scope of the data is the area in a program in which data is referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can only be used in that method - this is called local data

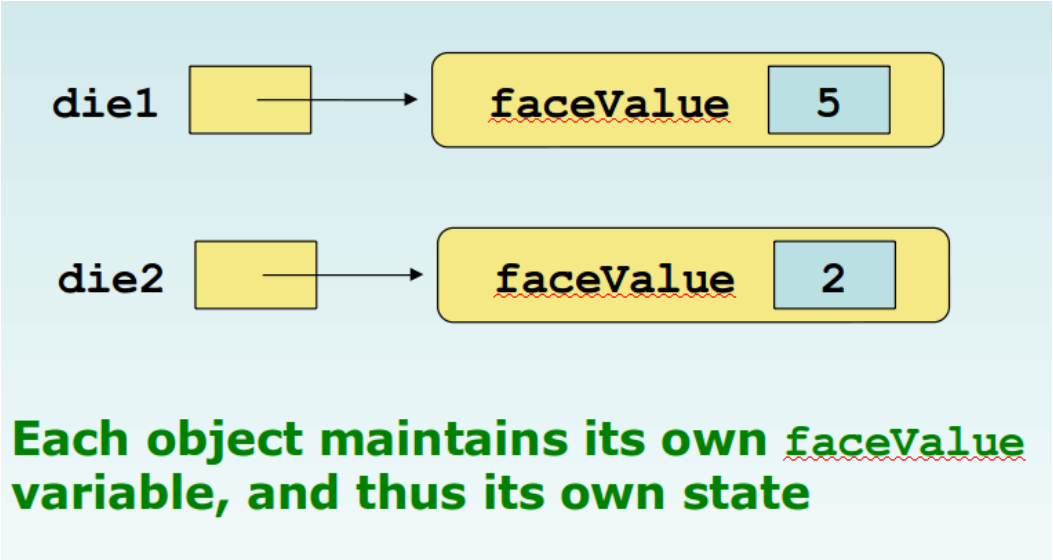
Local Data within main method

```
public static void main(String[] args){
    String dog = "woof";
}

public static void printDog(){
    String dog = "woof";
    System.out.println(dog);
}
```

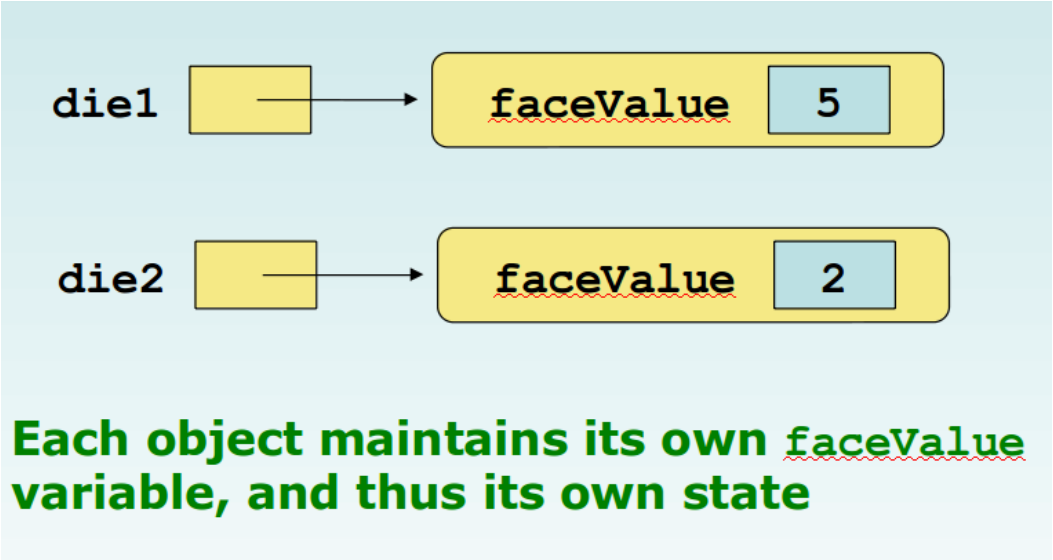
Instance Data

- A variable declared at the class level is instance data



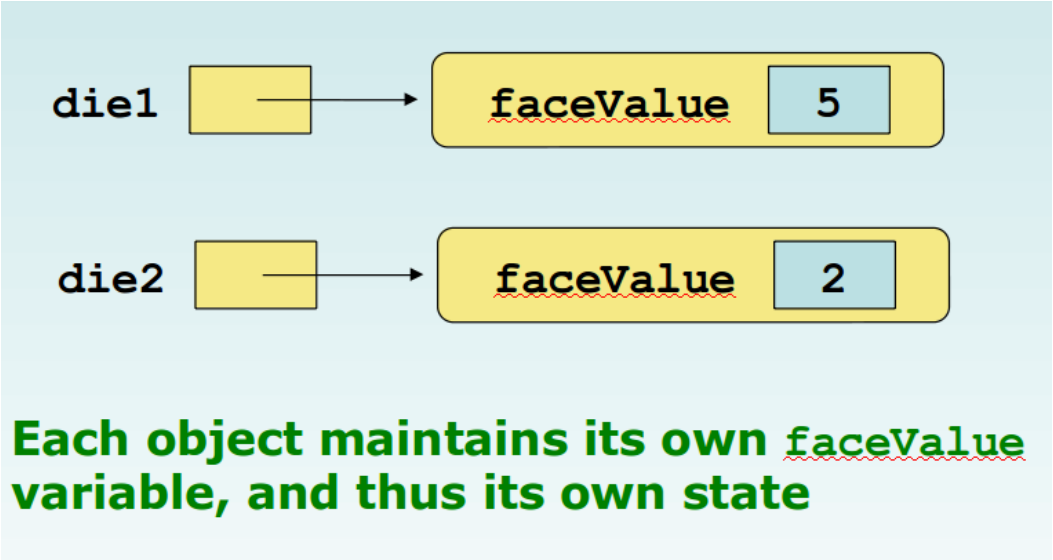
Instance Data

- A variable declared at the class level is instance data
- Each object has its own instance variables



Instance Data

- A variable declared at the class level is instance data
- Each object has its own instance variables
- The objects of a class share the method definitions, but each object has its own data space



UML Class Diagrams

- Unified Modeling Language (UML)

UML Class Diagrams

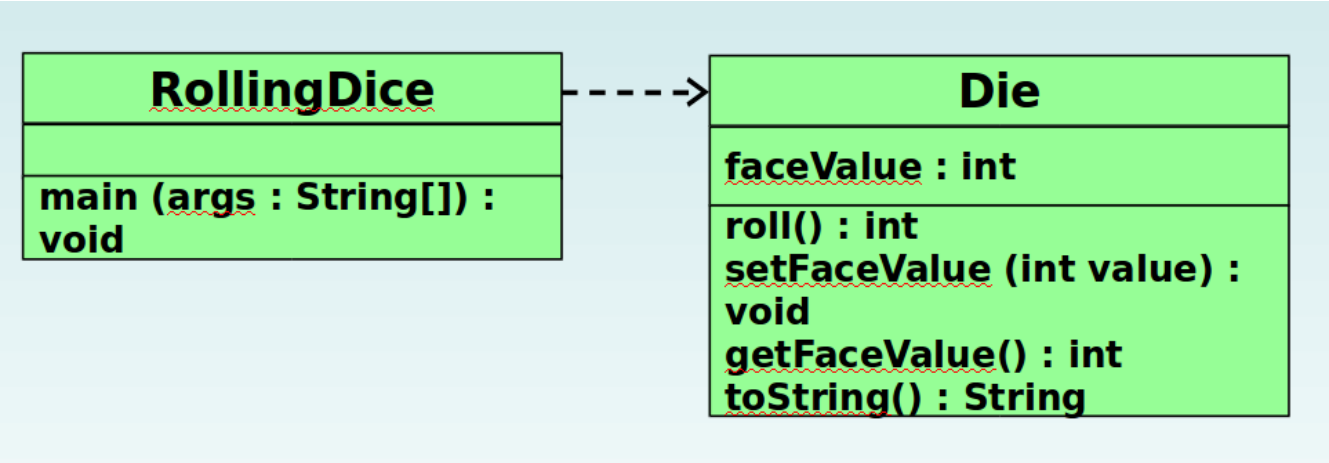
- Unified Modeling Language (UML)
- Show relationships among classes

UML Class Diagrams

- Unified Modeling Language (UML)
- Show relationships among classes
- Contain class name, attributes (variables), operations (methods)

UML Class Diagrams

- Unified Modeling Language (UML)
- Show relationships among classes
- Contain class name, attributes (variables), operations (methods)



4.3: Encapsulation

Encapsulation

- Two Views of an object

Encapsulation

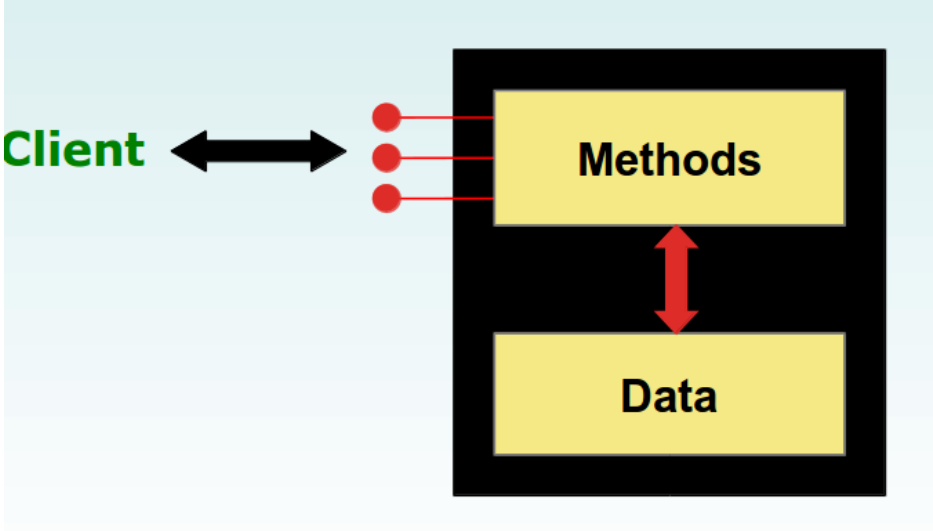
- Two Views of an object
- Internal: details of variables and methods of class that defines it

Encapsulation

- Two Views of an object
- Internal: details of variables and methods of class that defines it
- External: The services that an object provides and how the object interacts with rest of system

Encapsulation

- Two Views of an object
- Internal: details of variables and methods of class that defines it
- External: The services that an object provides and how the object interacts with rest of system
- We should make it difficult, if not impossible for a client to access an object's variables directly



Black Box Model: Client can call methods, but doesn't see code logic

Visibility Modifiers

- Encapsulation is done with visibility modifiers

Visibility Modifiers

- Encapsulation is done with visibility modifiers
- `public, protected, private`

Visibility Modifiers

- Encapsulation is done with visibility modifiers
- `public, protected, private`
- `final` describes constants

Visibility Modifiers

- Encapsulation is done with visibility modifiers
- `public, protected, private`
- `final` describes constants
- `public` variables can be referenced anywhere; violates encapsulation

Visibility Modifiers

- Encapsulation is done with visibility modifiers
- `public, protected, private`
- `final` describes constants
- `public` variables can be referenced anywhere; violates encapsulation
- Instance variables should not be public; they should be private

Visibility Modifiers

- Encapsulation is done with visibility modifiers
- `public, protected, private`
- `final` describes constants
- `public` variables can be referenced anywhere; violates encapsulation
- Instance variables should not be public; they should be private
- `private` can be referenced only within the class
- `protected` can be referenced within the same package

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- An accessor method returns current value of variable; `getX()`

Accessors and Mutators

- An accessor method returns current value of variable; `getX()`
- A mutator method changes the value of a variable; `setX()`

Accessors and Mutators

- An accessor method returns current value of variable; `getX()`
- A mutator method changes the value of a variable; `setX()`
- Known as getter and setter methods

Accessors and Mutators

- An accessor method returns current value of variable; `getX()`
- A mutator method changes the value of a variable; `setX()`
- Known as getter and setter methods
- A getter method retrieves information about the state of the object

Accessors and Mutators

- An accessor method returns current value of variable; `getX()`
- A mutator method changes the value of a variable; `setX()`
- Known as getter and setter methods
- A getter method retrieves information about the state of the object
- A setter method updates information about the state of the object

4.4: Anatomy of a Method

Method Declarations

- A method declaration specifies the code that will be executed when the method is invoked (called)

Method Declarations

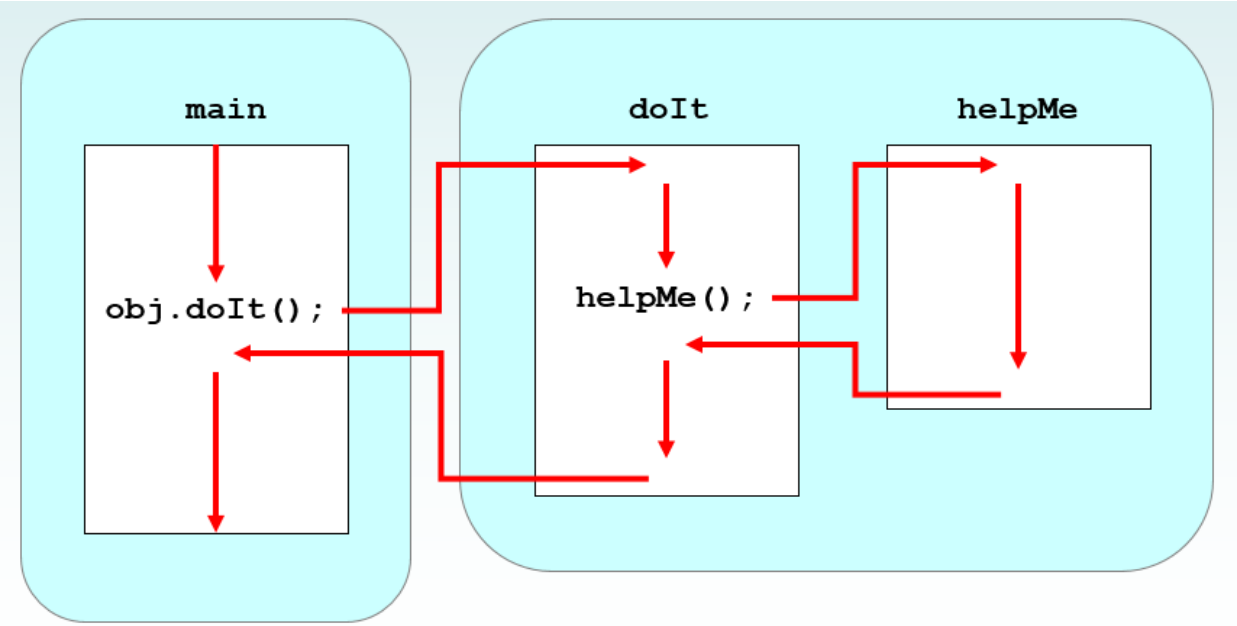
- A method declaration specifies the code that will be executed when the method is invoked (called)
- When method is invoked, flow of control goes to that method and executes its code

Method Declarations

- A method declaration specifies the code that will be executed when the method is invoked (called)
- When method is invoked, flow of control goes to that method and executes its code
- After method code is executed, flow of control goes back to the place where the method was originally called

Method Declarations

- A method declaration specifies the code that will be executed when the method is invoked (called)
- When method is invoked, flow of control goes to that method and executes its code
- After method code is executed, flow of control goes back to the place where the method was originally called



Method Header

```
char calc(int num1, int num2, String message)
```

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

The return expression must be consistent with the return type

sum and result are local data

They are created each time the method is called, and are destroyed when it finishes executing

The `return` Statement

- The `return` type of a method indicates the type of value that the method sends back to the original calling location

The `return` Statement

- The `return` type of a method indicates the type of value that the method sends back to the original calling location
- A method does not return a value if it has a `void` return type

The return Statement

- The return type of a method indicates the type of value that the method sends back to the original calling location
- A method does not return a value if it has a void return type

Examples

```
public static void main(String[])args{
    System.out.println("Hello World");
    // Print statements are different than return
}

public int getBalance(){
    return 5;
}
```

Parameters

- When a method is called, the actual parameters in the invocation are copied into the formal parameters of the method head.

Parameters

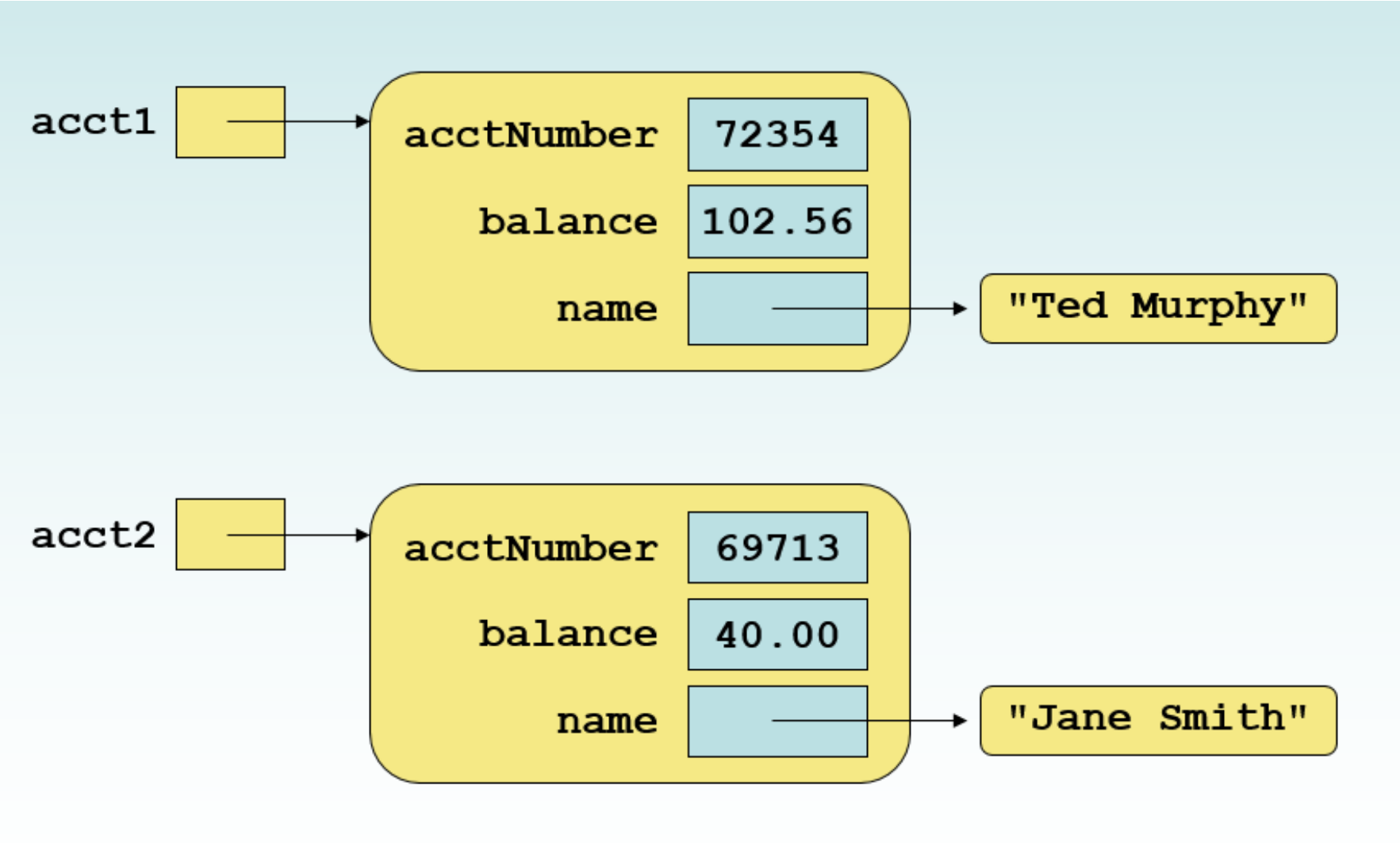
- When a method is called, the actual parameters in the invocation are copied into the formal parameters of the method head.

```
ch = obj.calc(25, count, "Hello");
```

```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

Method Body


acct1
acctNumber
72354
balance
102.56
name
"Ted Murphy"
acct2
acctNumber
69713
balance
40.00
name
"Jane Smith"

4.5: Constructors Revisited

Constructors Revisited

- Constructors have no return type; not even `void`

Example

```
// Default constructor
public Account(){
}

// Constructor with parameters
public Account(String name, int balance){
    name = this.name;
    balance = this.balance;
}
```

Constructors Revisited

- Constructors have no return type; not even `void`
- So, don't put a return type on a constuctor; that would make it a method

Example

```
// Default constructor
public Account(){
}

// Constructor with parameters
public Account(String name, int balance){
    name = this.name;
    balance = this.balance;
}
```

Constructors Revisited

- Constructors have no return type; not even `void`
- So, don't put a return type on a constuctor; that would make it a method
- Each class has a default constructor that accepts no parameters

Example

```
// Default constructor
public Account(){
}

// Constructor with parameters
public Account(String name, int balance){
    name = this.name;
    balance = this.balance;
}
```

QUESTIONS ???

Somewhere, something incredible is waiting to be known.

- Carl Sagan