



# *CLASS DESIGN I*

CIS\*2430 (Fall 2010)

# A Class is a Type

- A class is a programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or an instance of the class
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

# Primitive vs. Class Types

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

# Die Class

```
public class Die
{
    private int maxFaces;
    private int faceValue;

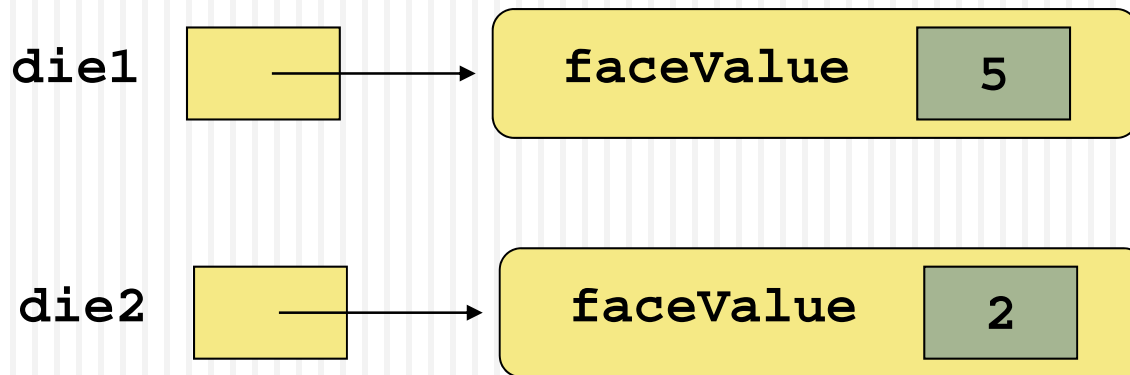
    public int roll()
    { //code to roll the die here }

    public int getFaceValue()
    { //code to get current face value here }

    public void setFaceValue(int value)
    { // code to set current value here }
}
```

# Instance Data/Variables

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) has its own memory for it



Each object maintains its own *faceValue* variable, and thus its own state

# Methods

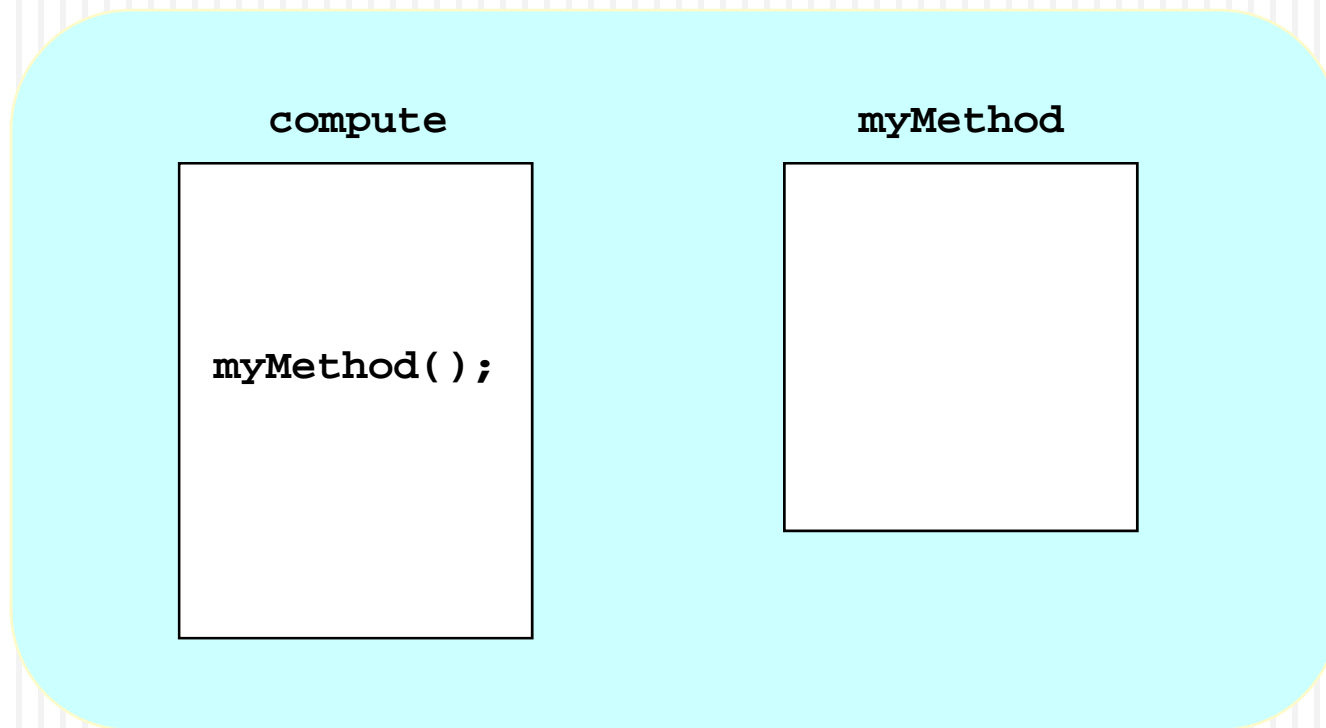
- Method definitions are divided into a *header* and a *body*:

```
public void myMethod()           Header
{
    // code to perform some action  Body
    ...
}
```

- Methods are invoked using the name of the calling object and the method name as follows:  
`classVar.myMethod( );`
- Invoking a method is equivalent to executing the method body

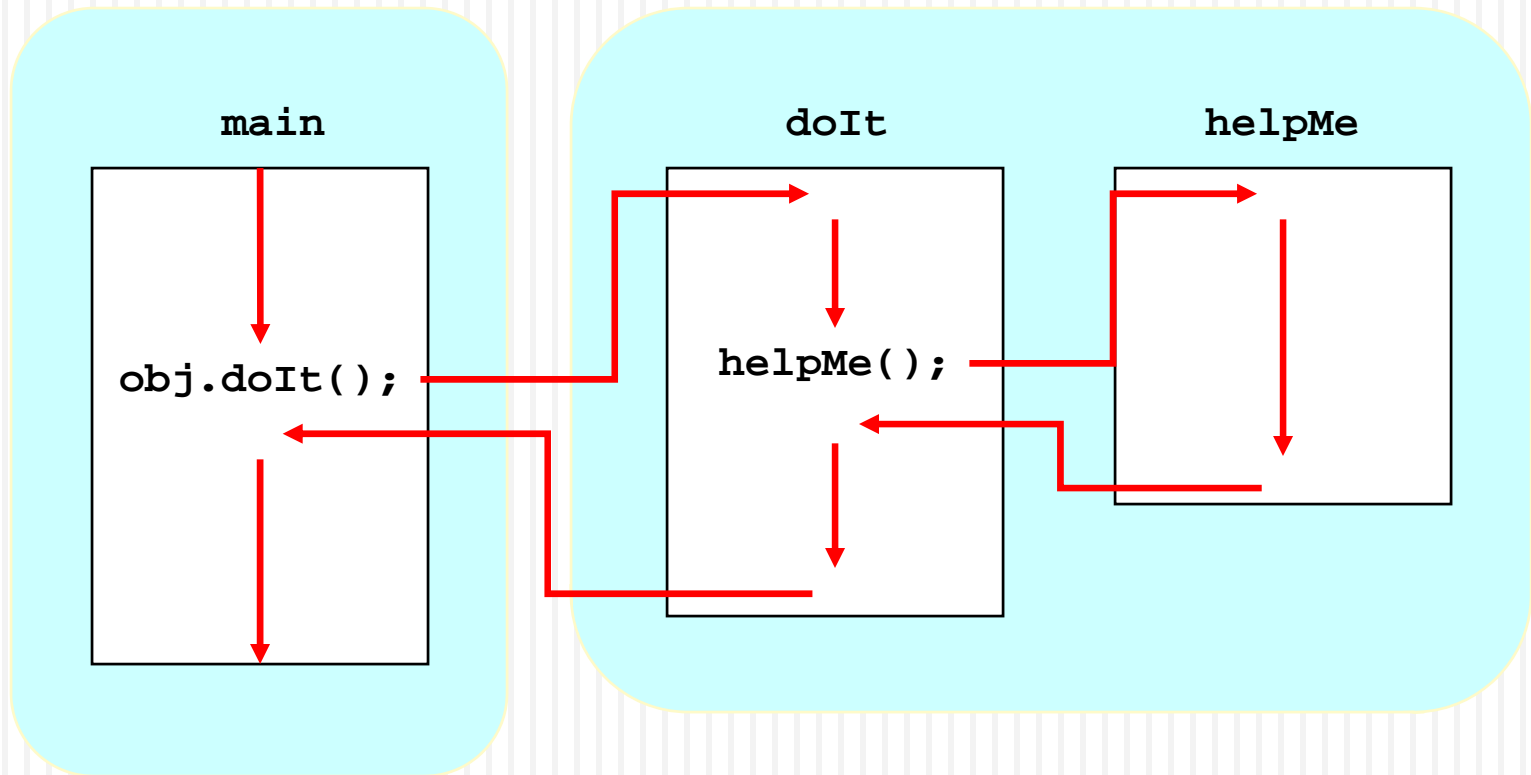
# Calling Methods

- If the called method is in the same class, only the method name is needed



# Calling Methods

- Typically the called method is in another class or object





# Method Header

- A method declaration begins with a *method header*

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

visibility      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

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

    return result;
}
```

sum and result  
are local variables

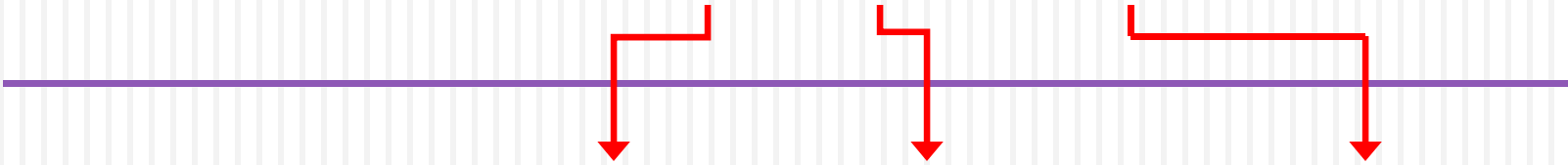
The return expression  
must be consistent with  
the return type

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

# Parameters

- The number, order, and type of the arguments or actual parameters must be compatible with those of the formal parameters:

```
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;
}
```

# Return statements

- A **return** statement specifies the value returned and ends the method invocation:  
**return Expression;**
  - **Expression** evaluates to something of the type listed in the method header
- A **void** method doesn't really need a **return** statement, but can be terminated early with an empty return statement:  
**return;**

# Invoking Methods

- An invocation of a method can be used as part of an expression:

```
typeReturned tRVariable;  
tRVariable = objectName.methodName( );
```

- We can just call the method and ignore the type  
`objectName.methodName( );`

- An invocation of a **void** method is simply a statement:

```
objectName.methodName( );
```

# Local Variables

- A variable declared within a method definition is called a *local variable*
  - All method parameters are also local variables
- If two methods each have a local variable of the same name, they are still two entirely different variables
- Java does not have global variables.

# Blocks

- A *block* is another name for a compound statement, which contains a set of statements enclosed in braces `{ }`
- A variable declared within a block is local to that block, and cannot be used outside the block
- Once a variable has been declared within a block, its name cannot be used for anything else within the same block

# Variables in for Statements

- You can declare one or more variables within the initialization portion of a **for** statement
- A variable so declared will be local to the **for** loop, and cannot be used outside of the loop
- If you need to use such a variable outside of a loop, then declare it outside the loop



# Parameters of Primitive Types

- For parameters of primitive types, the values of the arguments are copied to the variables for the formal parameters
  - Known as call-by-value mechanism
- If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion:

**byte→short→int→long→float→double**

# Parameters of Primitive Types

- A parameter is essentially a local variable
- When a method is invoked, the value of its argument is computed, and the corresponding parameter is initialized to this value
- Even if the value of a formal parameter is changed within a method, the value of the argument cannot be changed

# Bill Class (1 of 5)

## Display 4.6 A Formal Parameter Used as a Local Variable

---

*This is the file Bill.java.*

```
1  import java.util.Scanner;

2  public class Bill
3  {
4      public static double RATE = 150.00; //Dollars per quarter hour

5      private int hours;
6      private int minutes;
7      private double fee;
```

(continued)

# Bill Class (2 of 5)

## Display 4.6 A Formal Parameter Used as a Local Variable

```
8      public void inputTimeWorked()
9      {
10         System.out.println("Enter number of full hours worked");
11         System.out.println("followed by number of minutes:");
12         Scanner keyboard = new Scanner(System.in);
13         hours = keyboard.nextInt();
14         minutes = keyboard.nextInt();
15     }

16     public double computeFee(int hoursWorked, int minutesWorked)
17     {
18         minutesWorked = hoursWorked*60 + minutesWorked;
19         int quarterHours = minutesWorked/15; //Any remaining fraction of a
20                                             // quarter hour is not charged for.
21         return quarterHours*RATE;
22     }

23     public void updateFee()
24     {
25         fee = computeFee(hours, minutes);
26     }
```

*computeFee uses the parameter minutesWorked as a local variable.*

*Although minutes is plugged in for minutesWorked and minutesWorked is changed, the value of minutes is not changed.*

(continued)

# Bill Class (3 of 5)

## Display 4.6 A Formal Parameter Used as a Local Variable

---

```
27     public void outputBill()  
28     {  
29         System.out.println("Time worked: ");  
30         System.out.println(hours + " hours and " + minutes + " minutes");  
31         System.out.println("Rate: $" + RATE + " per quarter hour.");  
32         System.out.println("Amount due: $" + fee);  
33     }  
34 }
```

(continued)

# Bill Class (4 of 5)

## Display 4.6 A Formal Parameter Used as a Local Variable

---

```
1  public class BillingDialog
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Welcome to the law offices of");
6          System.out.println("Dewey, Cheatham, and Howe.");
7          Bill yourBill = new Bill();
8          yourBill.inputTimeWorked();
9          yourBill.updateFee();
10         yourBill.outputBill();
11         System.out.println("We have placed a lien on your house.");
12         System.out.println("It has been our pleasure to serve you.");
13     }
14 }
```

*This is the file BillingDialog.java.*

(continued)

# Bill Class (5 of 5)

## Display 4.6 A Formal Parameter Used as a Local Variable

---

### SAMPLE DIALOGUE

Welcome to the law offices of  
Dewey, Cheatham, and Howe.  
Enter number of full hours worked  
followed by number of minutes:  
**3 48**  
Time worked:  
2 hours and 48 minutes  
Rate: \$150.0 per quarter hour.  
Amount due: \$2250.0  
We have placed a lien on your house.  
It has been our pleasure to serve you.

# equal and toString Methods

- Java expects certain methods, such as **equals** and **toString**, to be in almost all classes
- The purpose of **equals**, a **boolean** valued method, is to compare two objects of the class to see if they are “equal”

```
public boolean equals(ClassName  
    objectName)
```

- The purpose of **toString** is to return a **String** value that represents the data in the object
- ```
public String toString()
```



# Information Hiding/Encapsulation

- The modifier **public** means that there are no restrictions on where an instance variable or method can be used
- The modifier **private** means that an instance variable or method can only be accessed within the class
- It is considered good programming practice to make **all** instance variables **private**
- Most methods are **public**, and thus provide controlled access to the object
- **private** methods are usually used as helping methods for other methods in the class

# Accessors and Mutators

- Accessor methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **get**
- *Mutator* methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **set**

# Mutators with boolean Returns

- Some mutator methods issue an error message and end the program whenever they are given invalid values
- Alternatively, we can have the mutator test the values and return a boolean value. The calling program will handle the cases where the input values do not make sense

# Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

# Issues with Overloading

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

# Invalid Overloading

- The signature of a method only includes the method name and its parameter types
  - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class
- Unlike C++, Java does not allow operator overloading

# Constructors

- A *constructor* is a special method used to set up an object after its initial creation
- A constructor is a method that has the same name as the class and no return type/value
- A constructor is typically overloaded

```
public Die()  
{  
    maxFaces = 6;  
    faceValue = 1;  
}
```

# Constructors

- A constructor is called when a new object of the class is created:

```
ClassName objectName = new  
    ClassName( anyArgs );
```

- This is the *only* valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again for the same variable, the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead



# No-Argument Constructors

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, be sure to provide your own no-argument constructor as well

# Default Initializations

- Instance variables are automatically initialized in Java: **boolean** to false, other primitives to 0, and class types to **null**
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Local variables are not automatically initialized

# this Parameter

- All instance variables and methods are understood to have `<the calling object>.` in front of them
- If an explicit name for the calling object is needed, the keyword `this` can be used
  - `myInstanceVariable` always means and is always interchangeable with `this.myInstanceVariable`

# this Parameter

- **this** must be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

↑  
local

↑  
instance

# ArrayList Class

- Unlike arrays, which have a fixed length once created, an **ArrayList** is an object that can grow and shrink at the running time
- It must be imported from the package **java.util**
- The base type of an ArrayList is specified as a *type parameter* :  

```
ArrayList<BaseType> aList =  
    new ArrayList<BaseType>( );
```

# Using ArrayList Class

- An initial capacity can be specified when creating an **ArrayList**
  - The following code creates an **ArrayList** that stores objects of the base type **String** with an initial capacity of 20 items

```
ArrayList<String> list =  
    new ArrayList<String>(20);
```
  - Specifying an initial capacity does not limit the size to which an **ArrayList** can eventually grow

# Using ArrayList Class

- The **add** method is used to set an element for the first time in an **ArrayList**

```
list.add("something");
```

- The **size** method is used to find out how many indices already have elements in the **ArrayList**

```
int howMany = list.size();
```

- The **set** method is used to replace any existing element, and the **get** method is used to access the value of any existing element

```
list.set(index, "something else");
```

```
String thing = list.get(index);
```

# API for ArrayList

## Display 14.1 Some Methods in the Class ArrayList

### ARRAYLIKE METHODS

```
public Base_Type set( int index, Base_Type newElement)
```

Sets the element at the specified `index` to `newElement`. Returns the element previously at that position, but the method is often used as if it were a void method. If you draw an analogy between the `ArrayList` and an array `a`, this statement is analogous to setting `a[index]` to the value `newElement`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws an `IndexOutOfBoundsException` if the `index` is not in this range.

```
public Base_Type get(int index)
```

Returns the element at the specified `index`. This statement is analogous to returning `a[index]` for an array `a`. The `index` must be a value greater than or equal to 0 and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range.

(continued)



# API for ArrayList

## Display 14.1 Some Methods in the Class ArrayList

### METHODS TO ADD ELEMENTS

```
public boolean add(Base_Type newElement)
```

Adds the specified element to the end of the calling ArrayList and increases the ArrayList's size by one. The capacity of the ArrayList is increased if that is required. Returns `true` if the add was successful. (The return type is `boolean`, but the method is typically used as if it were a void method.)

```
public void add( int index, Base_Type newElement)
```

Inserts `newElement` as an element in the calling ArrayList at the specified index. Each element in the ArrayList with an index greater or equal to `index` is shifted upward to have an index that is one greater than the value it had previously. The `index` must be a value greater than or equal to 0 and less than *or equal* to the current size of the ArrayList. Throws `IndexOutOfBoundsException` if the index is not in this range. Note that you can use this method to add an element after the last element. The capacity of the ArrayList is increased if that is required.

(continued)

# API for ArrayList

## Display 14.1 Some Methods in the Class ArrayList

---

### MEMORY MANAGEMENT (SIZE AND CAPACITY)

```
public boolean isEmpty()
```

Returns true if the calling ArrayList is empty (that is, has size 0); otherwise, returns false.

(continued)

## Display 14.1 Some Methods in the Class ArrayList

---

### METHODS TO REMOVE ELEMENTS

```
public Base_Type remove(int index)
```

Deletes and returns the element at the specified index. Each element in the ArrayList with an index greater than index is decreased to have an index that is one less than the value it had previously. The index must be a value greater than or equal to 0 and less than the current size of the ArrayList. Throws `IndexOutOfBoundsException` if the index is not in this range. Often used as if it were a void method.

(continued)

# API for ArrayList

## Display 14.1 Some Methods in the Class ArrayList

### SEARCH METHODS

```
public boolean contains(Object target)
```

Returns true if the calling ArrayList contains target; otherwise, returns false. Uses the method equals of the object target to test for equality with any element in the calling ArrayList.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to target. Uses the method equals of the object target to test for equality. Returns -1 if target is not found.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to target. Uses the method equals of the object target to test for equality. Returns -1 if target is not found.

(continued)

# API for ArrayList

## Display 14.1 Some Methods in the Class ArrayList

---

```
public int size()
```

Returns the number of elements in the calling ArrayList.

```
public void ensureCapacity(int newCapacity)
```

Increases the capacity of the calling ArrayList, if necessary, in order to ensure that the ArrayList can hold at least newCapacity elements. Using ensureCapacity can sometimes increase efficiency, but its use is not needed for any other reason.

```
public void trimToSize()
```

Trims the capacity of the calling ArrayList to the ArrayList's current size. This method is used to save storage space.

(continued)