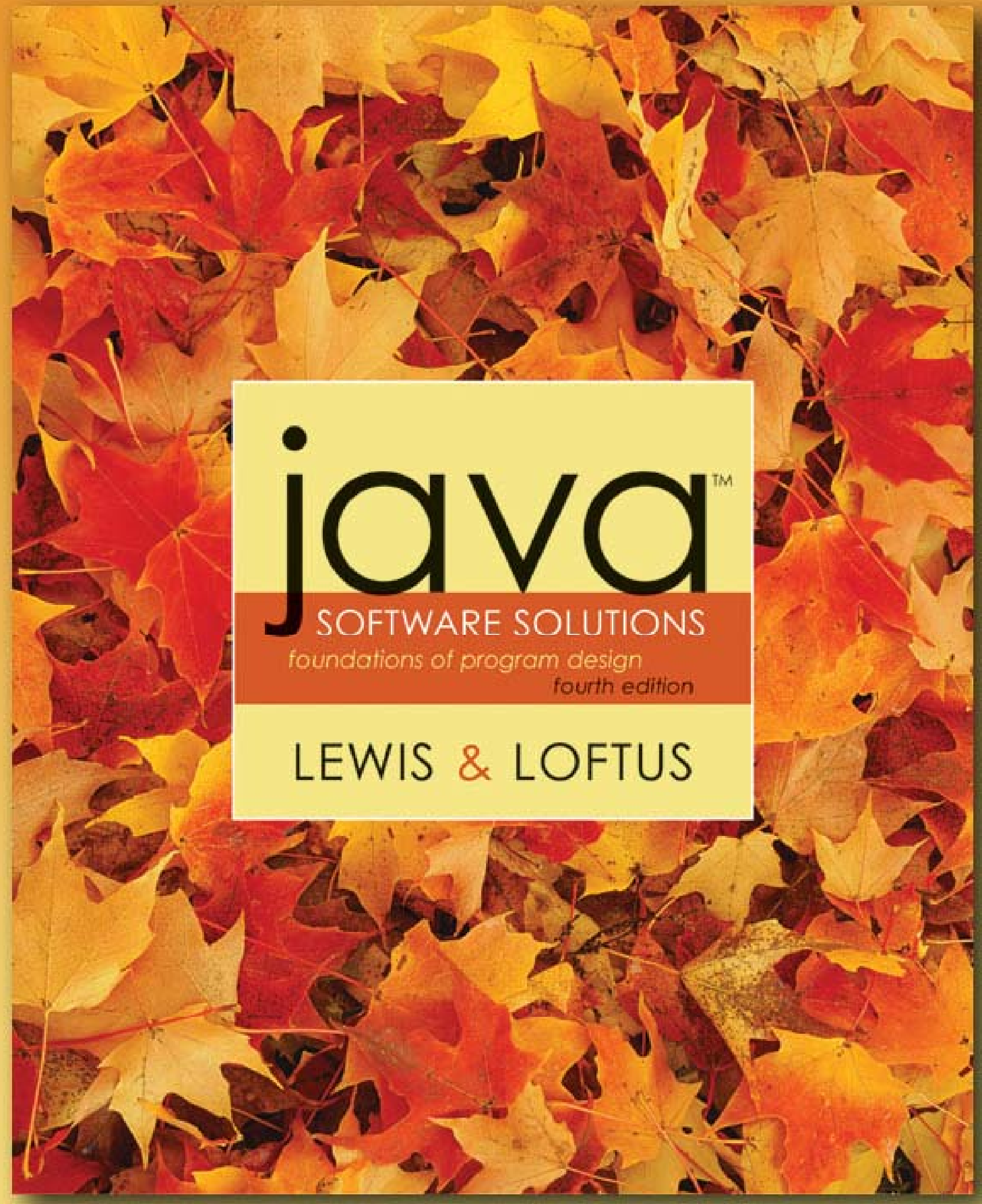
Lecture 4

Writing Classes



Writing Classes

- We've been using predefined classes. Now we will learn to write our own classes to define objects

- Chapter 4 focuses on:
  - class definitions
  - instance data
  - encapsulation and Java modifiers
  - method declaration and parameter passing
  - constructors

Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

Writing Classes

- The programs we've written in previous examples have used classes defined in the Java standard class library

- Now we will begin to design programs that rely on classes that we write ourselves

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

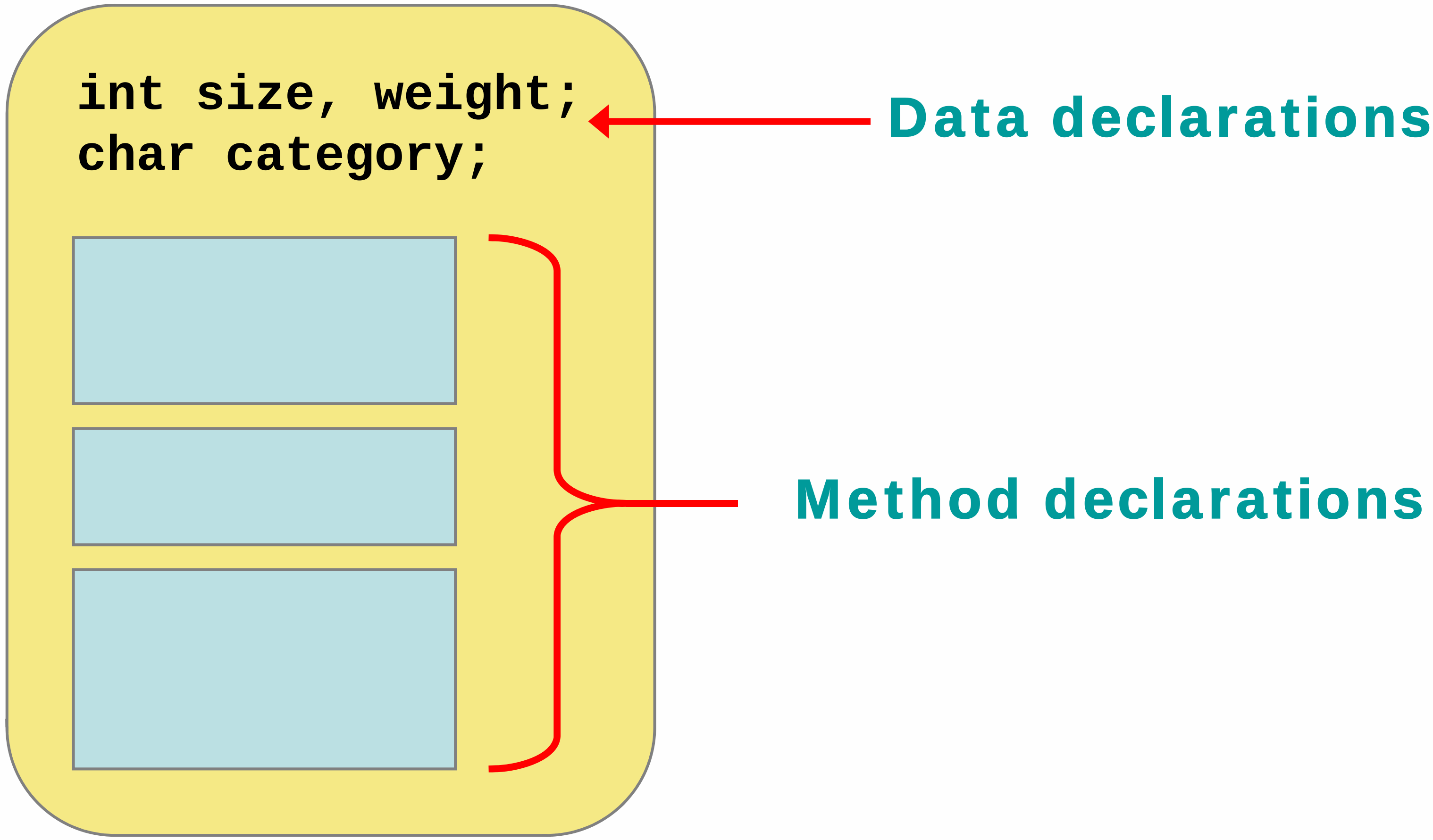
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Classes and Objects

- **Recall from our overview of objects in Chapter 1 that an object has *state* and *behavior***
- **Consider a six-sided die (singular of dice)**
  - **It's state can be defined as which face is showing**
  - **It's primary behavior is that it can be rolled**
- **We can represent a die in software by designing a class called `Die` that models this state and behavior**
  - **The class serves as the blueprint for a die object**
- **We can then instantiate as many die objects as we need for any particular program**

Classes

- A class can contain data declarations and method declarations



Classes

- **The values of the data define the state of an object created from the class**
- **The functionality of the methods define the behaviors of the object**
- **For our `Die` class, we might declare an integer that represents the current value showing on the face**
- **One of the methods would "roll" the die by setting that value to a random number between one and six**

Classes

- **We'll want to design the `Die` class with other data and methods to make it a versatile and reusable resource**

- **Any given program will not necessarily use all aspects of a given class**

- **See RollingDice.java (page 157)**
- **See Die.java (page 158)**

The Die Class

- **The `Die` class contains two data values**
  - **a constant `MAX` that represents the maximum face value**
  - **an integer `faceValue` that represents the current face value**
- **The `roll` method uses the `random` method of the `Math` class to determine a new face value**
- **There are also methods to explicitly set and retrieve the current face value at any time**

The toString Method

- All classes that represent objects should define a `toString` method
- The `toString` method returns a character string that represents the object in some way
- It is called automatically when an object is concatenated to a string or when it is passed to the `println` method

Constructors

- **As mentioned previously, a *constructor* is a special method that is used to set up an object when it is initially created**
- **A constructor has the same name as the class**
- **The `Die` constructor is used to set the initial face value of each new die object to one**
- **We examine constructors in more detail later in this chapter**

Data Scope

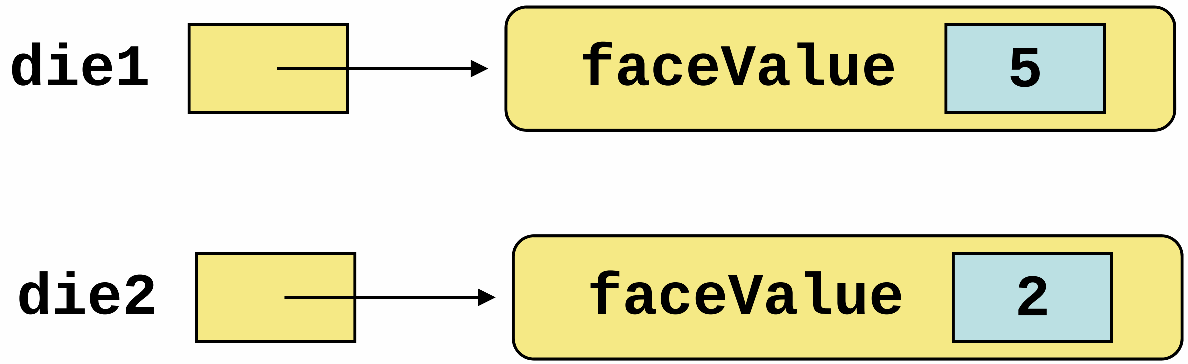
- **The *scope* of data is the area in a program in which that data can be referenced (used)**
- **Data declared at the class level can be referenced by all methods in that class**
- **Data declared within a method can be used only in that method**
- **Data declared within a method is called *local data***
- **In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else**

Instance Data

- **The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it**
- **A class declares the type of the data, but it does not reserve any memory space for it**
- **Every time a `Die` object is created, a new `faceValue` variable is created as well**
- **The objects of a class share the method definitions, but each object has its own data space**
- **That's the only way two objects can have different states**

Instance Data

- We can depict the two `Die` objects from the `RollingDice` program as follows:



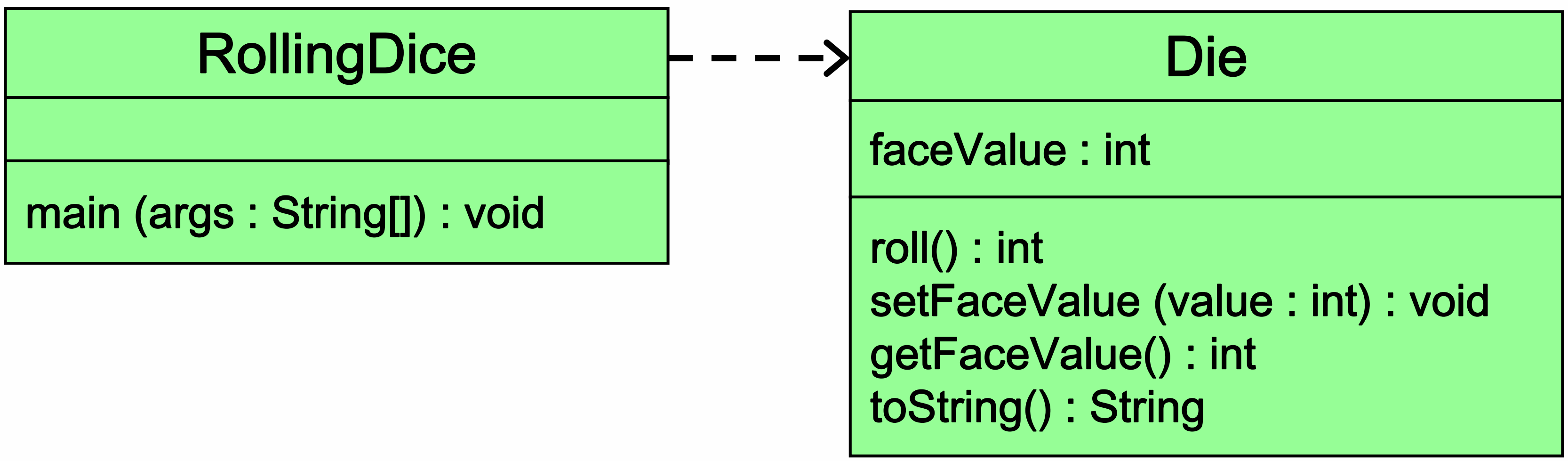
Each object maintains its own `faceValue` variable, and thus its own state

UML Diagrams

- **UML stands for the *Unified Modeling Language***
- ***UML diagrams* show relationships among classes and objects**
- **A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)**
- **Lines between classes represent *associations***
- **A dotted arrow shows that one class *uses* the other (calls its methods)**

UML Class Diagrams

- **A UML class diagram for the `RollingDice` program:**



Outline

Anatomy of a Class

Encapsulation

Anatomy of a Method

Encapsulation

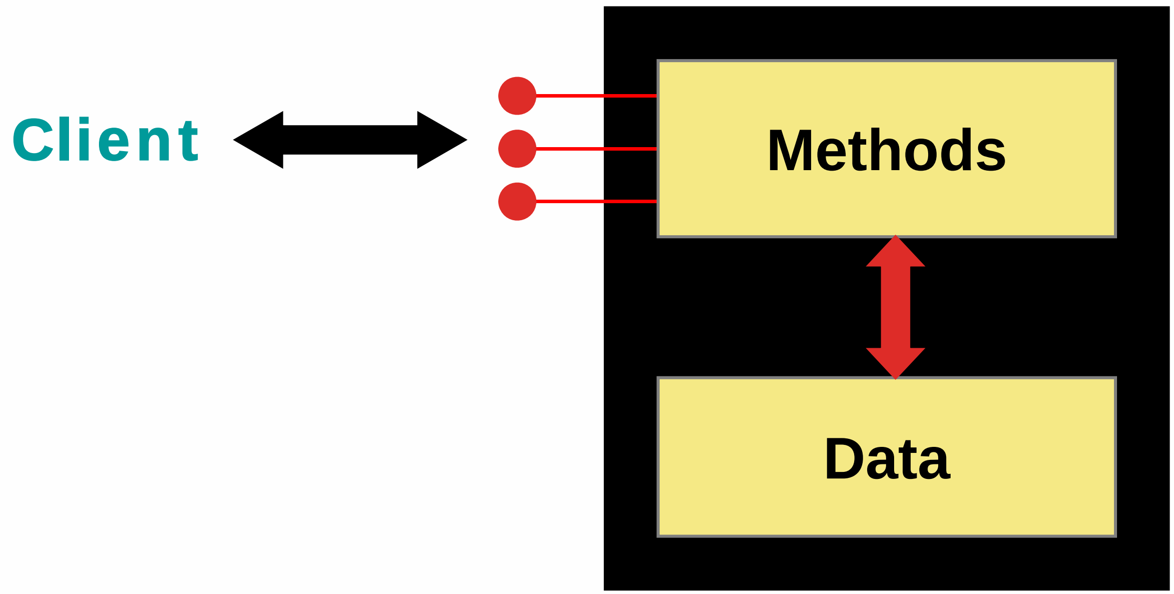
- **We can take one of two views of an object:**
  - **internal - the details of the variables and methods of the class that defines it**
  - **external - the services that an object provides and how the object interacts with the rest of the system**
- **From the external view, an object is an *encapsulated* entity, providing a set of specific services**
- **These services define the *interface* to the object**

Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object’s variables directly
- That is, an object should be *self-governing*

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data
- We've used the `final` modifier to define constants
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers

- **Members of a class that are declared with *public visibility* can be referenced anywhere**
- **Members of a class that are declared with *private visibility* can be referenced only within that class**
- **Members declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package**
- **An overview of all Java modifiers is presented in Appendix E**

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly
- Therefore instance variables should not be declared with public visibility
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by a client, it should not be declared with public visibility

Visibility Modifiers

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

Accessors and Mutators

- **Because instance data is private, a class usually provides services to access and modify 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 have restricted the value to the valid range (1 to `MAX`)**
- **We'll see in Chapter 5 how such restrictions can be implemented**

Outline

Anatomy of a Class

Encapsulation

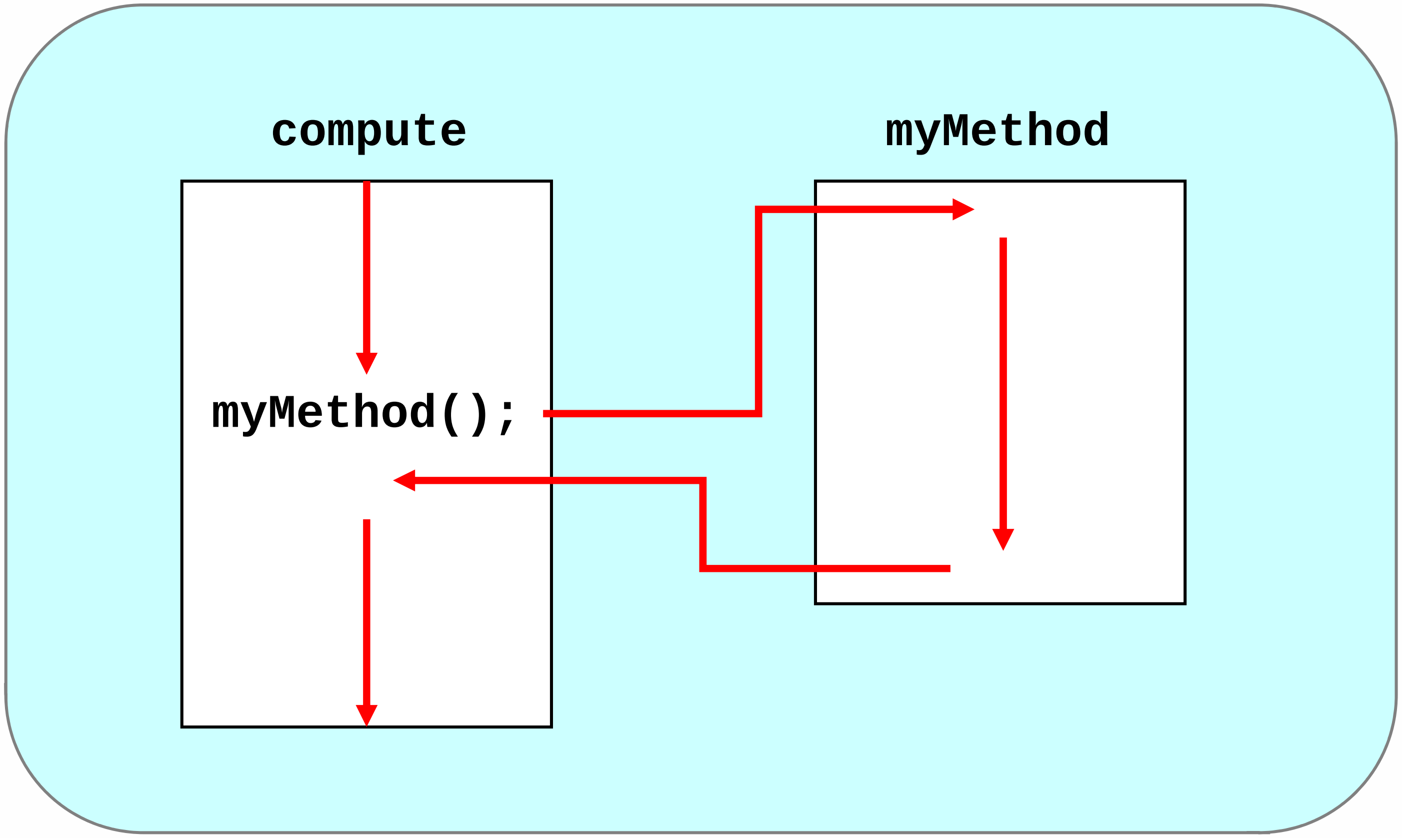
Anatomy of a Method

Method Declarations

- Let’s now examine method declarations in more detail
- 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

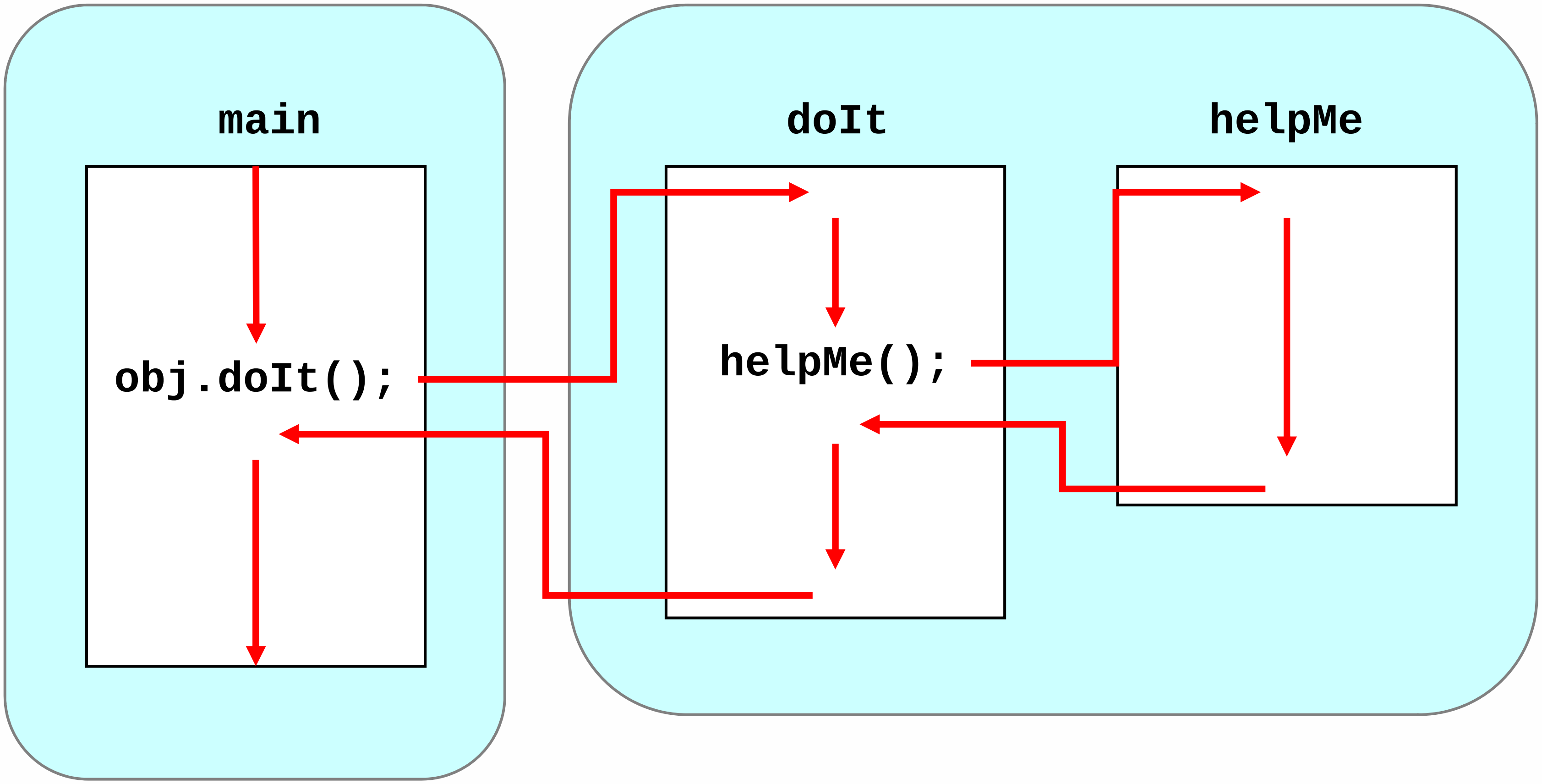
Method Control Flow

- **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**



Method Header

- A method declaration begins with a *method header*

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

method name

parameter list

return type

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

- The method header is followed by the *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 calling location
- A method that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type

Parameters

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

```
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;
}
```

Local Data

- As we've seen, local variables can be declared inside a method
- The formal parameters of a method create *automatic 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 level, exists as long as the object exists

Bank Account Example

- Let's look at another example that demonstrates the implementation details of classes and methods

- We'll represent a bank account by a class named `Account`

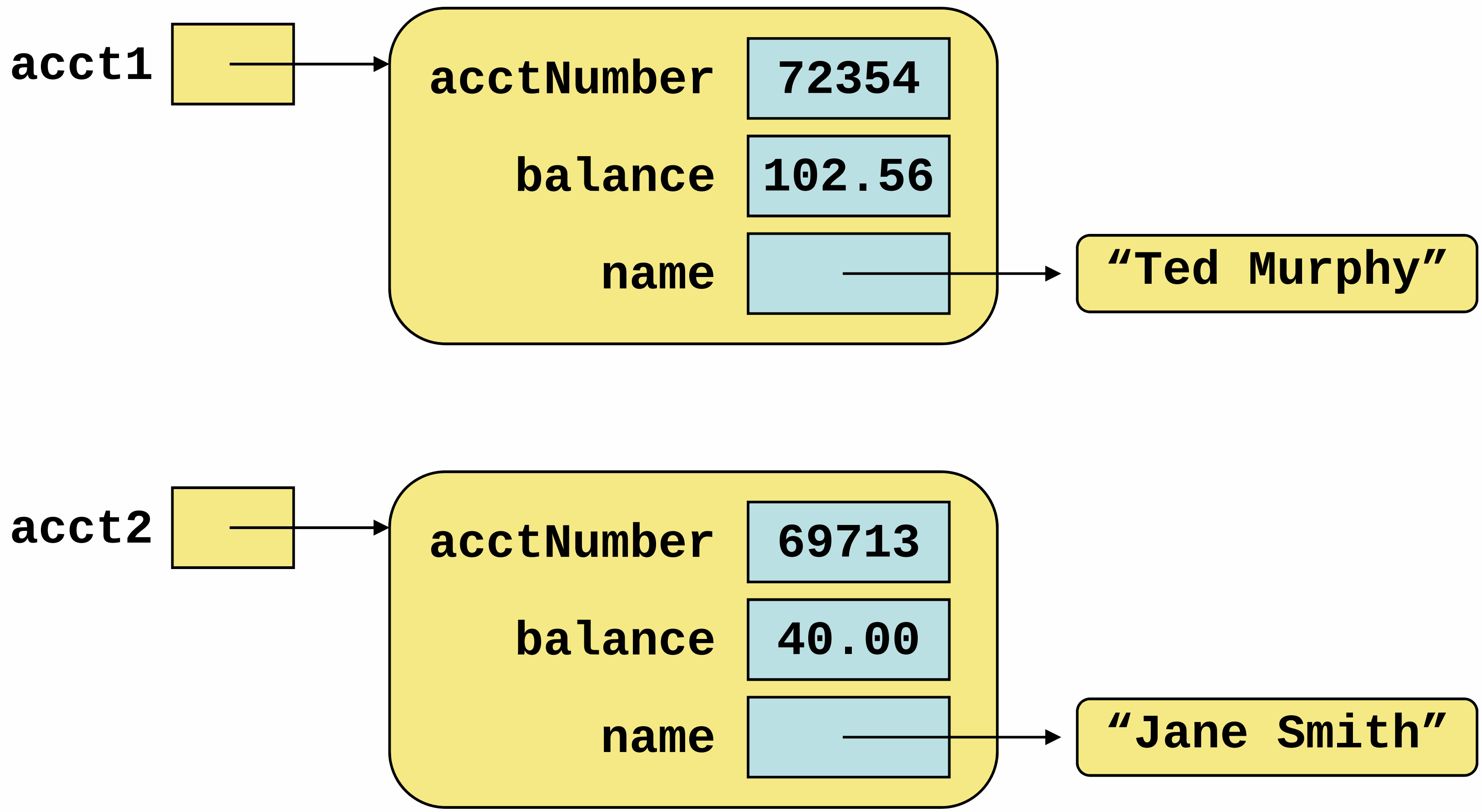
- It's state can include the account number, the current balance, and the name of the owner

- An account's behaviors (or services) include deposits and withdrawals, and adding interest

Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program
- Driver programs are often used to test other parts of the software
- The `Transactions` class contains a `main` method that drives the use of the `Account` class, exercising its services
- See Transactions.java (page 172)
- See Account.java (page 173)

Bank Account Example



Bank Account Example

- There are some improvements that can be made to the `Account` class
- Formal getters and setters could have been defined for all data
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive

Constructors Revisited

- **Note that a constructor has no return type specified in the method header, not even `void`**
- **A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class**
- **The programmer does not have to define a constructor for a class**
- **Each class has a *default constructor* that accepts no parameters**

Summary

- **Lecture 4 focused on:**
  - **class definitions**
  - **instance data**
  - **encapsulation and Java modifiers**
  - **method declaration and parameter passing**
  - **constructors**