AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

5-writing classes &inheritance

- writing Classes
  - instance variables
  - constructors
  - getters & setters
  - data scope
  - static modifier
  - comments & precondition & postcondition
- Inheritance
  - superclass & subclass
  - override method & super keyword
  - Polymorphism
  - String toString() & boolean equals(Object other)

I'm getting a bit hungry…what could I have as a snack?

These are all instances of a snack!

Snack Graphics © Czajka, 2020

Computer Science

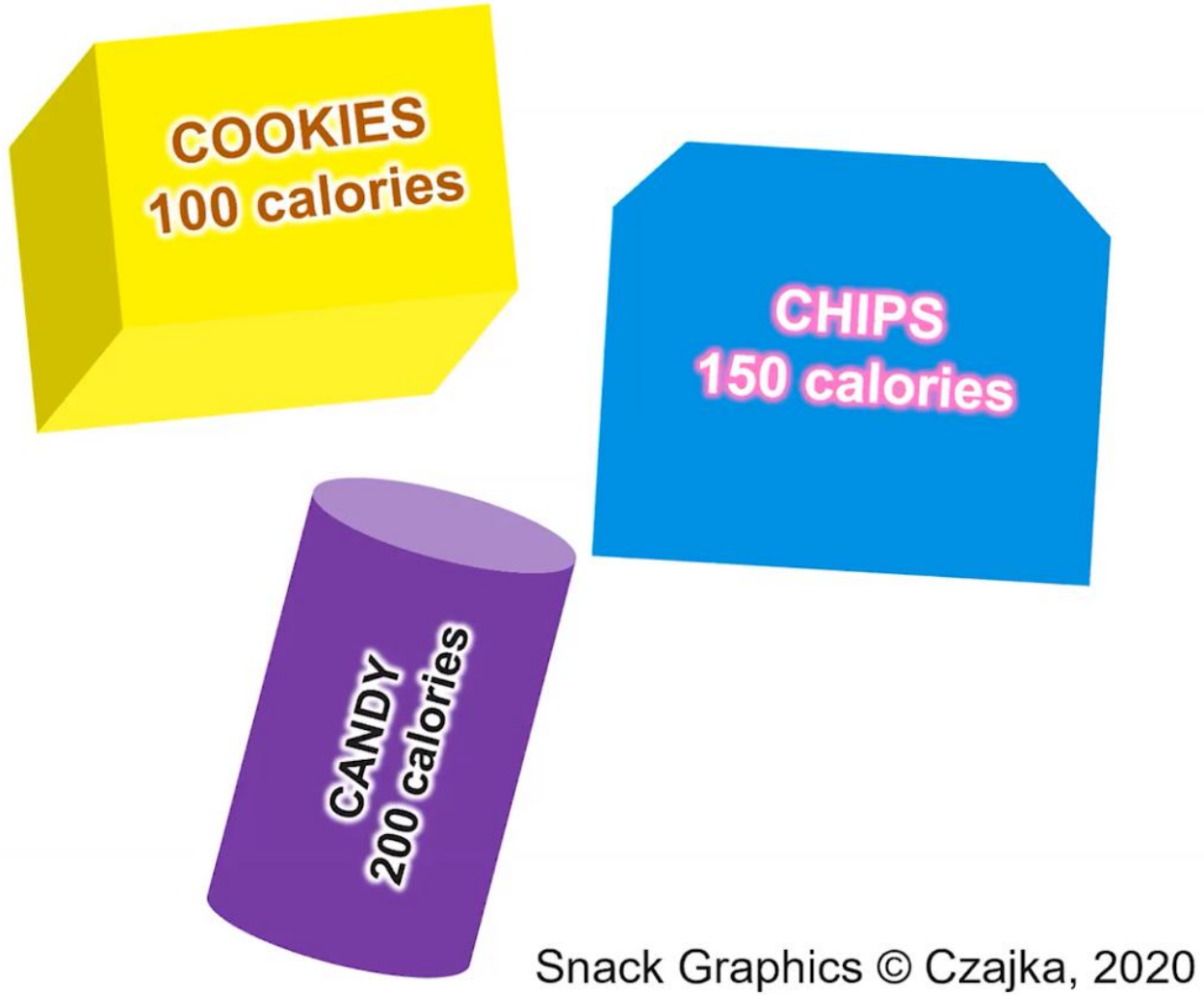
- Create models of things that exist in the real world
  - Blueprints → Class
  - Use the class → Instances of the class—Objects
  - Attributes of the objects → Instance variables
  - Behaviors of the objects → Methods

Class——Snack

Class—Snack

- Attributes
  - name
  - calories
- Behaviors
  - get name/calories
  - set name/calories

Let's take a look at what this class might look like…



Snack Graphics © Czajka, 2020

```
public class Snack{
    private String name;
    private int calories;

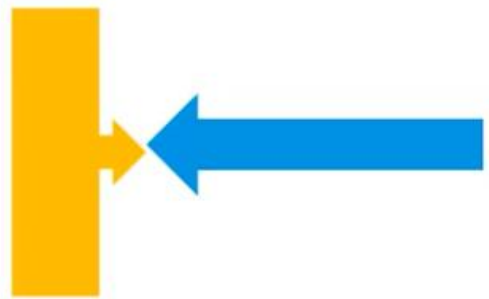
    public Snack(){
        name = "";
        calories = 0;
    }
    public Snack(String n, int c){
        name = n;
        calories = c;
    }
    public String getName(){
        return name;
    }
    public int getCalories(){
        return calories;
    }
    public void setName(String n) {
        name = n;
    }

    public void setCalories(int c) {
        calories = c;
    }
}
```

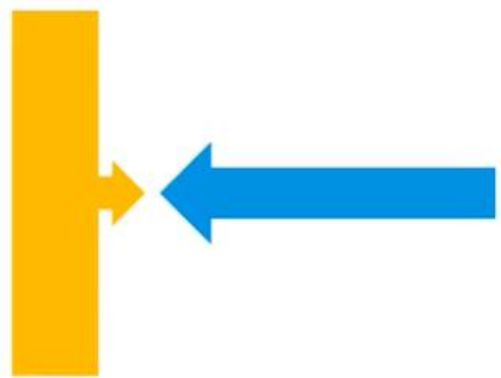
private instance variables

default constructor

overloaded constructor



accessor methods



mutator methods

```
public class Snack{
    private String name;
    private int calories;

    public Snack(){
        name = "";
        calories = 0;
    }
    public Snack(String n, int c){
        name = n;
        calories = c;
    }
    public String getName(){
        return name;
    }
    public int getCalories(){
        return calories;
    }
    public void setName(String n) {
        name = n;
    }

    public void setCalories(int c) {
        calories = c;
    }

    private boolean canEat() {
       return (calories < 150);
    }
}
```

`public` access modifiers

- no restriction on access
- other classes can access

For AP CS-A

- classes will be **`public`**
- constructors will be **`public`**

```
public class Snack{
     private String name;
     private int calories;

     public Snack(){
          name = "";
          calories = 0;
     }
     public Snack(String n, int c){
          name = n;
          calories = c;
     }
     public String getName(){
          return name;
     }
     public int getCalories(){
          return calories;
     }
     public void setName(String n) {
          name = n;
     }

     public void setCalories(int c) {
          calories = c;
     }

     private boolean canEat() {
        //implementation not shown
     }
}
```

`private` access modifiers

- restrictions on access
- only access in given cl

For AP CS-A

- instance variables will be **`private`**

Why make instance variables private?

- Restrict access (read-only)
- Option to provide validation checks
- For now, you need to make your instance variables private for AP CS-A

```
public class Snack{
     private String name;
     private int calories;

     public Snack(){
          name = "";
          calories = 0;
     }
     public Snack(String n, int c){
          name = n;
          calories = c;
     }
     public String getName(){
          return name;
     }
     public int getCalories(){
          return calories;
     }
     public void setName(String n) {
          name = n;
     }

     public void setCalories(int c) {
          calories = c;
     }

     private boolean canEat() {
        return (calories < 150);
      }
}
```

Methods can be **`public`** or **`private`**.

- Beware of accessibility.
- An object can call on **`public`** methods in any class.
- **`private`** methods can only be called in their own class.

SnackDriver Class with Main Method

Which of the following lines will cause an error?

```
public class SnackDriver {
	public static void main(String[] args) {
		Snack choiceOne = new Snack("cookies", 100);
		Snack choiceTwo = new Snack();
		System.out.println(choiceOne.getName());
		System.out.println(choiceOne.getCalories());
		choiceTwo.setName("chips");
		choiceTwo.calories = 150;
		System.out.println(choiceTwo.canEat());
	}
}
```

SnackDriver Class with Main Method

Which of the following lines will cause an error?

```
public class SnackDriver {
	public static void main(String[] args) {
		Snack choiceOne = new Snack("cookies", 100);
		Snack choiceTwo = new Snack();
		System.out.println(choiceOne.getName());
		System.out.println(choiceOne.getCalories());
		choiceTwo.setName("chips");
		choiceTwo.calories = 150;
		System.out.println(choiceTwo.canEat());
	}
}
```

✓
✓
✓
✓
✓
✗
✗

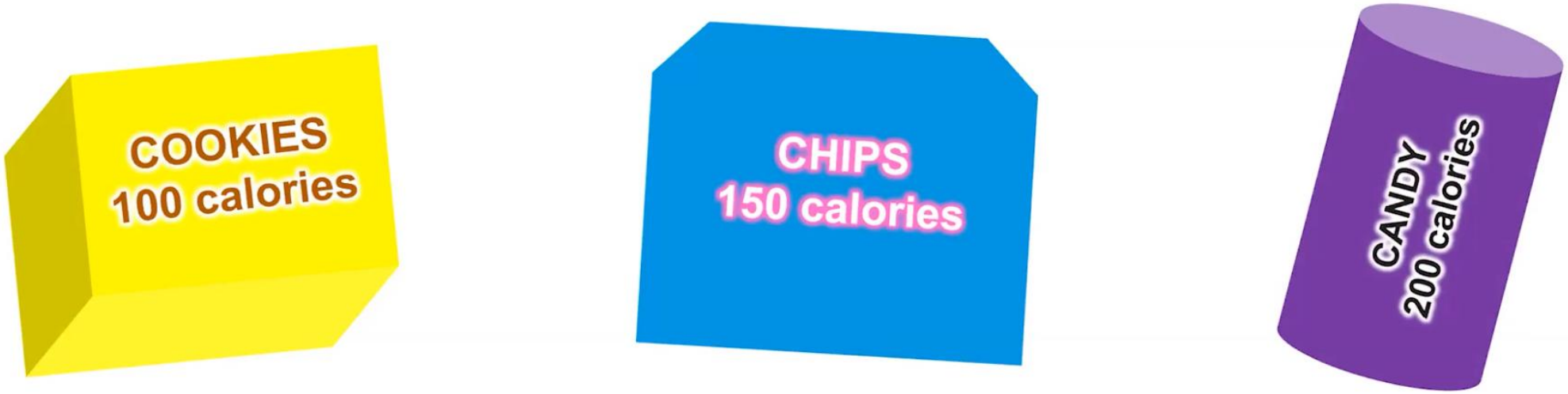
Encapsulation

- A fundamental concept of object-oriented programming
- Wrap the data (variables) and code that acts on the data (methods) in one unit (class)
- In AP CS-A, we will do this by:
  - Writing a class
  - Declaring the instance variables as **`private`**
  - Providing accessor (get) methods and modifier (set) methods to view and modify variables outside of the class

Instance Variables

“Has-a”

- Each instance of a class (object) “has-a” private instance variable.
- Recall the `Snack` class



Snack Graphics © Czajka, 2020

Snack Class

- Each instance of **Snack** "has-a"
  - -name
  - -number of calories

Compete the Constructors for the Snack Class

```
public class Snack {

	private String name;
	private int numCalories;

	public Snack() {
		name = "";
		numCalories = 0;
	}

	public Snack(String n, int nc) {
		name = n;
		numCalories = nc;
	}
}
```

Snack Class

- Each instance of **`Snack`** “has-a”
  - -name
  - -number of calories

Remember：

if you don‘t initialize an instance variable， the compiler provides reasonable **default** values for primitive variables (`0` for numbers, `false` for booleans) and provides `null` for reference

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

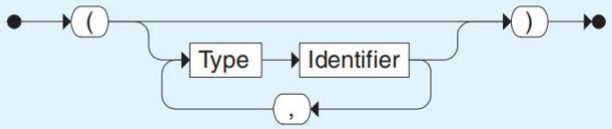
- The programmer does not have to define a constructor for a class

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 }
```

Constructor

```
public class Sport {

        private String name;
        private int numAthletes;

        public Sport(){
                name = "";
                numAthletes = 0;
        }

        public Sport(String n, int numAth){
                name = n;
                numAthletes = numAth;
        }

        public void setName(String n){
                name = n;
        }
}
```

Constructors are used to initialize the attributes for an object.

The constructor that is used to set the state depends on the way the object is instantiated.

Only one constructor will be used to set the initial state of the instance variables.

```
Sport tbd = new Sport( );
```

```
Sport wp = new Sport("Water Polo", 14);
```

Constructor

```
public class Sport {

        private String name;
        private int numAthletes;

        public String getName(){
                return name;
        }

        public int getNumAthletes(){
                return numAthletes;
        }
}
```

If no constructor is provided, Java provides a default constructor.

All instance variables are set to default values:
int – 0
double – 0.0
Strings and other objects – null

Be careful! This may cause a null pointer exception when other methods are called.

Getter (Accessor) Methods

- Allow safe access to instance variables
- Often referred to as *get methods*
- Sometimes referred to as *getters*
- If there is any need for a different class to access the instance variables, accessor methods are necessary.

Getter (Accessor) Methods

Visibility, Return Type, and Signature

- Must be `public`
- Return type must match the type of the instance variable to be accessed
- Name is often getNameOfVariable
- No parameters

```
public String getName()

public int getCalories()
```

Getter (Accessor) Methods

Be careful in your calls to accessor methods!

```
public class PetTester {
    public static void main(String[] args) {
        Pet p = new Pet("Abu", "cat", 14);
        p.getName();
        p.getTypeOfPet();
        p.getAge();
      }
    }
}
```

Accessor Methods

Non-void methods return a value that is the same type as the return type in the signature. To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression.

Returning a value is not the same as printing a value!
You must print the returned values.

```
public class PetTester {
    public static void main(String[] args) {
        Pet p = new Pet("Abu", "cat", 14);
        System.out.println(p.getName());
        System.out.println(p.getTypeOfPet());
        System.out.println(p.getAge());
       }
    }
}
```

```
Abu
cat
14
```

Setter (Mutator) Method

- Allow the change of values for instance variables outside of the class
- Often referred to as *set methods*
- Sometimes referred to as *setters*
- If there is any need for a different class to modify the instance variables, mutator methods are necessary.

Setter (Mutator) Method

Visibility, Return Type, and Signature

- Must be `public`
- Return type must be `void`
- Name is often setNameOfVariable
- Parameter type must match the type of the instance variable to be modified

```
public void setName(String n)

public void setCalories(int c)
```

Setter (Mutator) Method

Here is an example of calling on mutator methods and accessor methods:

```
public class PetTester {
    public static void main(String[] args) {
        Pet p = new Pet();
        p.setName("Gus");
        p.setTypeOfPet("cat");
        p.setAge(12);
        System.out.println(p.getName());
        System.out.println(p.getTypeOfPet());
        System.out.println(p.getAge());


    }
}
```

```
Gus
cat
12
```

Writing Method

A Method Header

A method header consists of 5 parts:

- Access level – set by an access modifier: **`public`** or **`private`**
- Ownership – set by whether or not **`static`** is included
- Return type – the data type of the value returned by the method, can be primitive, reference or void (when no value is returned)
- Identifier – the name of the method, should be meaningful
- Parameter list – enclosed in parentheses, states the data type and identifier for each parameter used in the method
  - A parameter is information needed by the method to complete its task
  - If the method does not use parameters, the parentheses are still needed but are left empty

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

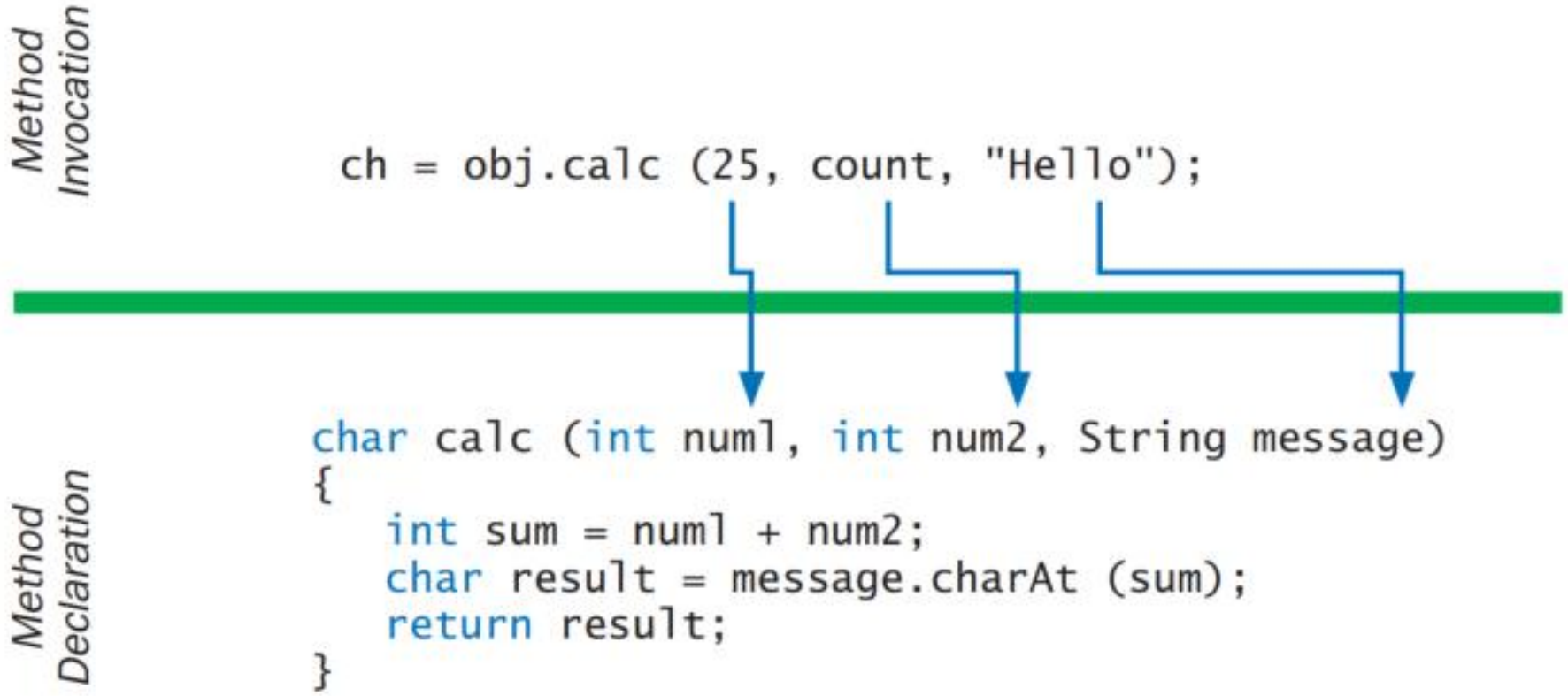
- A *return statement* specifies the value that will be returned

```
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

```
//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;
}
```

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
- initialize before using local variables
- 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

Find errors

- initialize before using local variables

```
public class test
{
    // 实例变量 - 用你自己的变量替换下面的例子

    public static void main(String[] args)
    {
        // 在这里加入你的代码
        //int i=0;
        int b=5;
        String a;
        String c="happy";
        for (int i=0;i<b;i++){
            a=c.substring(i,i+1);
        }
        System.out.println(a);
    }
}
```

可能尚未初始化变量a

```
int ha=1;
int def;
if(ha>0){def=0;}
System.out.println(def);
```

可能尚未初始化变量def

modified

```
        int ha=1;
        int def;
        if(ha>0){def=0;}
        else {def=1;}
        System.out.println(def);
    }
}
```

编译完成 - 没有语法错误

local variable & instance variable

difference	instance variable	local variable
scope	class scope	method scope
extend	extends from the opening brace to the closing brace of the class definition	extends from the point where it is declared to the end of the block in which its declaration occurs
initialization	if you don‘t initialize an instance variable ， the compiler provides reasonable default values for primitive variables (0 for numbers, false for booleans） and provides null for reference	If you fail to initialize a local variable in a method before you use it, you will get a compile-time error.
access modifier	public:can be access by arbitrary instance of this class private: can only be access in this class file	can’t be declared as private or public
others		Local variables take precedence over instance variables with the same name. (You should avoid usingthe same name,or use this keyword)

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.

- Static methods cannot reference instance variables, because instance variables don't exist until an object exists

- However, a static method can reference static variables or local variables

- Static variables are used with the class name and the dot operator, since they are associated with a class, not objects of a class. eg,

```
Math.sqrt (25)
```

The static modifier

- Static variables are also called *class variables*
- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists
- ```
  private static float price;
  ```
- Memory space for a static variable is created when the class in which it is declared is loaded
- All objects created from the class share static variables
- The most common use of static variables is for constants
  ```

The static modifier

- 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.

instance variable VS static variable

difference	instance variable	static variable
scope	class scope	class scope
attached to	object	class
values	for different object in this class, the instance variable may assign different values	all the object in this class share the same value

Comments

- Ignored by compilers/interpreters
- Help make code more readable
- Prevent execution when testing alternative code
- Not required on AP Exam FRQs but are an important habit
- Helpful in reading FRQs
- Software development—team effort
  - communicate among programmers
  - maintain the code for years

Comments

- There are three types of comments

```
// Single-line comment

/* Multi-line
   comment */

/** Documentation
 * comment
 * to create Javadoc
 */
```

Java has a tool called Javadoc that comes with the Java JDK and will pull out all of these comments to make documentation of the class in the form of a web page.

There are tags that can be used in Javadoc.

pay attention to the tag:
@return
@param

`@param`	Inserts a "Parameters" section, which lists and describes parameters for a particular constructor/method.
`@return`	Inserts a "Returns" section, which lists and describes any return values for a particular constructor/method. Use: `@return description`. An error will be thrown if included in a comment of a method with the void return type.

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

Preconditions

- Comments for a method
- Conditions that must be true for the method to work
  - Often a guarantee about a state of a parameter
- Code in method is based on preconditions being true
  - You do **not** need to check the preconditions in the method!
- Found in documentation comments before the method on AP FRQs

Postconditions

- Comments for a method
- Conditions that must be true after the method is executed
  - What is the outcome
  - State of instance variables
- Code should be written that meets the postconditions
  - Be sure you do not violate the postconditions in your code!
- Found in documentation comments before the method on AP FRQs

what is inheritance

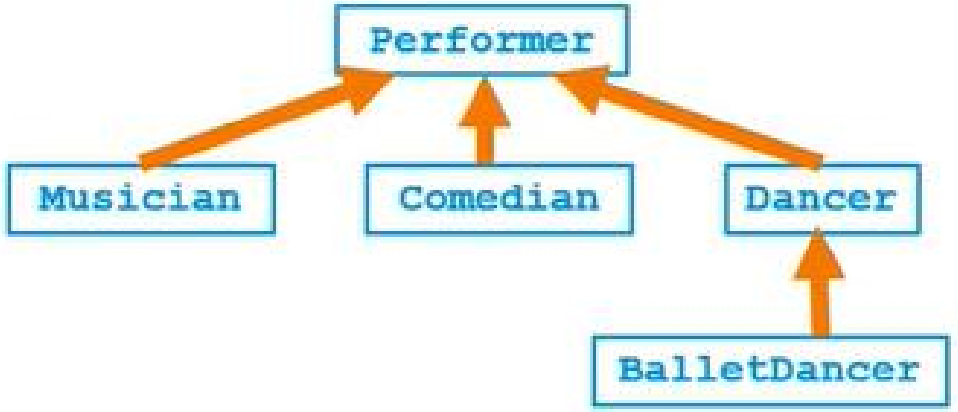
- you may have heard of someone coming into an inheritance, which often means they were left something from a relative who died. Or, you might hear someone say that they have inherited musical ability from a parent.
- In Java all classes can inherit attributes (instance variables) and behaviors (methods) from another class. The class being inherited from is called the parent class or superclass. The class that is inheriting is called the child class or subclass. Classes can be organized into inheritance hierarchies. Every subclass is-a or is a kind of the superclass.

While a person can have two parents, **a Java class can only inherit from one superclass**. If you do not point to the superclass explicitly when you declare a class then the class will **inherit from the Object class** that is already defined in Java.

Why do we use inheritance in java?

- Code reusability
  Higher-level classes can be used over and over again in many situations.
- Prevents repeating code
  Common methods and variables are now in one location, rather than many.
- Readability and organization
  Having a solid, organized structure of your classes and objects allows for greater readability and cohesion.
- Ease of maintenance
  Changing a general behavior in one place rather than many saves time and effort.

Superclass and subclass



- `Musician` is a subclass of the `Performer` superclass.
- `Comedian` is a subclass of the `Performer` superclass.
- `Dancer` is a subclass of the `Performer` superclass.
- `BalletDancer` is a subclass of the `Dancer` superclass.

How do we know which way the arrows go?

Using the "is-a" method!

- A `Musician` is a `Performer` — YES!
- A `Comedian` is a `Performer` — YES!
- A `Dancer` is a `Performer` — YES!
- A `BalletDancer` is a `Dancer` — YES!

- A `Comedian` is a `Dancer` — NO!
- A `Performer` is a `Musician` — NO!
- A `Performer` is a `Ballet Dancer` — NO!
- A `Dancer` is a `Ballet Dancer` — NO!

Writting subclass

- Using keywords : **extends**

superclass

```
public class Performer{……}
```

subclass

```
public class Musician extends Performer{……}

public class Comedian extends Performer{……}

public class Dancer extends Performer{……}

public class BalletDancer extends Dancer{……}
```

Writting subclass

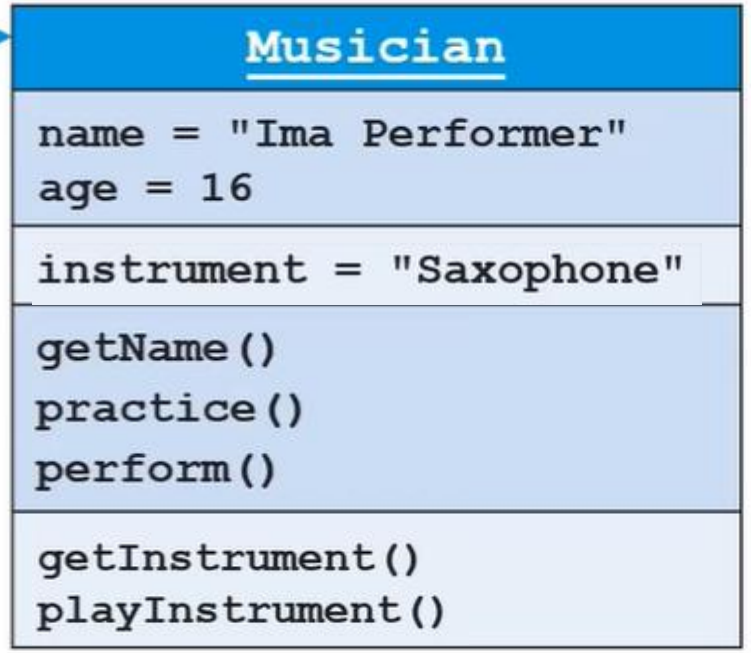
- The instance variables and non-constructor methods of superclass will be inherited
- the constructors are **not** inherited
- Calling Superclass Constructors
  - Using `super()` will call the no-argument constructor of the superclass.
  - If you don't include a call to `super()` in your constructor, Java inserts a call to the superclass's no-argument constructor.
  - If you **do** include a call to `super()` in your constructor, it must be the **first line** of the body of the constructor.
  - Whether the superclass constructor is called implicitly or explicitly, the process of calling superclass constructors continues up the inheritance hierarchy, with each constructor calling the constructor of its superclass until the `Object` constructor is called (the `Object` class is covered more in **Topic 9.7**).

```java
public class Performer {
    private String name;
    private int age;
    public Performer() {
        name = "Ima Performer";
        age = 16;
    }
    public Performer(String n, int a) {
        name = n;
        age = a;
    }
```

```java
public class Musician extends Performer{
    private String instrument;
    public Musician() {
        instrument = "Piano";
    }
    public Musician(String inst) {
        instrument = inst;
    }
}
```

```java
Musician branford = new Musician("Saxophone");
```

branford = 



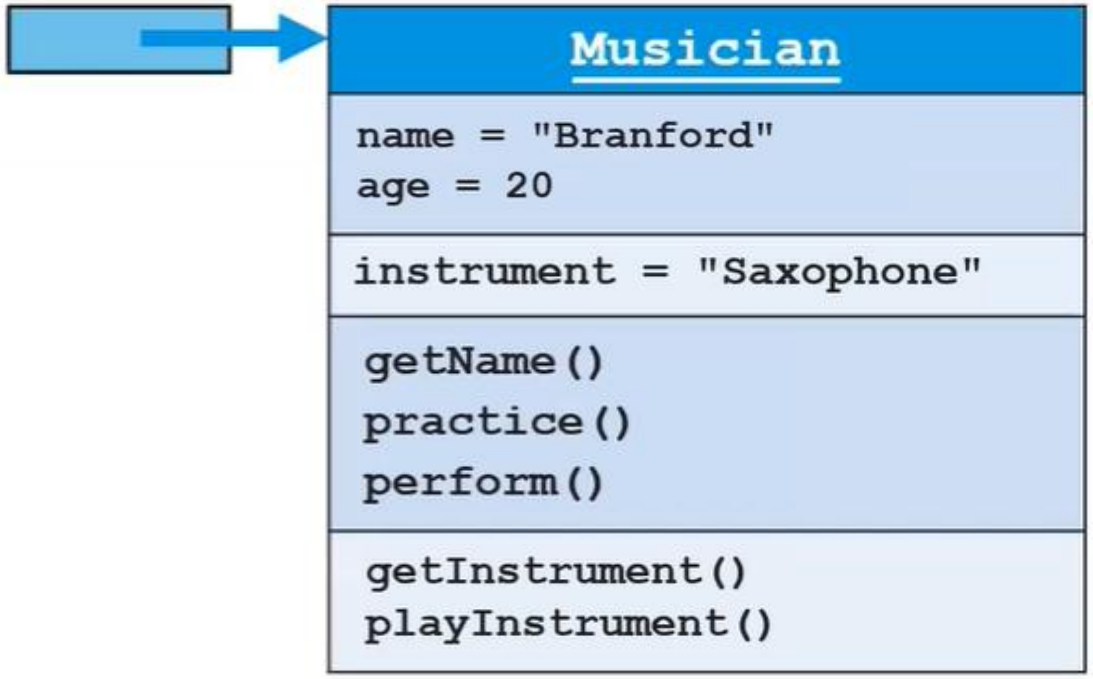
By default, the no-argument constructor of the superclass is called.

```java
public class Performer {
    private String name;
    private int age;
    public Performer() {
        name = "Ima Performer";
        age = 16;
    }
    public Performer(String n, int a) {
        name = n;
        age = a;
    }
```

```java
public class Musician extends Performer{
    private String instrument;
    public Musician() {
        instrument = "Piano";
    }
    public Musician(String inst) {
        instrument = inst;
    }
    public Musician(String n, int a, String inst){
        super(n, a);
        instrument = inst;
    }
}
```

```java
Musician branford =
    new Musician("Branford", 20, "Saxophone");
```

branford =



Iheritance and Methods

What are our options with methods when we extend a superclass?

- **Inherit methods**: Any public methods in the superclass become valid public methods of the subclass. These are especially important to access private instance variables of the superclass.

- **Write new methods**: The subclass can have additional methods that are completely independent of methods in the superclass. This includes methods that are overloaded (same method name, but different signatures) and treated as independent methods.

- **Override methods**: Write a new and different implementation of a method that already exists in the superclass.

Overriding a Method

```
public class Performer {
    private String name;
    private int age;
    public Performer(String n, int a) { ... }
        // Other constructors not shown
    public String getName(){
        return name;
    }
    public void practice() {
        System.out.println("Honing my craft!");
    }
    public void perform() {
        System.out.println(
            "Performing for an audience!");
    }
}
```

Inherit method

```
public class Comedian extends Performer{
    private ArrayList<String> jokes;

    public Comedian(String n, int a) {...}
        // Other constructors not shown

    public void writeJokes() { ... }
        // adds jokes to the ArrayList

    public void perform() {
        for (String joke : jokes)
            System.out.println(joke);
    }
}
```

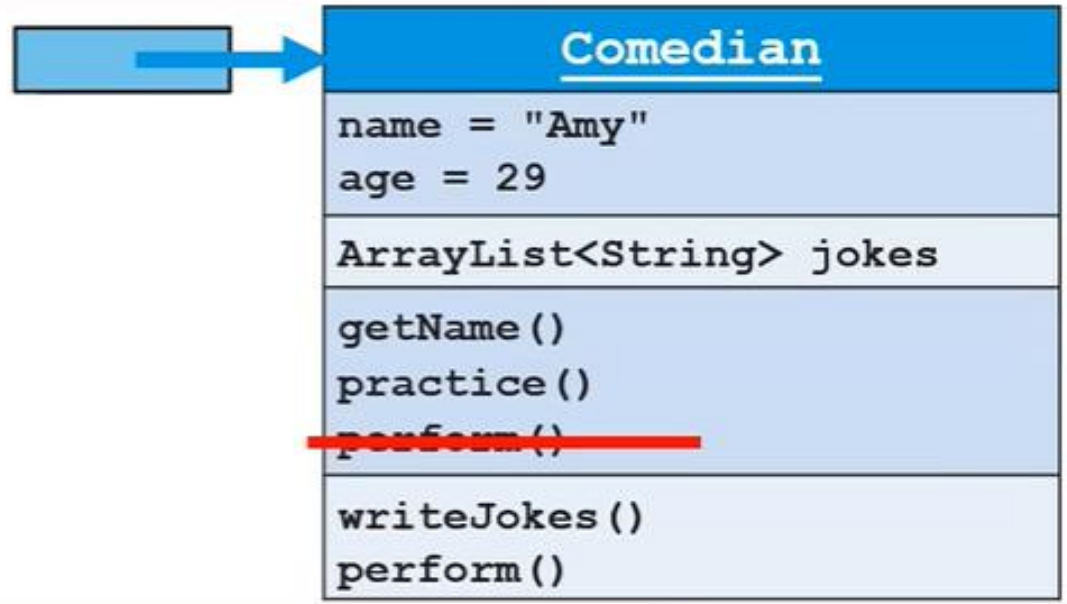
New method

Override method

```
public class Performer {
    public String getName(){... }
    public void practice() {... }
    public void perform() {... }
```

```
public class Comedian extends Performer{
    public void writeJokes(){... }
```

amy =




```
public static void main(String[] args)
{
    Comedian amy = new Comedian("Amy", 29);
    System.out.println("My name is " + amy.getName());
    amy.writeJokes();
    amy.practice();
    amy.perform();
}
```

```
My name is Amy
Honing my craft!
My software doesn't have bugs - it has "unintentional features"!
I did great on my 15-point hexadecimal test - I got an "F"!
How do you cut a program in half?  With a C-saw!
```

Sometimes we hope to call the superclass' s method， what should we do?

```
public class Dancer extends Performer{
    // Rest of class not shown
    public void perform() {
        System.out.println("Dancing on the stage!");
    }
}
```

```
public class BalletDancer extends Dancer{
    // Rest of class not shown
    public void jete(){
        System.out.println("Leaping...");
    }
    public void pirouette(){
        System.out.println("Spinning...");
    }
    public void perform() {
        jete();
        pirouette();
        super.perform();
    }
}
```

```
public static void main(String[] args)
{
    BalletDancer mikhail = new BalletDancer(...);
    mikhail.practice();
    mikhail.perform();
}
```

```
Honing my craft!
Leaping...
Spinning...
Dancing on the Stage!
_
```

The superclass method call does **not** have to be on the first line.

Overriding method &super key word

- Essential knowledge
- Method overriding occurs when a public method in a subclass has the same method signature as a public method in the superclass.
- Any method that is called must be defined within its own class or its superclass.
- A subclass is usually designed to have modified (overridden) or additional methods or instance variables.
- A subclass will inherit all public methods from the superclass; these methods remain public in the subclass
- The keyword super can be used to call a superclass’ s constructors and methods.
- The superclass method can be called in a subclass by using the keyword super with the method name and passing appropriate parameters.
- The superclass method call does not have to be the first line.

Polymorphism

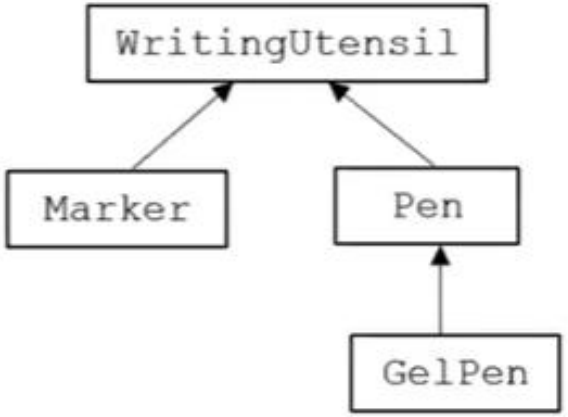
- A reference variable is **polymorphic** when it can refer to objects from different classes at different points in the code.
  - A reference variable can store a reference to its declared class or to any subclass of its declared class.
- A method is considered **polymorphic** when it is overridden in at least one subclass.
- **Polymorphism** is the act of executing an overridden non-`static` method from the correct class at runtime based on the actual object type.

- Utilize the Object class through inheritance.
- **At compile time**, methods in or inherited by the declared type determine the correctness of a non-static method call.
- **At run-time**, the method in the actual object type is executed for a non-static method call.

Inheritance Hierarchy

Consider the following class declarations:

```
public class WritingUtensil{}
public class Marker extends WritingUtensil{}
public class Pen extends WritingUtensil{}
public class GelPen extends Pen{}
```



In some other class, assuming each of the above classes contains a parameterless constructor:

```
WritingUtensil writer2 = new Marker();
WritingUtensil writer3 = new Pen();
Pen writer4 = new GelPen();
WritingUtensil writer5 = new GelPen();
```

Marker is-a WritingUtensil
Pen is-a WritingUtensil
GelPen is-a Pen
GelPen is-a WritingUtensil

is-a

```
GelPen bad = new Pen();
```

The power of polymorphic variables

Why would we declare a variable using a superclass if we plan to store a reference to a subclass object?

- A collection (array or **`ArrayList`**) needs to be declared as a datatype.
- What do we store in a pencil case?
  - **All** of our writing utensils!

```
WritingUtensil [] pencilCase = new WritingUtensil[3];
pencilCase[0] = new Pen();
pencilCase[1] = new GelPen();
pencilCase[2] = new Marker();
```

- This in turn supports polymorphism. Each of these writing utensils likely displays writing in a different way. Let's say the behavior is implemented through a method **`public void write(String text)`** in the **`WritingUtensil`** class, which is overridden appropriately in each subclass.

```
for(WritingUtensil wu: pencilCase)
    wu.write("Hello!");
```

result

```
Pen is writing Hello!
GelPen is writing Hello!
Marker is writing Hello!
```

Postcondition: write () method is overrided in subclass

Polymorphism & Method Parameters

Consider the following class hierarchy:

```
public class WritingUtensil{}

public class Marker extends WritingUtensil{}

public class Pen extends WritingUtensil{}

public class GelPen extends Pen{}
```

Marker is-a WritingUtensil
Pen is-a WritingUtensil
GelPen is-a Pen
GelPen is-a WritingUtensil

Now consider a method in a different class that contains a `WritingUtensil` parameter:

```
public void displayText(WritingUtensil wu, String text){
    //implementation not shown
}
```

Due to the inheritance relationship between these classes, this method can be given **any** of the above objects!

```
WritingUtensil wtg = new WritingUtensil();
displayText(wtg, "hello");

GelPen gp = new GelPen();
displayText(gp, "hello");
```

```
Marker m = new Marker();
displayText(m, "hello");

Pen p = new Pen();
displayText(p, "hello");
```

which method calls compile?
which methods execute?

```
public class Entertainer{
    private String talent;
    public Entertainer (String t){
        talent = t;
    }
    public String getTalent(){
        return talent;
    }
}

public class Comedian extends Entertainer{
    private ArrayList<String> jokes;
    public Comedian(String t, ArrayList<String> jks){
        super(t);
        jokes = jks;
    }
    public String getTalent(){
        return "Comedy style: " + super.getTalent();
    }
    public String tellJoke(){
        return jokes.get((int)(Math.random()*jokes.size()));
    }
}
```

```
Entertainer fred = new Entertainer("musician");
System.out.println(fred.getTalent());
System.out.println(fred.tellJoke());
```

- since fred was declared as Entertainer type, and it also references an Entertainer.
  - ✓ at compile time, the compiler make sure that `getTalent()` exists in the Entertainer class
  - ✓ At runtime getTalent() is executed from the Entertainer class
- what about `telljoke()`, can it compile?

Authored by: Sage Miller, 2020

which method calls compile? which methods execute?

```
public class Entertainer{
    private String talent;
    public Entertainer (String t){
        talent = t;
    }
    public String getTalent(){
        return talent;
    }
}

public class Comedian extends Entertainer{
    private ArrayList<String> jokes;
    public Comedian(String t, ArrayList<String> jks){
        super(t);
        jokes = jks;
    }
    public String getTalent(){
        return "Comedy style: " + super.getTalent();
    }
    public String tellJoke(){
        return jokes.get((int)(Math.random()*jokes.size()));
    }
}
```

downcast sally to `Comedian`, so it can compile, then during the run-time, it will call the `tellJoke()` method in `Comedian`

```
ArrayList<String> oneLiners = new ArrayList<String>();
//code to add jokes to oneLiners
Entertainer sally = new Comedian("satire", oneLiners);
System.out.println(sally.getTalent());
System.out.println(((Comedian)sally).tellJoke());
```

- since sally was declared as `Entertainer` type, and it also references an `Comedian`.
  - ✓ at compile time, the compiler make sure that `getTalent()` exists in the Entertainer class
  - ✓ At runtime `getTalent()` is executed from the Comedian class
- what if asked sally to tell joke?

downcasting

the toString method

- The **default** `toString` method is declared in the `Object` class, it will transform the parameters into `String` type.
- when you call the `System.out.print` or `System.out.println`, the `toString` method is called also.
- If System.out.print or System.out.println is passed an **`object`**, it will printed the reference,
- 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.*(if you didn't override the toString method ,the default one will call ,and you'll get the reference of the object printed)*

Object's toString()

when a class doesn't not explicitly extend another class, then it implicitly extends object.

```
public class WritingUtensil extends Object
{
    private int tipSize;
    public WritingUtensil(int pixels){
        tipSize = pixels;
    }
}
```

when an object is passed as a parameter to the print() or pirntln() method, the object's toString() method is implicitly called.

```
WritingUtensil@2a139a55
_
```

In a different class:

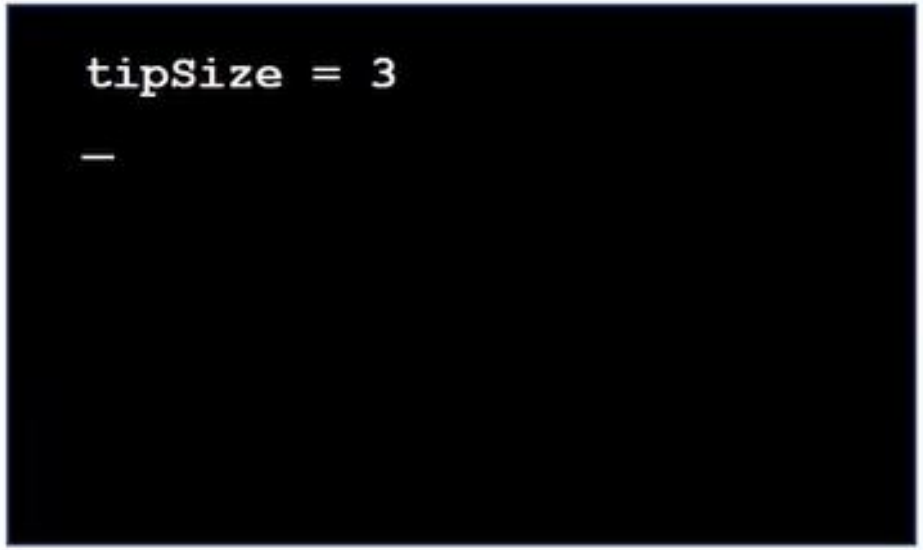
```
WritingUtensil wu = new WritingUtensil(3);
System.out.println(wu.toString());
```

Object's toString()——overriding toString()

```
public class WritingUtensil
{
    private int tipSize;
    public WritingUtensil(int pixels){
        tipSize = pixels;
    }
    public String toString(){
        return "tipSize = " + tipSize;
    }
    //additional methods not shown
}
```

In a different class:

```
WritingUtensil wu = new WritingUtensil(3);
System.out.println(wu);
```



A subclass toString()

```
public class Marker extends WritingUtensil{
    private String tipType;
    public Marker(int pixels, String type){
        super(pixels);
        tipType = type;
    }
}
```

Because `Marker` does not override `toString()`, the `toString()` from `WritingUtensil` is inherited.

```
tipSize = 5
_
```

In a different class:

```
Marker m = new Marker(5, "chisel");
System.out.println(m);
```

But Marker m also has tipType, so sometimes we also need to display other instance variables,how to override the toString() method?

A subclass toString()

```
public class Marker extends WritingUtensil{
    private String tipType;
    public Marker(int pixels, String type){
        super(pixels);
        tipType = type;
    }
    public String toString(){
        String str = super.toString();
        str += "\n";
        str += "tipType = " + tipType;
        return str;
    }
}
```

```
tipSize = 5
tipType = chisel
_
```

In a different class:

```
Marker m = new Marker(5, "chisel");
System.out.println(m);
```

Authored by: Sage Miller, 2021

the equals method

- The == operator compares object references for equality, returning `true` if two object have the same reference and values
    - `bishop1 == bishop2`
- A method called `equals` is defined for all objects, but unless we override it when we write a class, it has the same semantics as the == operator
    - `bishop1.equals(bishop2)`
- We can override the `equals` method to return `true` under whatever conditions we think are appropriate

Object's equals()

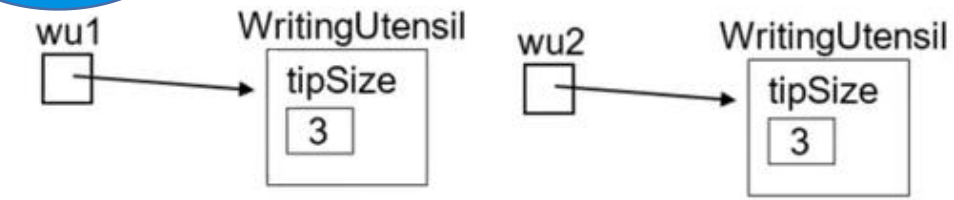
```
public class WritingUtensil
{
    private int tipSize;
    public WritingUtensil(int pixels){
        tipSize = pixels;
    }
}
```

Even though an `equals()` method is not explicitly defined in the `WritingUtensil` class, it can be called from a `WritingUtensil` object because `equals()` is inherited from `Object`.

```
false
_
```

In a different class:

```
WritingUtensil wu1 = new WritingUtensil(3);
WritingUtensil wu2 = new WritingUtensil(3);
System.out.println(wu2.equals(wu1));
```



Object's equals()

```
public class WritingUtensil
{
    private int tipSize;
    public WritingUtensil(int pixels){
        tipSize = pixels;
    }

    public boolean equals(Object other){
        if (!(other instanceof WritingUtensil))
            return false;
        WritingUtensil that = (WritingUtensil) other;
        return this.tipSize == that.tipSize;
    }
    //additional methods not shown
}
```

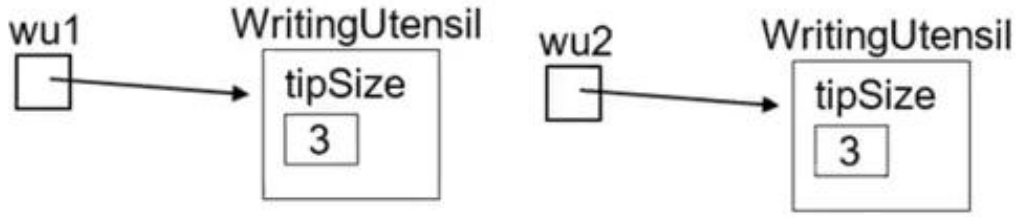
The `instanceof` operator is not tested on the AP exam

```
true

_
```

In a different class:

```
WritingUtensil wu1 = new WritingUtensil(3);
WritingUtensil wu2 = new WritingUtensil(3);
System.out.println(wu2.equals(wu1));
```



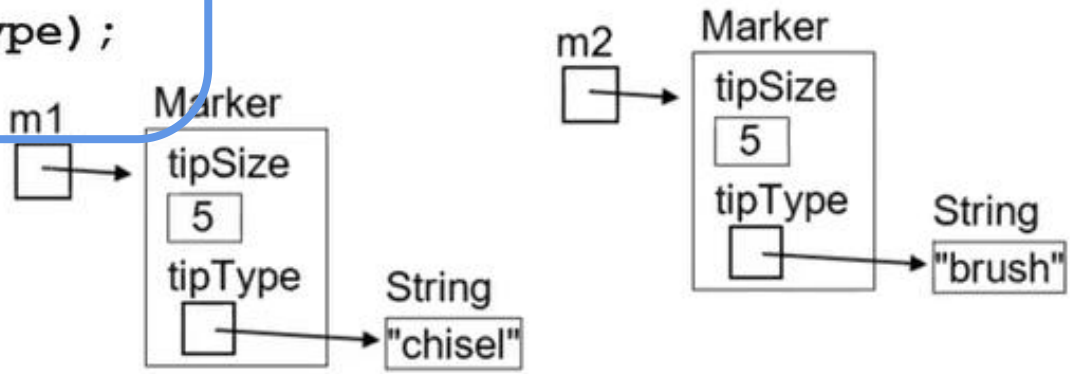
A Subclass equals()

```
public class Marker extends WritingUtensil{
    private String tipType;
    public Marker(int pixels, String type){
        super(pixels);
        tipType = type;
    }
    public boolean equals(Object other){
        if(!(other instanceof Marker))
            return false;
        Marker that = (Marker)other;
        return super.equals(other)
            && this.tipType.equals(that.tipType);
    }
}
```

In a different class:

```
Marker m1 = new Marker(5, "chisel");
Marker m2 = new Marker(5, "brush");
System.out.println(m1.equals(m2));
```

```
false
_
```



Authored by: Sage Miller, 2021