AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

Writing Classes (5-7.5%)

- We've been using predefined classes. Now we will learn to write our own classes to define objects
- Chapter 5 focuses on:
  - class definitions
  - encapsulation and Java modifiers
  - method declaration, invocation, and parameter passing
  - method overloading
  - method decomposition

Preparing for the AP Exam

- On the free-response section of the exam, students are required to design program code that demonstrates the functionality available for an object of a new class, based on the prompt’ s specification.
- This process includes：
  - identifying the attributes (data) that define an object of a class
  - identifying the behaviors (methods) that define what an object of a class can do.
- Students should have many opportunities in this course to design their own classes based on program specifications and on observations of real-world objects.

Anatomy of OOP

- https://www.youtube.com/watch?v=CqIM7JjnAi4

Objects

- An object has:
  - *state (attribute)* - descriptive characteristics
  - *behaviors* - what it can do (or what can be done to it)
- For example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
- The state of the coin is its current face (heads or tails)
- The behavior of the coin is that it can be flipped
- Note that the behavior of the coin might change its state

Classes

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
- Each `String` object contains specific characters (its state)
- Each `String` object can perform services (behaviors) such as `toUpperCase`,`subSting`

Classes

- The `String` class was provided for us by the Java standard class library
- But we can also write our own classes that define specific objects that we need
- For example, suppose we want to write a program that simulates the flipping of a coin
- We can write a `Coin` class to represent a coin object

Encapsulation

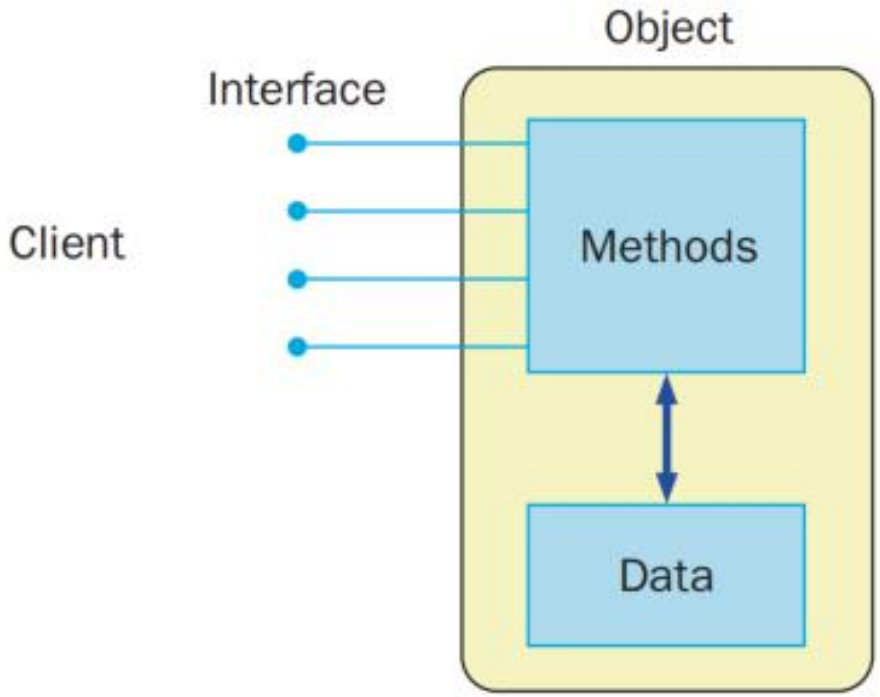
- We can take one of two views of an object:
  - internal - the variables the object holds and the methods that make the object useful
  - external - the services that an object provides and how the object interacts
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object
- Recall from Chapter 2 that an object is an *abstraction*, hiding details from the rest of the system

Encapsulation

- An object should be *self-governing*
- Any changes to the object's state (its variables) should be made only by that object's methods
- We should make it difficult, if not impossible, to access an object’s variables other than via its methods
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which invokes only the interface methods



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 value
- We've used the modifier `final` to define a constant
- We will study two visibility modifiers: `public` and `private`

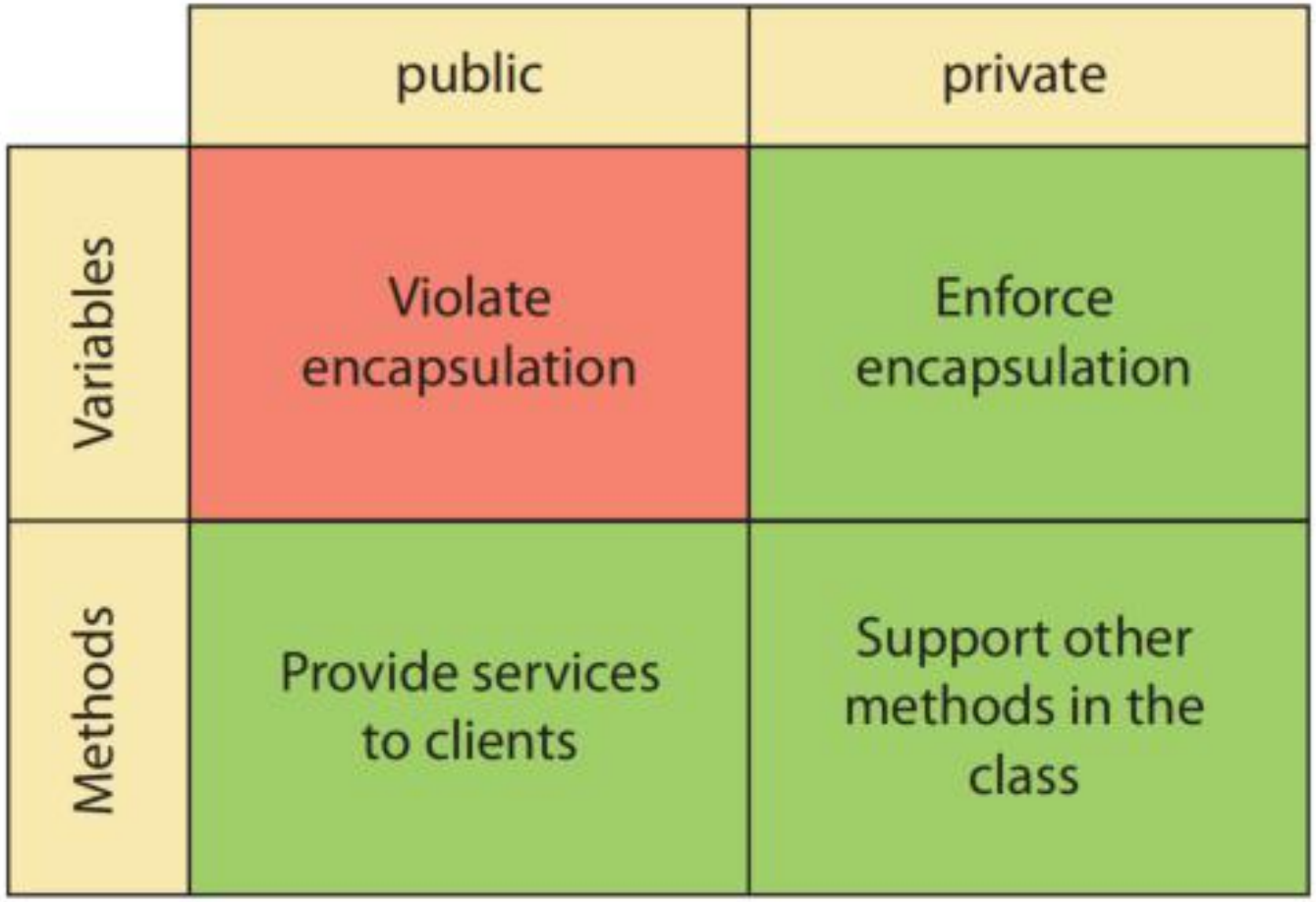
Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Public variables violate encapsulation
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package

Visibility Modifiers

- Methods that provide the object's services are usually 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——it is declared as private

Visibility Modifiers



The effects of public and private visibility

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

ESSENTIAL KNOWLEDGE

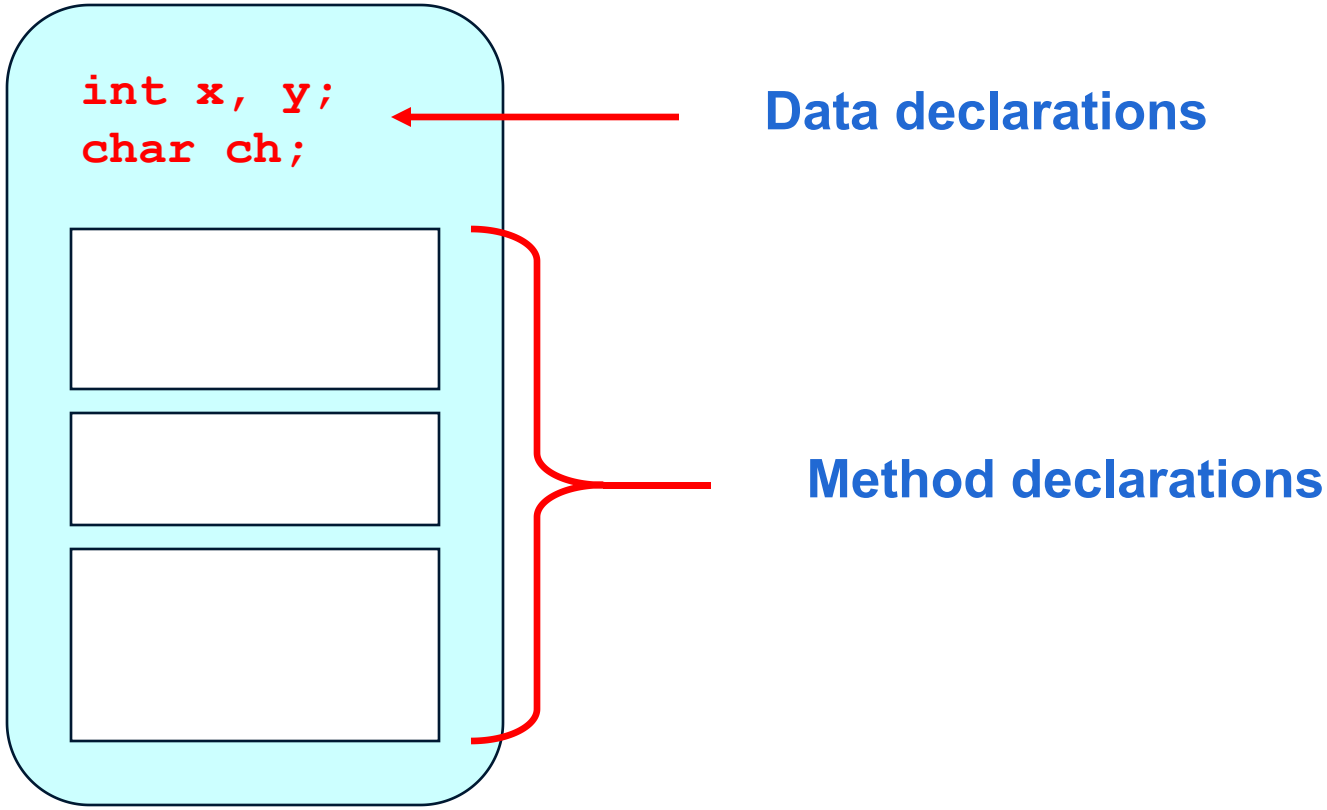
- The keywords public and private affect the access of classes, data, constructors, and methods.
- The keyword private restricts access to the declaring class, while the keyword public allows access from classes outside the declaring class.
- Classes are designated public.
- Access to attributes should be kept internal to the class. Therefore, instance variables are designated as private.
- Constructors are designated public.
- Access to behaviors can be internal or external to the class. Therefore, methods can be
- designated as either public or private.

ESSENTIAL KNOWLEDGE

- Data encapsulation is a technique in which the implementation details of a class are kept hidden from the user.
- When designing a class, programmers make decisions about what data to make accessible and modifiable from an external class. Data can be either accessible or modifiable, or it can be both or neither.
- Instance variables are encapsulated by using the private access modifier.
- The provided accessor and mutator methods in a class allow client code to use and
- modify data.

Classes

- A class contains data declarations and method declarations



Cteating a Class

- To write your own class, you typically start a class declaration with public then class then the name of the class. The body of the class is defined inside a { and a }.
- For example, the class House below. Then, you can create objects of that new House type by using Classname objectname = new Classname();

```
public class House
{
    // define class here - a blueprint

}

House myHouse = new House();
House neighborsHouse = new House();
```

Cteating a Class

```
public class Person
{
   /* instance variables for Person attributes */
   private String name;
   private String email;
   private String phoneNumber;

    /** a constructor to initialize the attributes
     for a Person object with the given parameters*/
   public Person(String initName, String initEmail,
                 String initphoneNumber)
   { /* Implementation not shown*/ }

   /* method to print Person attributes */
   public void print()
   { /* Implementation not shown*/ }
}
```

Person Instance Variables

Person Constructors

Person Methods

https://csawesome.runestone.academy/runestone/books/published/csawesome/Unit5-Writing-Classes/topic-5-1-parts-of-class.html

Instance Variables

- Instance variables are declared right after the class declaration.
- They usually start with private then the type of the variable and then a name for the variable.
- Private means only the code in this class has access to it.

Person Class

Private Encapsulated Data:
-name
-email
-phoneNumber

Public Methods:
+Person(String n, String e, String p)
+print()

p1 object
---
name = "Sana"
email = "sana@gmail.com"
phoneNumber="123-456-7890"

p2 object
---
name = "Jean"
email = "jean@gmail.com"
phoneNumber="404- 899-9955"

Methods

- Methods define what the object can do.
- They typically start with public then a type, then the name of the method followed by parentheses for optional parameters.
- Methods defined for an object can access and use its instance variables!

```
public void print()
{
  System.out.println("Name: " + name);
  System.out.println("Email: " + email);
  System.out.println("Phone Number: " + phoneNumber);
}
```

Methods

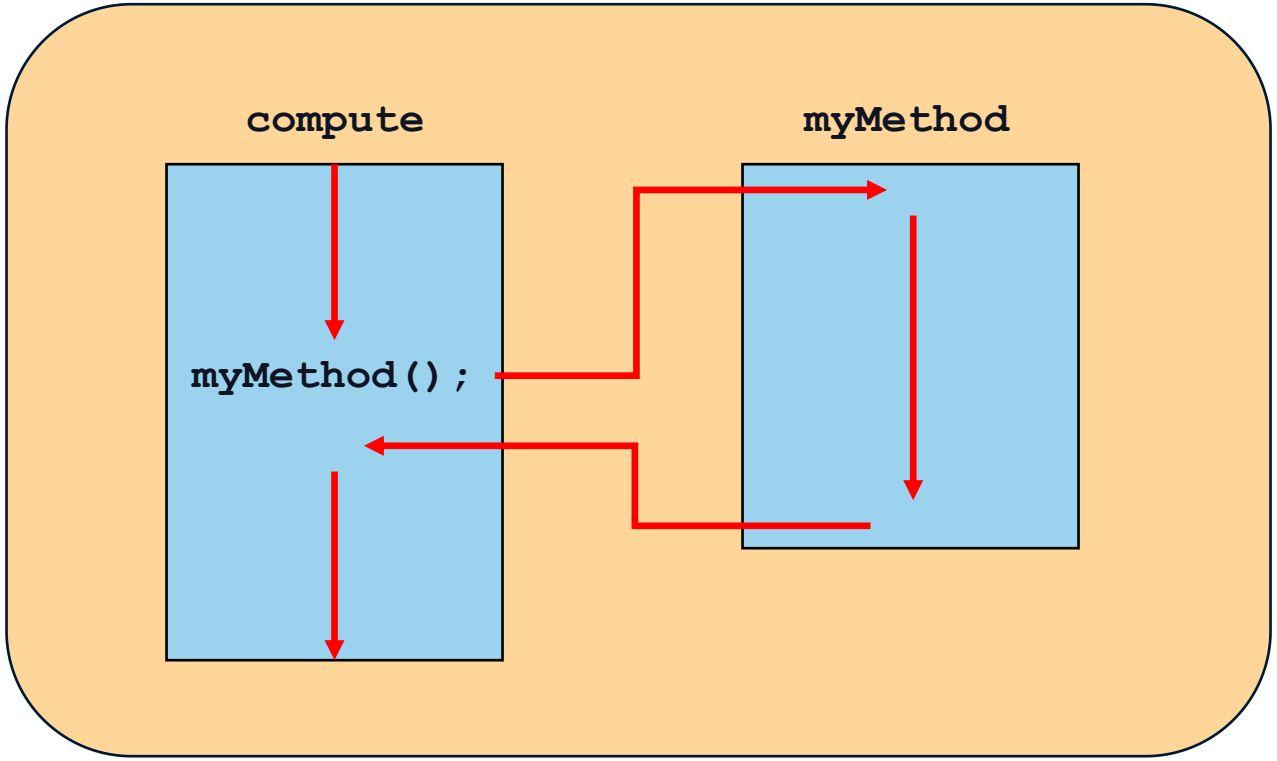
- To call a method to do its job, we create an object of the class and then use the dot (.) operator to call its public methods,
- for example p1.print() means call p1′ s print method.

```
public void print()
{
  System.out.println("Name: " + name);
  System.out.println("Email: " + email);
  System.out.println("Phone Number: " + phoneNumber);
}
```

```
// call the constructor to create a new person
Person p1 = new Person("Sana", "sana@gmail.com", "123-456-7890");
// call p1's print method
p1.print();
```

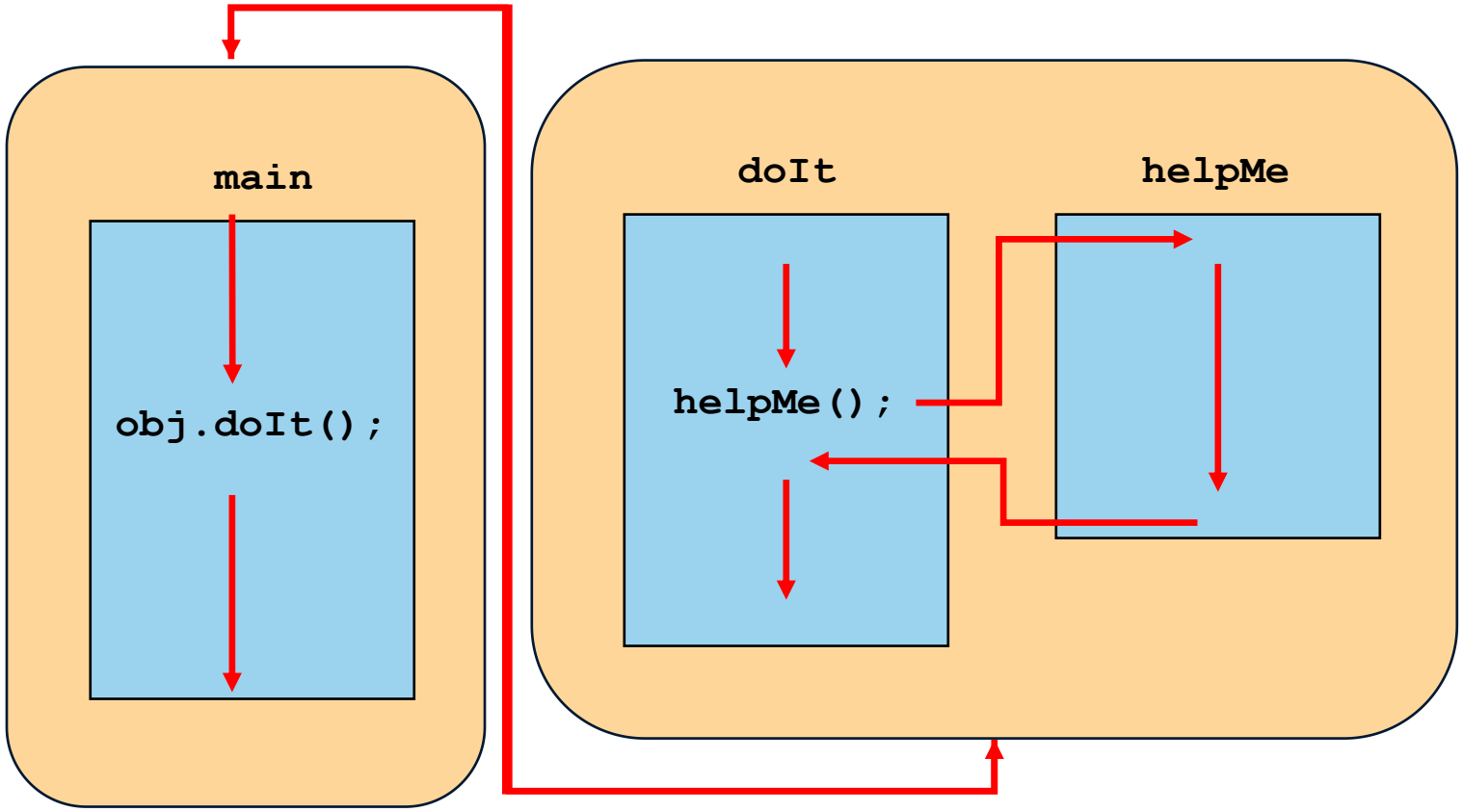
Method Control Flow

- The called method can be within the same class, in which case only the method name is needed



Method Control Flow

- The called method can be part of another class or object



Constructors Revisited

- Recall that a constructor is a special method that is used to initialize a newly created object

- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it typically sets the initial values of instance variables

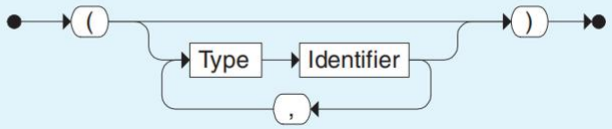
- The programmer does not have to define a constructor for a class

Constructors

constructor:

```
public classname (parameters);
```

Parameters



construct an object named objname:

```
classname objname = new classname(parameters);
```

```
1  public class Fraction
2  {
3     //  instance variables
4     private int numerator;
5     private int denominator;
6     // constructor: set instance variables to default values
7     public Fraction()    {
8        numerator = 1;
9        denominator = 1;    }
10    // constructor: set instance variables to init parameters
11    public Fraction(int initNumerator, int initDenominator)    {
12       numerator = initNumerator;
13       denominator = initDenominator;    }
14
15    // Print fraction
16    public void print()
17    {     System.out.println(numerator + "/" + denominator);    }
18
19    // main method for testing
20    public static void main(String[] args)    {
21       Fraction f1 = new Fraction();
22       Fraction f2 = new Fraction(1,2);
23       // What will these print out?
24       f1.print();
25       f2.print();
26    }
27 }
```

Overloading Methods

- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object
- See SnakeEyes.java
- See Die.java

Overloading Methods

Version 1

```
double tryMe (int x)
{
      return x+0.375;
}
```

Version 1

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

Invocation

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

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloaded Methods

- 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:");
System.out.println (total);
```

ESSENTIAL KNOWLEDGE

- An object’ s state refers to its attributes and their values at a given time and is defined by instance variables belonging to the object. This creates a “has-a” relationship between the object and its instance variables.
- Constructors are used to set the initial state of an object, which should include initial values for all instance variables.
- Constructor parameters are local variables to the constructor and provide data to initialize instance variables.
- When a mutable object is a constructor parameter, the instance variable should be initialized with a copy of the referenced object. In this way, the instance variable is not an alias of the original object, and methods are prevented from modifying the state of the original object.
- When no constructor is written, Java provides a no-argument constructor, and the instance variables are set to default values.

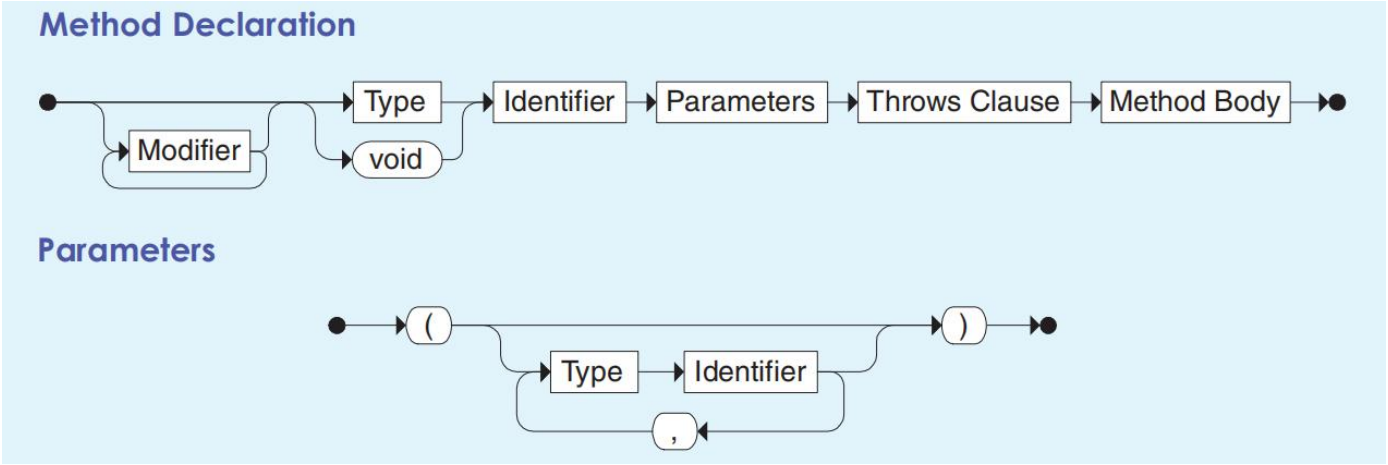
Game Time

```
public class CSA1{
        private String initname;  //instance variables
        public CSA1  (String name)  { //constructors
                initname = name;}
        public void handclap(int count){   // the object call this method will handclap count times
        /*implementation is not shown*/}
        public void repeat(String word,int count){   // the object call this method will repeat the word for count times
        /*implementation is not shown*/}
        public static void main(String[] args){
                CSA1 p1 = new CSA1("chen junhao");
                CSA1 p2 = new CSA1("liu yuxin");
                p1.handclap(3);
                p2.repeat(“hello world”,2);
                CSA1.handclap(3); // is this statement executable?why?
}
```

Game Time

```
public Class CSA2{
        private String initname;  //instance variables
        public CSA2  (String name)  { //constructors
                initname = name;}
        public void handclap(int count){   // the object call this method will handclap count times
        /*implementation is not shown*/}
        public void repeat(String word,int count){   // the object call this method will repeat the word for count times
        /*implementation is not shown*/}
        public static void main(String[] args){
                CSA2 p1 = new CSA2("sun siyang");
                CSA2 p2 = new CSA2("wang yizhou");
                p1.handclap(3);
                p2.repeat(“hello world”,2);
                CSA2.handclap(3); // is this statement executable?why?
}
```

Method Declarations



- A method is defined by optional modifiers（public or private）, followed by a return Type （int or double or void,etc）, followed by an Identifier that determines the method name, followed by a list of Parameters, followed by the Method Body.
- The return Type indicates the type of value that will be returned by the method, which may be void.
- The Method Body is a block of statements that executes when the method is invoked.
- The Throws Clause is optional and indicates the exceptions that may be thrown by this method.

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (or 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 Header

- A method declaration begins with a *method header*

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

The parameter list specifies the type and name of each parameter

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

Method Body

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

```
{
    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 variables**

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

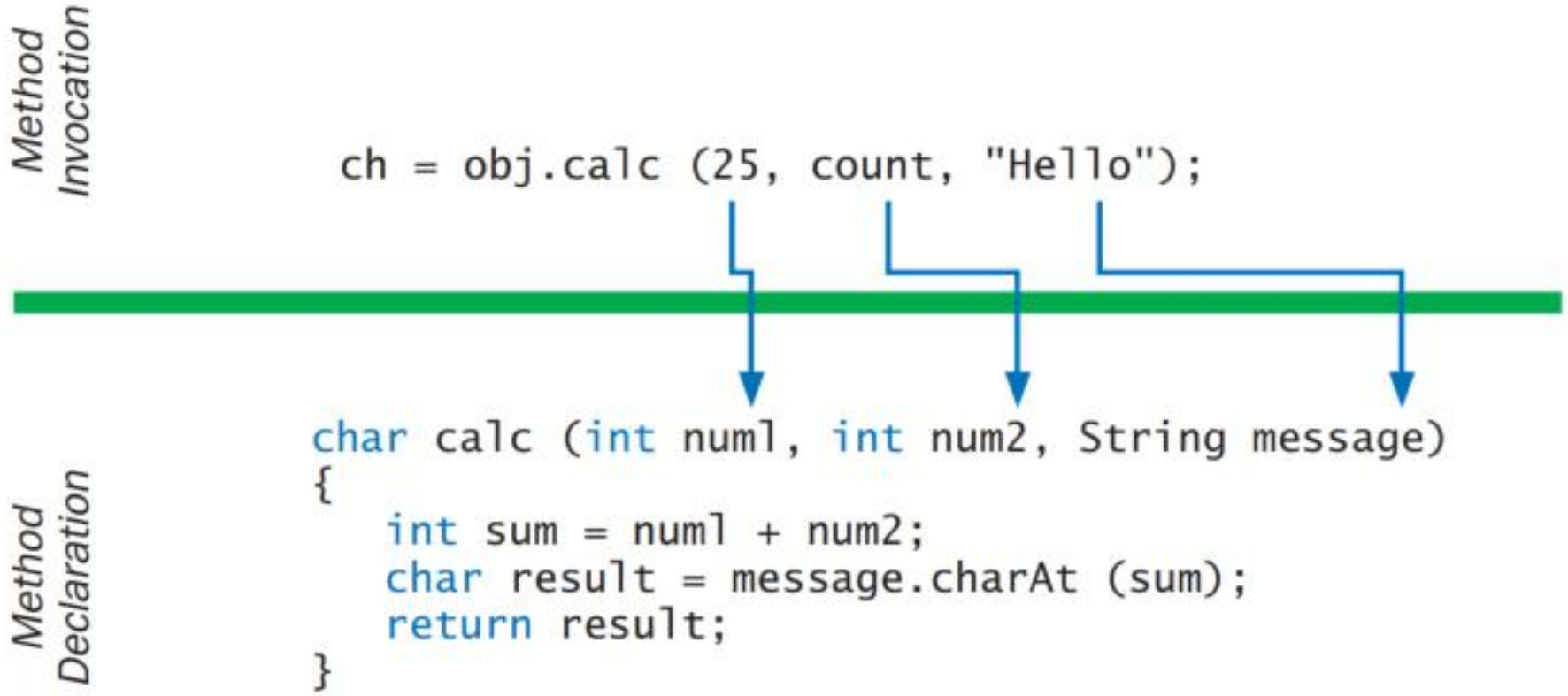
- A *return statement* specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type

Parameters

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



passing parameters from the method invocation to the declaration

Accessors and Mutators

- Since instance variable usually has private visibility, it can only be accessed through methods
- An *accessor method* provides read-only access to a particular value
- A *mutator method* changes a particular value
- For a data value `X`, accessor and mutator methods are usually named `getX` and `setX`

Accessors and Mutators

```
//Instance variable declaration
private typeOfVar varName;
```

```
/** setVar sets varName to a newValue
 *  @param newValue
 */
public void setVarName(typeOfVar newValue)
{
    varName = newValue;
}
```

```
/** getVar() returns varName's value
 *  @return varName
 */
public typeOfVar getVarName()
{
    return varName;
}
```

ESSENTIAL KNOWLEDGE

- An accessor method allows other objects to obtain the value of instance variables or static variables.
- A non-void method returns a single value. Its header includes the return type in place of the keyword void.
- In non-void methods, a return expression compatible with the return type is evaluated, and a copy of that value is returned. This is referred to as "return by value."
- When the return expression is a reference to an object, a copy of that reference is returned, not a copy of the object.
- The return keyword is used to return the flow of control to the point immediately following where the method or constructor was called.
- A void method does not return a value. Its header contains the keyword void before the method name.
- A mutator (modifier) method is often a void method that changes the values of instance variables or static variables.

Data Scope

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

Local variables

- 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
- Any method in the class can refer to instance variables

Preconditions and Postconditions

- A *precondition* is a condition that should be true when a method is called
- A *postcondition* is a condition that should be true when a method finishes executing
- These conditions are expressed in comments above the method header
- Both preconditions and postconditions are a kind of *assertion*, a logical statement that can be true or false which represents a programmer´s assumptions about a program

ESSENTIAL KNOWLEDGE

- Comments are ignored by the compiler and are not executed when the program is run.
- Three types of comments in Java include /* */, which generates a block of comments, //, which generates a comment on one line, and /** */, which are Javadoc comments and are used to create API documentation.
- A precondition is a condition that must be true just prior to the execution of a section of program code in order for the method to behave as expected. There is no expectation that the method will check to ensure preconditions are satisfied.
- A postcondition is a condition that must always be true after the execution of a section of program code. Postconditions describe the outcome of the execution in terms of what is being returned or the state of an object.
- Programmers write method code to satisfy the postconditions when preconditions are met.

the toString method

- The toString method is an overridden method that is included in classes to provide a description of a specific object. It generally includes what values are stored in the instance data of the object.

- If System.out.print or System.out.println is passed an object, that object's toString method is called, and the returned string is printed.

This keyword

- Within a non-static method or a constructor, the keyword this is a reference to the current object—the object whose method or constructor is being called.
- The keyword this can be used to pass the current object as an actual parameter in a method call.

```
public class Person
{
   // instance variables
   private String name;
   private String email;
   private String phoneNumber;

   // constructor: construct a Person copying in the data into the instance variab
   public Person(String name, String email, String phone)
   {
      this.name = name;
      this.email = email;
      this.phoneNumber = phone;
   }
```

The static modifier

- “static” is a reserved word in Java and indicates that a variable or method is attached to a class instead of an object.

- EG: a static variable may count how many objects of a specific class get created or may hold a value which is true for all objects.

- A static method cannot access non-static instance variables (because all of those instance variables are attached to objects). This makes a static method similar to a “function” (rather than a method) in other languages.

ESSENTIAL KNOWLEDGE

- Static methods are associated with the class, not objects of the class.
- Static methods include the keyword staticin the header before the method name.
- Static methods cannot access or change the values of instance variables.
- Static methods can access or change the values of static variables.
- Static methods do not have a this reference and are unable to use the class’ s instance variables or call non-static methods.
- Static variables belong to the class, with all objects of a class sharing a single static variable.
- Static variables can be designated as either public or private and are designated with the static keyword before the variable type.
- Static variables are used with the class name and the dot operator, since they are associated with a class, not objects of a class.

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate

Pig Latin

- The process of translating an English sentence into Pig Latin can be decomposed into the process of translating each word

- The process of translating a word can be decomposed into the process of translating words that

    - begin with vowels
    - begin with consonant blends (sh, cr, tw, etc.)
    - begins with single consonants

- See PigLatin.java

- See PigLatinTranslator.java

Summary

- Chapter 4 has focused on:
  - class definitions
  - encapsulation and Java modifiers
  - method declaration, invocation, and parameter passing
  - method overloading
  - method decomposition

Another example

The Coin Class

- In our `Coin` class we could define the following data:
  - `face`, an integer that represents the current face
  - `HEADS` and `TAILS`, integer constants that represent the two possible states
- We might also define the following methods:
  - a `Coin` constructor, to initialize the object
  - a `flip` method, to flip the coin
  - a `getFace` method, to determine the face of the coin
  - a `toString` method, to return a string description for printing

The Coin Class

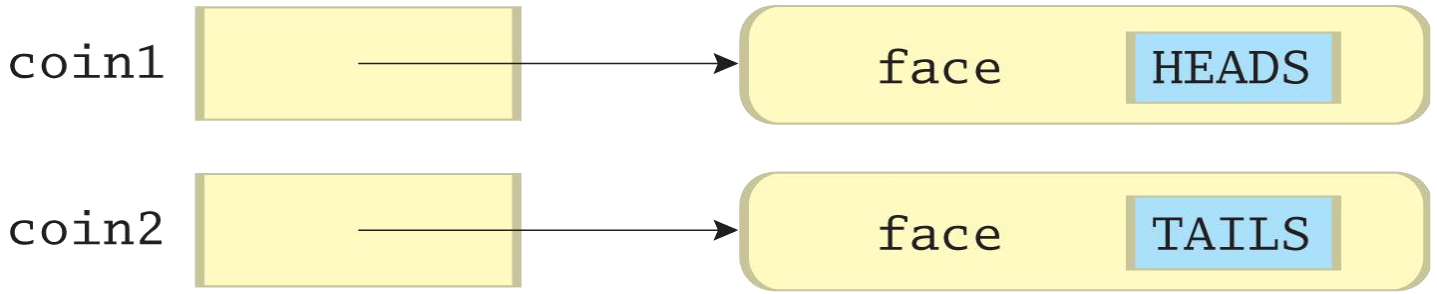
- See CountFlips.java
- See Coin.java
- Note that the `CountFlips` program did not use the `toString` method
- A program will not necessarily use every service provided by an object
- Once the `Coin` class has been defined, we can use it again in other programs as needed

Instance variables

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

Instance variables

- See FlipRace.java



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 `CountFlips` class contains a `main` method that drives the use of the `Coin` class, exercising its services
- See CountFlips.java
- See Coin.java