Classes, Encapsulation, Methods and Constructors (Continued)

- Class definitions
- Instance data
- Encapsulation and Java modifiers
- Method declaration and parameter passing
- Constructors
- Method Overloading
- Reading for this lecture: L&L, 4.1-4.5 & App E
Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (called).
- When a method is invoked, the flow of control jumps to the method and executes its code.
- When complete, the flow returns to the place where the method was called and continues.
- The invocation may or may not return a value, depending on how the method is defined.
Method Control Flow

• If the called method is in the same class, only the method name is needed
Method Control Flow

- The called method is often part of another class or object
A method declaration begins with a *method header*

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

- **Return type**: `char`
- **Method name**: `calc`
- **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

• The method header is followed by the *method body*

```java
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.
Local Data

• Local variables can be declared inside a method
• The formal parameters of a method are also *local variables* when the method is invoked
• When the method finishes, all local variables are destroyed (including the formal parameters)
• Keep in mind that instance variables, declared at the class/object level, exist for as long as the object exists
The return Statement

• The *return type* of a method indicates the type of value that the method sends back to the caller

• A method that does not return a value has a *void* return type

• A *return statement* specifies the value that will be returned upon completion of the method code
  
  ```
  return expression;
  ```

• Its expression must conform to the return type
Parameters

• When a method is called, the *actual parameters* in the call are copied into the *formal parameters* in the method header

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

ch = obj.calc (25, count, "Hello");
Objects as Parameters

- Another important issue related to method design involves parameter passing.

- Parameters in a Java method are passed by value.

- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header).

- Therefore passing parameters is similar to an assignment statement.

- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other.
Passing Objects to Methods

• What a method does with a parameter may or may not have a permanent effect (outside the method)

• See ParameterTester.java (page 333-334)
• See ParameterModifier.java (page 335)
• See Num.java (page 336)

• Note the difference between changing the internal state of an object versus changing the value of a reference to point to a different object
Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions.
- If a method is overloaded, the method name is not sufficient to determine which method is being called.
- The *signature* of each overloaded method must be unique.
- The signature includes the number, type, and order of the parameters.
Method Overloading

• The compiler determines which method is being invoked by analyzing the parameters

```java
float tryMe(int x) {
    return x + .375;
}

float tryMe(int x, float y) {
    return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```
Method Overloading

• The `println` method is overloaded:

  `println (String s)`
  `println (int i)`
  `println (double d)`

  and so on...

• The following lines invoke different versions of the `println` method:

  `System.out.println ("The total is:"es);`
  `System.out.println (3);`
Method Overloading

• The return type of the method is not part of the signature
• Overloaded methods cannot differ only by their return type
• Constructors can be overloaded and often are
• Overloaded constructors provide multiple ways to initialize a new object
Accessors and Mutators

- A class usually provides methods to indirectly access and modify the private data values.
- An *accessor method* returns the current value of a variable.
- A *mutator method* changes the value of a variable.
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value.
- They are sometimes called “getters” and “setters.”
Mutator Restrictions

• The use of mutators gives the class designer the ability to restrict a client’s options to modify an object’s state

• A mutator is often designed so that the values of variables can be set only within particular limits

• For example, the `setFaceValue` mutator of the `Die` class should restrict the value to the valid range (1 to `MAX`)