



清森学校
BEIJING QINGSEN
SCHOOL

AP-CSA

Writing a class

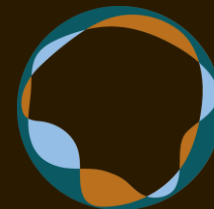
YING HUANG

NOV.2022

REVIEW

```
public class House1 {  
    private String color;  
    private String owner;  
    private int ID;  
    public House1(String color, String owner, int ID)  
    {  
        this.color = color;  
        this.owner = owner;  
        this.ID = ID;  
    }  
    public void changeColor(String c) { color=c; }  
  
    public int sum(int num1, int num2){...}  
}
```





清森学校
BEIJING QINGSEN
SCHOOL

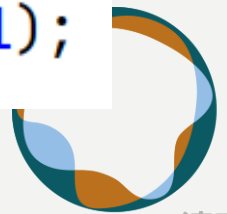
ANATOMY OF CLASS

ANATOMY OF CLASS

In Unit 2, we learned to use **classes** and **objects** that are built-in in Java(String, Math) or written by other programmers. In this unit, you will learn to write your own classes and objects!

Remember that a **class** in programming defines a new **abstract data type**. When you create **objects**, you create new variables or **instances** of that class data type.

```
String a = new String("hello");  
Scanner input = new Scanner(System.in);  
House1 own = new House1("Blue", "Ying", 1001);
```

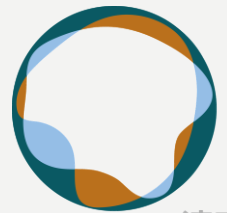


CLASS DECLARATION

- 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 the curly braces {}.

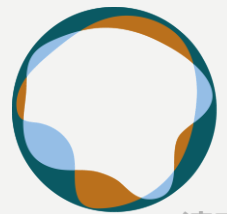
- Syntax:

```
public class ClassName {  
    // define class here - a blueprint  
  
}
```



INSTANCE ATTRIBUTES/METHODS

- Remember that objects have attributes and behaviors. These correspond to **instance variables/ attributes** and **methods** in the class definition.
- **Attributes** hold the data for objects whereas the **methods** code the behaviors or the actions that can manipulate the data of the object.
- A class also has **constructors** which initialize the instance variables when the object is created.

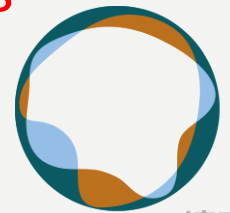


EXAMPLE

```
public class Point {  
    int x;  
    int y;  
    public Point(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

← declare instance
variables/ attribute

← constructor: initialize
variables



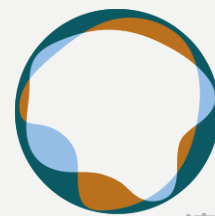
CONSTRUCTOR 构造函数

The **constructor** of a class is a method that allows us to initialize the attributes(variables) of an object when it is first created.

The name of constructor is same as class's name!

Syntax:

```
public className(...)  
{  
    ....  
}
```



OVERLOADED CONSTRUCTORS

Constructors are said to be **overloaded** when there are multiple constructors with the same name but a different signature.

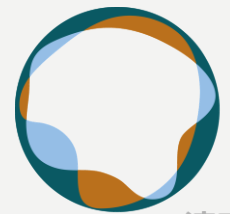
```
public house()  
{  
    ...  
}
```

no parameter

```
public house(String color)  
{  
    ...  
}
```

one parameter

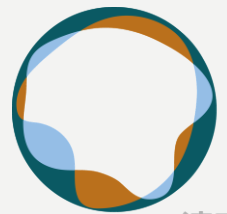
A parameter is a variable used to define a particular value during a function definition.



OVERLOADED CONSTRUCTORS

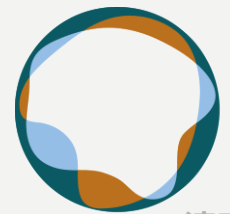
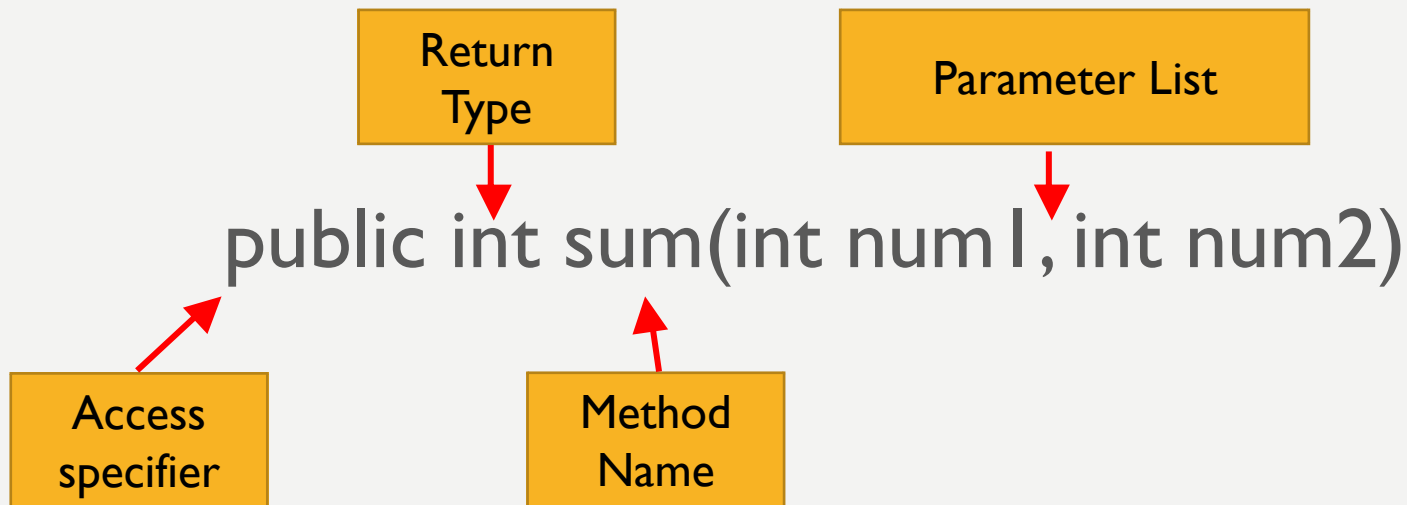
We can call different constructors to initialize our objects.

```
house h1 = new house();  
// default constructor initializes  
house h2 = new house("Red");  
// h2.color = red  
house h3 = new house("Red", "Ying", 111);  
// h3.color = Red, h3.owner = Ying h3.ID = 111
```



METHOD DECLARATIONS

Method define the behaviors or functions for objects.

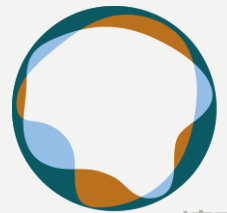


ACCESS SPECIFIER

Access specifier determine whether classes, data, constructors and methods can be accessed outside of the declaring class.

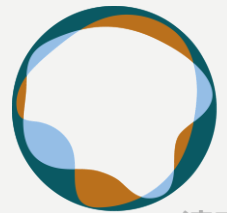
Type of Access:

- **public**: be accessed from **outside** the class.
(default)
- **private**: can only be accessed from **inside** the enclosing class.



public and private Access

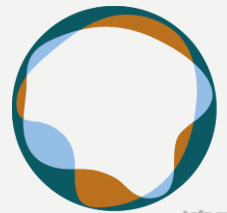
- class and constructors are designated as **public** so that they can be accessed outside of the class file.
- attribute/ instance variable are designated as **private** so that they can be more safe



ENCAPSULATION

Object-oriented Programming stresses **data encapsulation** where the data (instance variables) and the code acting on the data (methods) are wrapped together into a single unit and the implementation details are hidden.

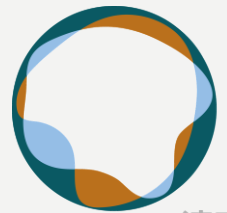
The data is protected from harm by being kept private. Anything outside the class can only interact with the public methods and cannot interact directly with the private instance variables.



EXAMPLE 1

In this exercise, you are going to create the instance variables for an Employee class. The class needs to store the employee's first name, last name, monthly salary, and the number of hours in their shift. You will need to give each instance variable an appropriate name, type, and privacy setting.

After defining the instance variables, create the structure for the constructor. Make sure you set the privacy settings on the constructor correctly.



ACCESSOR AND MUTATOR

- Accessors/ getters: Methods generally have a return value, which data type should be match the method signature/ attribute return type

– For example:

```
public int getAge(){ return age;}
```

- Mutator/ setter: Methods are often void methods that change the value of instance and static variables.

– For example:

```
public void setAge( int newage)  
{ age = Newage; }
```



EXAMPLE 2 – CODEHS

5.5.5 RECTANGLE CLASS

Write your own accessor and mutator method for the Rectangle class instance variables. You should create the following methods:

- getHeight
- setHeight
- getWidth
- setWidth
- getArea
- getPerimeter
- toString- The output of a rectangle with width 10 and height 4 method should be:

Rectangle width: 10, Rectangle height: 4

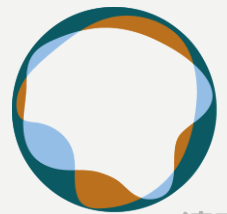


COMMENTS

Adding comments to your code helps to make it more readable and maintainable.

In the commercial world, software development is usually a team effort where many programmers will use your code and maintain it for years.

Commenting is essential in this kind of environment and a good habit to develop. Comments will also help you to remember what you were doing when you look back to your code a month or a year from now.



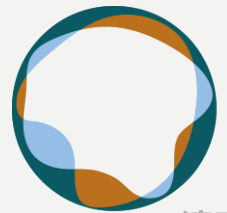
COMMENTS

There are 3 types of comments in Java:

1. `//` Single line comment
2. `/*` Multiline comment `*/`
3. `/**` Documentation comment `*/`

We have seen the first two types of comments. The third is also a special version of the multi-line comment, `/** */`, called the **documentation comment**.

Java has a tool called **javadoc** that comes with the Java JDK that will pull out all of these comments to make documentation of a class as a web page.



PRECONDITIONS/ POSTCONDITIONS

A **precondition** is a condition that must be true for your method code to work, for example the assumption that the parameters have values and are not null. There is no expectation that the method will check to ensure preconditions are satisfied.

The methods could check for these preconditions, but they do not have to. The precondition is what the method expects in order to do its job properly.

A **postcondition** is a condition that is true after running the method. It is what the method promises to do. Postconditions describe the outcome of running the method, for example what is being returned or the changes to the instance variables.



PRECONDITIONS/ POSTCONDITIONS

*/***

** Constructor that takes the x and y position of Sprite object*

** Preconditions: parameters x and y are coordinates from 0 to the width and height of the window*

** Postconditions: the Sprite object is placed in (x,y) coordinates*

** @param x the x position to place the Sprite*

** @param y the y position to place the Sprite */*

```
public Sprite(int x, int y) {  
    center_x = x;  
    center_y = y;  
}
```



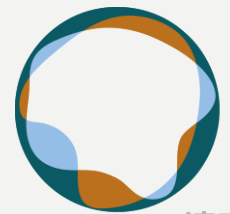
CLASS/HOME WORK

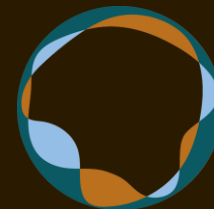
Code HS:

5.2.5 – 5.2.8

5.4.5 – 5.4.8

5.5.5 -5.5.7



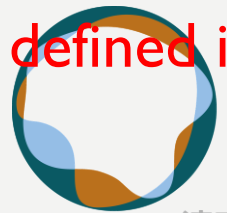


清森学校
BEIJING QINGSEN
SCHOOL

STATIC VS NON-STATIC

STATIC VS NON-STATIC

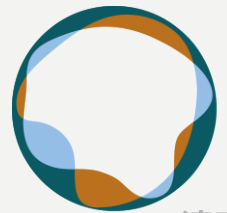
- Variables and methods can be classified as **static** or **nonstatic(instance)**.
- **Non-static or instance:** Part of an object, **Non-static methods are called using the dot operator along with the object variable name.**
- **static:** Part of a class. Not copied into each object; shared by all objects of that class. **Static methods are called using the dot operator along with the class name unless they are defined in the enclosing class.**



STATIC METHODS

- **Static Methods** are the methods in Java that can be called without creating an object of a class. They are referenced by the class name itself.
 - Stored in a class, not in an object.
 - Shared by all objects of the class, not replicated.
- Syntax

```
public static type name(parameters) {  
    statements;  
}
```



ERROR

```
public class House1 {  
  
    public String color;  
    private String owner;  
    private int ID;  
    public House1(String color, String owner, int ID)  
    {  
        this.color = color;  
        this.owner = owner;  
        this.ID = ID;  
    }  
    public static String getColor() { return color(); }  
    public static String getOwner() { return owner(); }  
    //...  
}
```

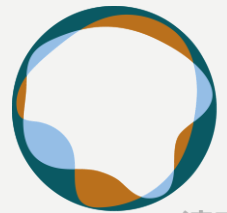
Because static methods belong to the class, and not an object, they cannot access instance variables or non-static methods of the class.



STATIC VARIABLES

static variable: Stored in the class instead of each object.

A "shared" global field that all objects can access and modify.
Like a class constant, except that its value can be changed.



FINAL STATIC FIELDS

- **Final static variable:**

- A class constant whose value cannot be changed. Usually public.
- ALL CAPS by convention.

- **Syntax**

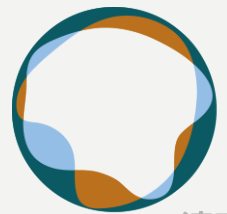
```
public static final type name;
```

or,

```
public static final type name = value;
```

- **For example:**

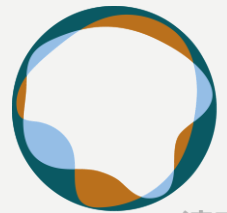
```
public static final int NUMOFMONTHS = 12;
```



Swapping values 1.0

What is wrong with this code? What is its output?

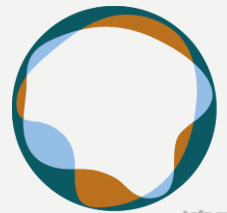
```
int a = 7;  
int b = 35;  
// swap a with b  
a = b;  
b = a;  
System.out.println(a + " " + b);
```

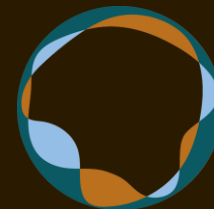


Swapping values 2.0

What is wrong with this code? What is its output?

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
    // swap a with b  
    swap(a, b);  
    System.out.println(a + " " + b);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```





清森学校
BEIJING QINGSEN
SCHOOL

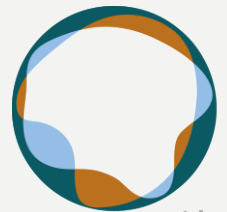
SEMANTICS

VALUE SEMANTICS

value semantics: Behavior where values are copied when assigned, passed as parameters, or returned.

- All primitive types in Java use value semantics.
- When one variable is assigned to another, its value is copied.
- Modifying the value of one variable does not affect others.

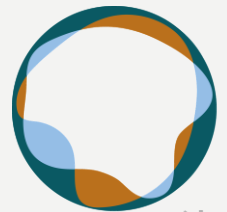
```
int x = 5;  
int y = x;    // x = 5, y = 5  
y = 17;      // x = 5, y = 17  
x = 8;       // x = 8, y = 17
```



REFERENCE SEMANTICS (OBJECTS)

- **reference semantics:** Behavior where variables actually store the address of an object in memory.
 - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*(aliases).
 - Modifying the value of one variable *will* affect others.

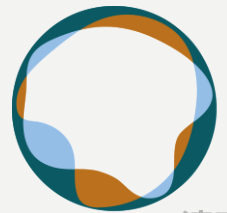
```
House1 h1 = new House1("blue", "Ying", 344);  
House1 h2 = h1;  
h2.color="white";  
System.out.println(h1.color);
```



PRIMITIVE AS PARAMETERS

- The primitive types `int`, `double`, `boolean` all use value semantics.
- When an **actual parameter** is a primitive value, the **formal parameter** is initialized with a copy of that value. Changes to the formal parameter have no effect on the corresponding actual parameter.
- For example:

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
    // swap a with b  
    swap(a, b);  
    System.out.println(a + " " + b);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```



OBJECT AS PARAMETERS

- When an **actual parameter** is a reference to an object, the **formal parameter** is initialized with that refers to the same object.
 - *efficiency*. Copying large objects slows down a program.
 - *sharing*. It's useful to share an object's data among methods.
 - **Except String**

- For example

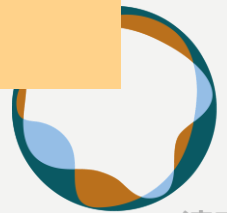
```
public static void main(String[] args) {  
    House1 h1 = new House1("bule", "Ying", 344);  
    restColor(h1);  
    System.out.println(h1.color);  
}  
  
public static void restColor(House1 house){  
    house.color="black";  
}
```

STRING AS PARAMETERS

String uses value semantics like primitive types. It's the only object class that uses value semantic.

Example:

```
public static void repeat(String str) {  
    str = str + str;  
}  
  
public static void main(String[] args) {  
  
    String str = "hi";  
    repeat(str);  
    System.out.println(str);  
}
```

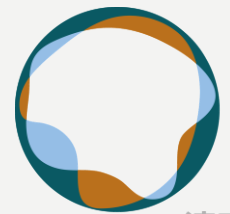


CLASS/HOME WORK

Code HS:

5.6.5 - 5.6.7

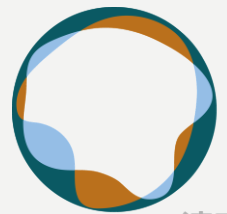
5.7.5 -5.7.7

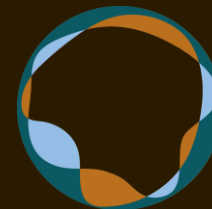


WHAT IS THE ERROR?

```
int x = 4;  
if(x>0){  
    int y =2;  
}  
System.out.println(y);
```

outside scope





清森学校
BEIJING QINGSEN
SCHOOL

SCOPE AND ACCESS

SCOPE

- In general, a variable exists from the point where it is declared, until the end of the block it is declared inside of.
- Variable can be only used in the block.
- A block to code enclosed in curly braces:

```
{  
    // code  
}
```

```
int x = 4;  
if(x>0){  
    int y =2;  
}  
System.out.println(y);
```



LOCAL VARIABLES

Local variables only exist in the context of the method or constructor they are created in.





- Local variables cannot be declared as public or private
- For example

```
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```



EXAMPLE

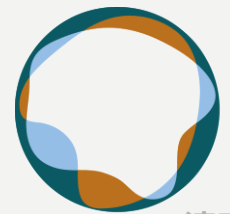
```
public class Example
{
    private int instanceVar = 2;
    public int add(int x)
    {
        int localVar = x;
        for(int i = 0; i < x; i++)
        {
            localVar += instanceVar * i;
        }
        return localVar;
    }
}
```

-  = instanceVar exists
-  = x exists
-  = localVar exists
-  = i exists

WHAT IS THE OUTPUT?

```
public class WriteClassExample {  
    private int number = 0;  
  
    public void printNumber(){  
        int number = 10;  
        System.out.println(number);  
    }  
}
```

The local variable number has **more specific** scope than the instance variable number!

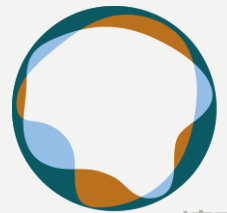


NAMING CONFLICTS

If two variables have the same name inside of the same scope:

- The variable with the more specific scope takes precedence.
- The variable with the more general scope no longer exists in that location.

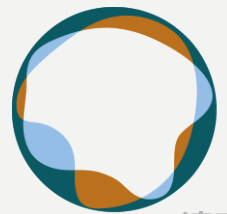
This is called **shadowing**



WHAT IS THE OUTPUT?

```
public class WriteClassExample {  
    private int number =0;  
  
    public void printNumber(int number){  
        System.out.println(number);  
    }  
}
```

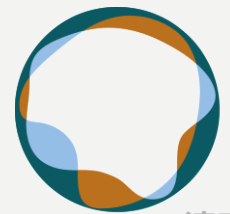
The formal parameter number will **take precedence over** then instance variable.



SCOPE IMPLICATIONS

```
for (int i = 1; i <= 100 * line; i++) {  
    int i = 2;  
    System.out.print("/");  
}  
i = 4;
```

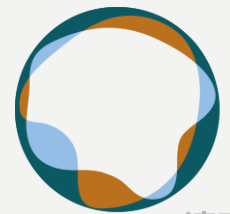
- The error is **overlapping scope**
- A variable can't be declared twice or used out of its scope.



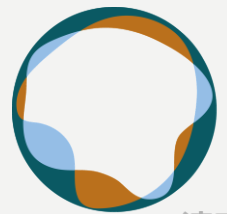
SCOPE IMPLICATIONS

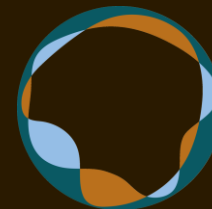
```
for (int i = 1; i <= 100; i++) {  
    System.out.print("A");  
}  
  
for (int i = 1; i <= 100; i++) {  
    System.out.print("BB");  
}  
  
int i = 5;
```

Variables without overlapping scope can have same name.



```
public class WriteClassExample {  
    private int number = 0;  
  
    public void printNumber(){  
        int number = 10;  
        System.out.println(number);  
    }  
}
```





清森学校
BEIJING QINGSEN
SCHOOL

“this”
KEYWORD

this Keyword

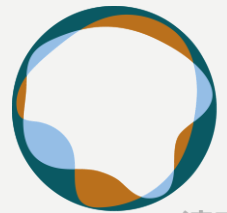
Within a non-static method or a constructor, the keyword **this** is a reference to the current object whose methods or constructors are being called.

Syntax:

Refer to a field: `this.field`

Call a method: `this.method(parameters);`

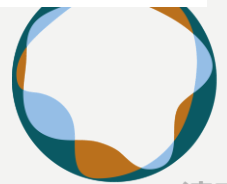
In constructor `this.variableName;`



this Keyword

Example:

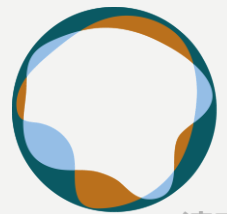
```
public class rectangle {  
    private int number;  
    private int width;  
    private int height;  
  
    public rectangle(int number, int width, int height){  
        this.number = number;  
        this.width = width;  
        this.height = height;  
    }  
}
```



this Keyword

- **this** can also be used to call an object's methods

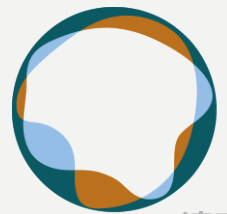
```
public int getArea(){  
    return this.width*this.height;  
}  
  
public boolean hasGreaterArea(rectangle other){  
    return this.getArea()>other.getArea();  
}
```



this Keyword

- **this** can also be used as an actual parameter to pass the current object to a method

```
public int getArea(rectangle rect){  
    return rect.width*rect.height;  
}  
  
public boolean hasGreaterArea(rectangle other){  
    return getArea(this)>getArea(other);  
}
```



CLASS/HOME WORK

Code HS:

5.8.7 - 5.8.9

5.9.5 – 5.9.7

