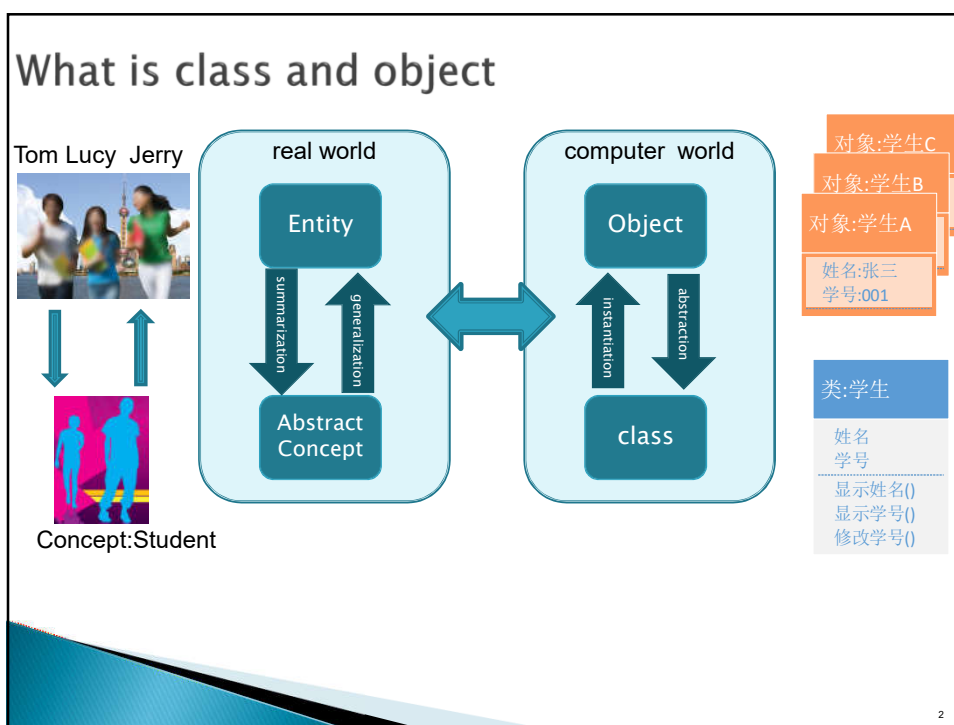


Chapter 3 Classes and Objects



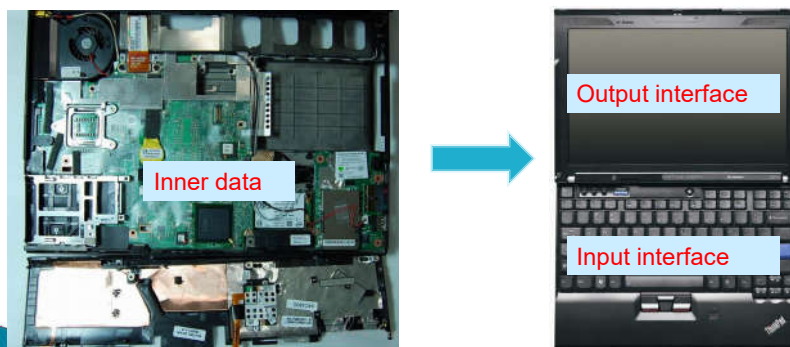
Key Elements of OOP

- ▶ Encapsulation
 - ▶ a programmer can bundle data and methods into a single entity
- ▶ Inheritance
 - ▶ the ability to use the data and methods of one kind of object by another
- ▶ Polymorphism
 - ▶ an object that inherits appears to be both kinds of object

3

Encapsulation

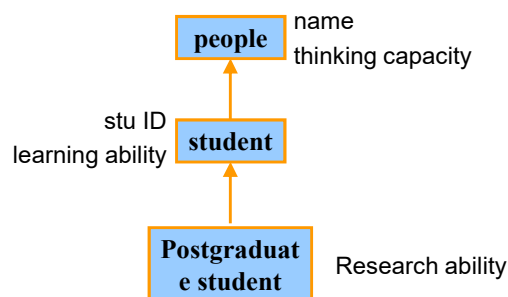
- ▶ binding together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse
- ▶ lead to **data hiding, high cohesion and low coupling**



4

Inheritance

- ▶ Inheritance in most class-based object-oriented languages is a mechanism in which one object acquires all the properties and behaviors of the parent object
- ▶ Improve code level **reusability**, lead to **incremental** programming



5

Polymorphism

- ▶ A polymorphic type is one whose operations can also be applied to values of some other type, or types
- ▶ Improve interface level reusability



6

Advantages of Object-oriented

- ▶ Improved software-development productivity
- ▶ Improved software maintainability
- ▶ Faster development
- ▶ Lower cost of development
- ▶ Higher-quality software
 - modularity, extensibility, and reusability

7

Disadvantages of Object-oriented

- ▶ Steep learning curve
- ▶ Larger program size
- ▶ Slower programs
- ▶ Not suitable for all types of problems

8

面向对象环卫车

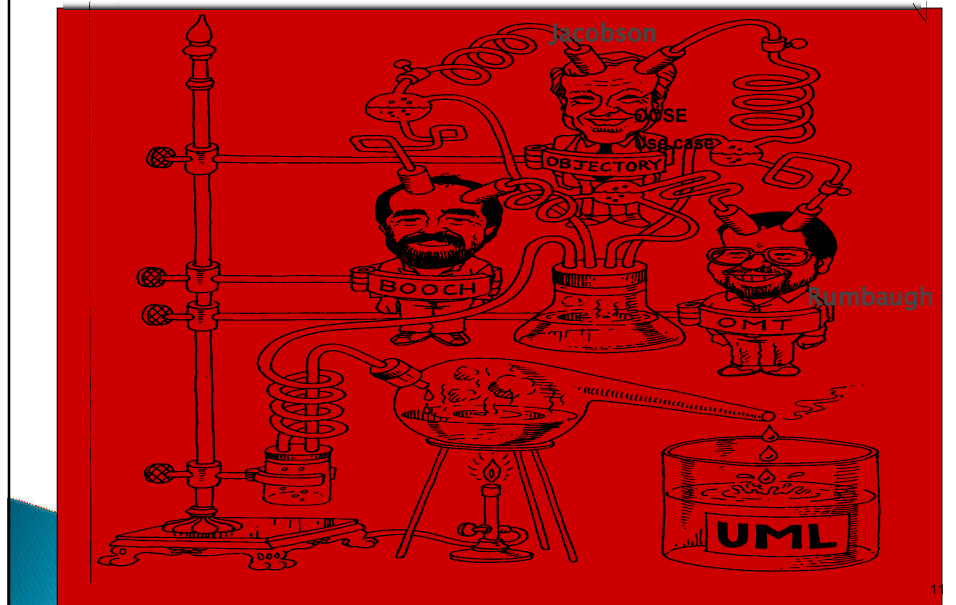


- ▶ 封装：无需知道如何运作，开动即可
- ▶ 继承：继承自拖拉机，实现扫地接口
- ▶ 多态：上班扫地，下班代步
- ▶ 重用：重复利用发动机的能量
- ▶ 多线程：多个扫把同时工作
- ▶ 低耦合：扫把可以换成拖把
- ▶ 面向构件：每个配件都可独立使用
- ▶ 适配器模式：只取拖拉机的动力方法
- ▶ 代码托管：无需垃圾管理，扫到路边即可

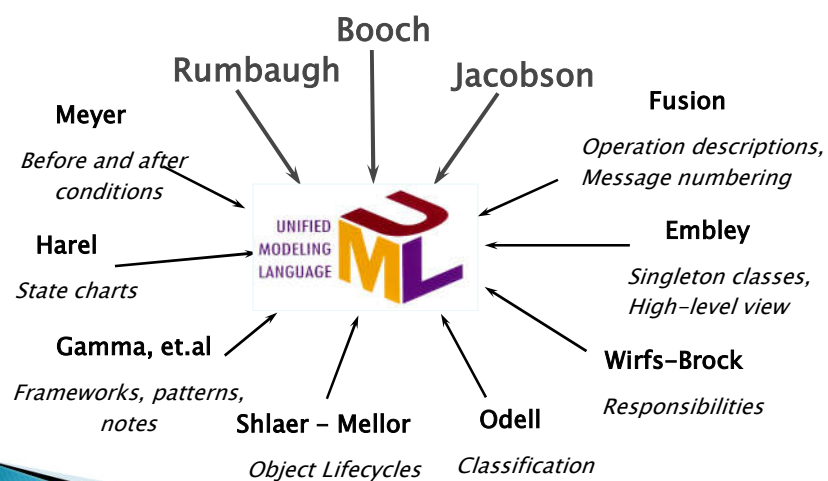
Object-Orientation and the UML

- ▶ **UML** is a modeling language used to model software and non-software systems. Although **UML** is used for non-software systems, the emphasis is on modeling OO software applications. Most of the **UML** diagrams discussed so far are used to model different aspects such as static, dynamic, etc.

History of UML



History of UML



Class Declaration

- ▶ An **object** has a unique identity, state, and behaviors
- ▶ The **state** of an object is represented by field (also known as member variables) with their current values
- ▶ The **behavior** of an object is defined by a set of methods

13

Class Declaration

- ▶ All the qualities of a class are specified in the class declaration, which consists of
 - optional **fields** (attributes, member variables),
 - optional **methods** (member methods),
 - optional **constructors**,
 - and optional **member classes** (inner classes).

14

Class Declaration

Car
name: String height, width: double x, y: int direction: byte
moveAhead(int distance) turnRight() turnLeft()

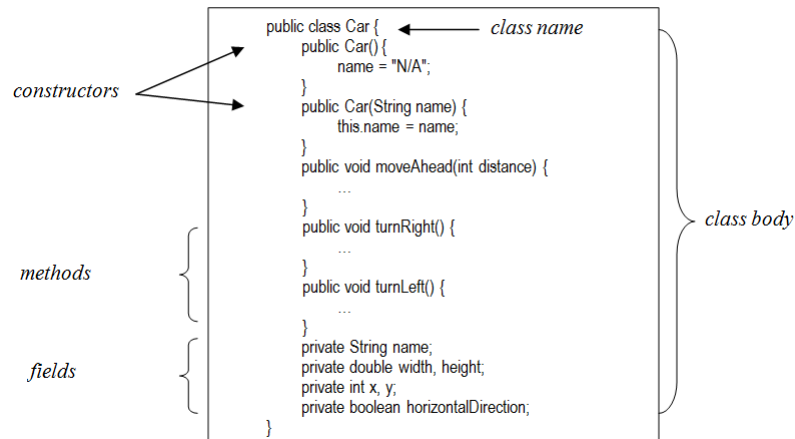
15

Class Declaration

- ▶ There are several kinds of variables in a class:
 - **member variables** in a class—these are also called fields
 - Variables in a method or block of code—these are called **local variables**
 - Variables at the head of a method—these are called **parameters**
- ▶ A class is a template for multiple objects with similar attributes and behaviors

16

Components of the Car class



17

Method declaration

- ▶ The structure of a method includes a method signature and a code body:

```

<access_modifier> <return_type> <method_name> (<list_of_parameters>) {
    .....
}
  
```

- ▶ The method's name and the parameter types of a method declaration comprise the method signature.
- ▶ The method body, enclosed between braces, consists of the method's code and the declaration of local variables.

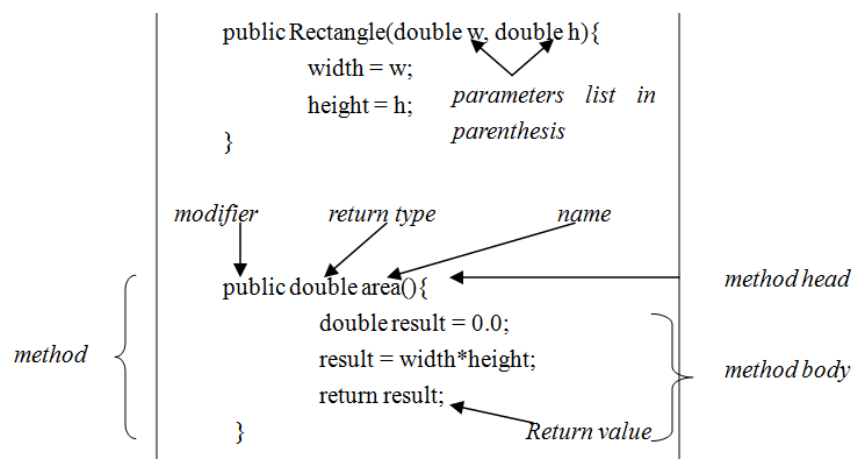
18

Constructor

- ▶ A constructor is called when an **instance** of this class is first created.
- ▶ It is used to initialize the member variables.
- ▶ The constructor signature looks similar to that of a regular method.
- ▶ It has a **name that matches the class name** and holds a list of parameters in parentheses.
- ▶ However, a constructor has **no return type**.
- ▶ Constructors are invoked using the **new** operator when an object is created.

19

Method declaration



20

Default constructor

- ▶ If there is no constructor in a class declaration, the Java compiler will create a **default constructor** with an **empty parameter list**.
- ▶ For example, if there is no constructor is defined in the class Car, JVM will use the default parameterless constructor.
- ▶ The default constructor calls the default parent constructor (super()) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).

Type	byte	Short	int	long	float	double	char	boolean	object
Initial value	0	0	0	0L	0.0f	0.0d	'\u0000'	false	null

21

Default constructor

- ```

▶ public class Test {
▶ public static void main(String[] args) {
▶ Car a = new Car();
▶ }
▶ }

▶ //Car.java
▶ class Car {
▶ String name;
▶ }

```

22

## Default constructor

```

▶ public class Test {
▶ public static void main(String[] args) {
▶ Car a = new Car(); //Compilation error
▶ }
▶ }

```

Once any constructor is defined in the class Car, there will be no constructor generated

```

▶ //Car.java
▶ class Car {
▶ public Car (String name) {
▶ this.name = name;
▶ }
▶ String name;
▶ }

```

23

## Initialization

```

▶ class Car1 {
▶ public Car1() {
▶ width = 1.0;
▶ height = 2.0;
▶ }
▶ private String name;
▶ private double width, height;
▶ private int x, y;
▶ }
▶ public class Test1{
▶ public static void main(String[]
args) {
▶ Car1 a = new Car1();
▶ }
▶ }

```

```

▶ class Car2 {
▶ private String name;
▶ private double width =
1.0, height = 2.0;
▶ private int x, y;
▶ }
▶ public class Test2 {
▶ public static void
main(String[] args) {
▶ Car2 a = new Car2();
▶ }
▶ }

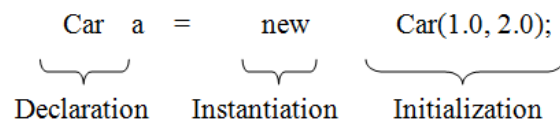
```

24

## Creating Objects

- ▶ An object is created from a class
- ▶ Each of these statements has three parts:
  - declaration,
  - instantiation
  - and initialization

`Car a = new Car(1.0, 2.0);`



Declaration      Instantiation      Initialization

25

## Creating Objects

- ▶ **Declaration** means to associate a variable name with an object type.
- ▶ **Instantiation** implies to create the object with the new operator.
- ▶ A constructor is called to initialize the new object when the **initialization** performs.

26

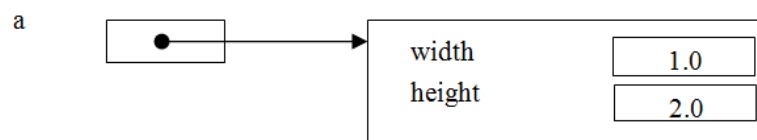
## create an object

- ▶ allocate memory space for the member variables of the object,
- ▶ execute the constructor
- ▶ and return the reference to the new object.

27

## create an object

- ▶ The statement:
- ▶ `a = new Car(1.0, 2.0);`
- ▶ can be read as: create an object of type Car and assign its reference to a.



28

## Accessing Objects via Reference Variables

- `<object_reference> . <method_name>(<arguments_list>)`
- ▶ or:
  - `<object_reference> . <method_name>()`
- ▶ Sending a message to that object is the same as invoking a method on a particular object.

29

## Accessing Objects

- ▶ Object member variables are accessed by their name within its own class
- ▶ Example,

```
class Car{
 double width;
 double height;
 public double area(){
 double result = 0.0;
 result = width*height;
 return result;
 }
}
```

30

## Accessing Objects

- Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a member variable name

```
class Test{
 Car a = new Car();
 System.out.println("Width of a: " + a.width);
 System.out.println("Height of a: " + a.height);
}
```

31

## A simple Javabeen

```
public class Car {
 private double height;
 private double width;
 private double length;
 public double getHeight() {
 return height;
 }
 public void setHeight(double height) {
 this.height = height;
 }
 public double getWidth() {
 return width;
 }
 public void setWidth(double width) {
 this.width = width;
 }
 public double getLength() {
 return length;
 }
 public void setLength(double length) {
 this.length = length;
 }
}

public class Test {
 public static void main(String[] args) {
 Car car = new Car();
 car.setHeight(1.0);
 car.Height = 1.0;
 }
}
```

encapsulation

32



## Object Reference this

- ▶ **this** is a reference to the current object which is the object whose method or constructor is being called.
- ▶ It refers to the newly created object in **constructors**, or refers to the object that a method belongs to in the method.

33

## this

- ▶ 

```
public class Car {
```
- ▶ 

```
 public Car(double width, double height) {
```
- ▶ 

```
 this.width = width;
```
- ▶ 

```
 this.height = height;
```
- ▶ 

```
 }
```
- ▶ 

```
 ...
```
- ▶ 

```
 private double width, height;
```
- ▶ 

```
}
```
- ▶ this is used to distinguish member variables from parameters in the constructor of the class Car declaration

34

## this

- ▶ public class Car {
  - ▶   public Car(double width, double height) {
  - ▶     this.width = width;
  - ▶     this.height = height;
  - ▶   }
  - ▶   public Car() {
  - ▶     this(0.0, 0.0);
  - ▶   }
  - ▶   ...
  - ▶   private double width, height;
  - ▶ }
- ▶ By this, a constructor can invoke another constructor
  - ▶ Note that this(0.0, 0.0); must be the first statement in the constructor in this case.

35

## Parameter Passing

- ▶ The list of variables in a method head declaration are the **parameters**
- ▶ The actual values that are passed when the method is invoked are called the **arguments**
- ▶ The arguments must match the corresponding parameters in type and order.

36

## The typical method call sequence

- ▶ 1. Evaluate arguments left-to-right.
  - If an argument is a simple variable or a literal value, there is no need to evaluate it.
  - When an expression is used, the expression must be evaluated before the call can be made.
- ▶ 2. When a method is called, a temporary piece of memory is required to store the following information:
  - parameter and local variable storage,
  - where to continue execution when the called method returns
  - and any other working storage needed by the method.

37

## The typical method call sequence

- ▶ 3. Initialize the parameters.
  - When the arguments are evaluated, they are assigned to the local parameters in the called method.
- ▶ 4. Execute the method.
  - Execution starts with the first statement and continues as normal.
- ▶ 5. Return from the method.
  - When a return statement is encountered, or the end of a void method is reached, the method returns.
  - For non-void methods, the return value is passed back to the calling method.
  - Execution is continued in the calling method immediately following where the call took place.

38

- ▶ Any changes to the values of the parameters exist only within the scope of the method
- ▶ However, in the case of reference type, the values of the object's fields can be changed in the method

39

## Example

```

▶ public class Car {
▶ public Car() {
▶ this(0, 0);
▶ }
▶ public Car(double width, double height) {
▶ this.width = width;
▶ this.height = height;
▶ }
▶ public double area() {
▶ return width * height;
▶ }
▶ public void setLocation(Point p){
▶ currentLocation = p;
▶ }
▶ public void setLocation(int x, int y){
▶ Point p = new Point(x, y);
▶ currentLocation = p;
▶ }
▶ private double width, height;
▶ private Point currentLocation;
▶ //The current location of this object
▶ }

▶ public class Point {
▶ public Point() {
▶ }
▶ public Point(int xValue, int yValue) {
▶ x = xValue;
▶ y = yValue;
▶ }
▶ // return x from coordinate pair
▶ public int getX() {
▶ return x;
▶ }
▶ // return y from coordinate pair
▶ public int getY() {
▶ return y;
▶ }
▶ public void setXY(int x, int y) {
▶ this.x = x;
▶ this.y = y;
▶ }
▶ public String toString() {
▶ return "[" + getX() + ", " + getY() + "]";
▶ }
▶ private int x; // x part of coordinate pair
▶ private int y; // y part of coordinate pair
▶ }

```

40

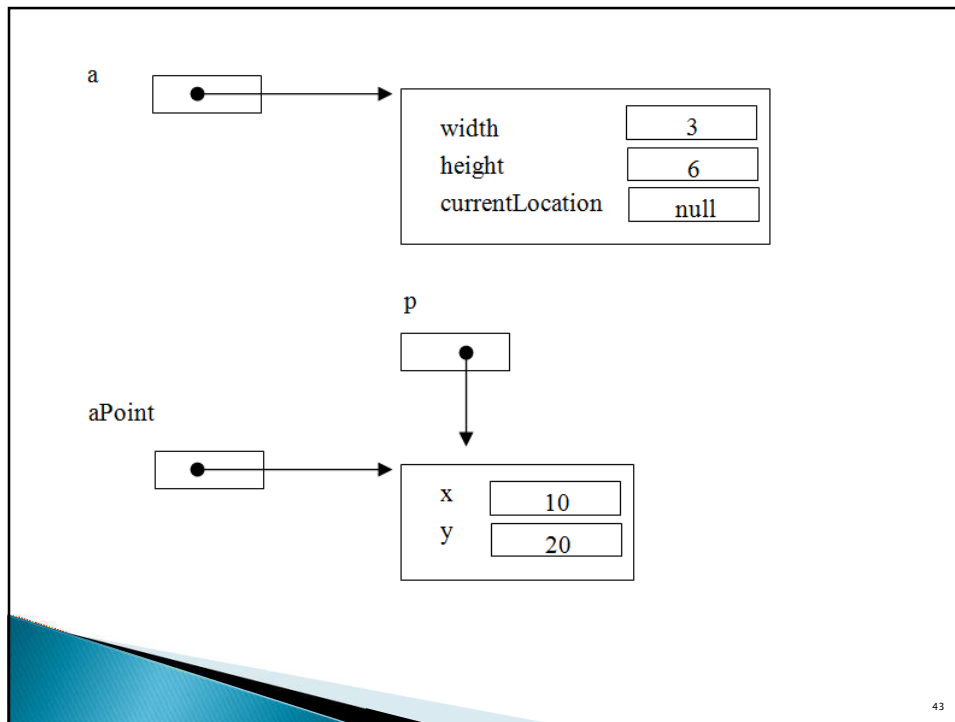
## parameter passing

- ▶ public class ParaDemo {
  - ▶   public static void main(String args[]) {
  - ▶     Car a = new Car(3,6);
  - ▶     a.setLocation(10, 20);
  - ▶   }
- ▶ After invoking setLocation(10, 20), the parameter x holds 10 and y holds 20.

41

- ▶ public class ParaDemo {
  - ▶   public static void main(String args[]) {
  - ▶     Car a = new Car(3, 6);
  - ▶     Point aPoint = new Point(10, 20);
  - ▶     a.setLocation(aPoint);
  - ▶   }
  - ▶ }
  - ▶ }
- ▶ Assume setLocation() is invoked as below.
  - ▶ public void setLocation(Point p){
  - ▶   >
  - ▶   currentLocation = p;
  - ▶ }

42

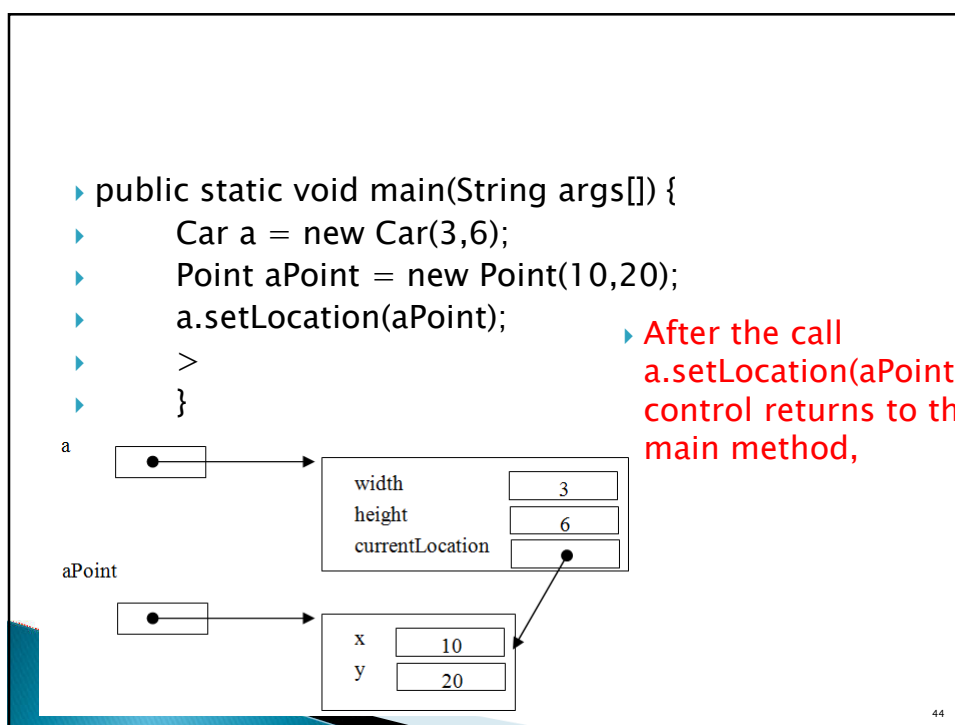


```

public static void main(String args[]) {
 Car a = new Car(3,6);
 Point aPoint = new Point(10,20);
 a.setLocation(aPoint);
 >
}

```

After the call `a.setLocation(aPoint)` control returns to the main method,



## pass-by-value

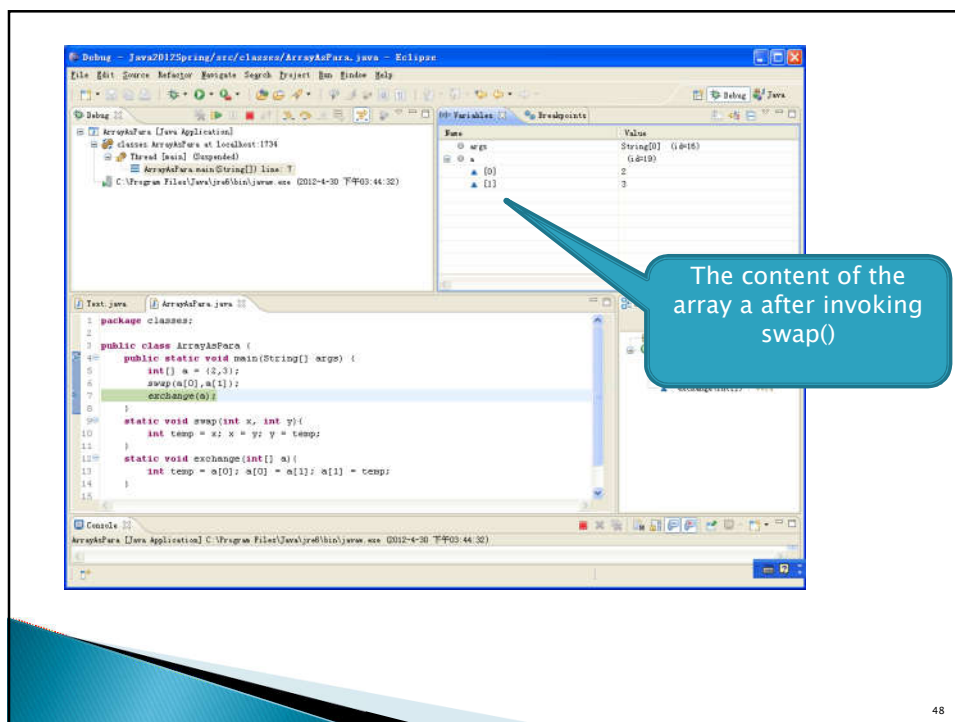
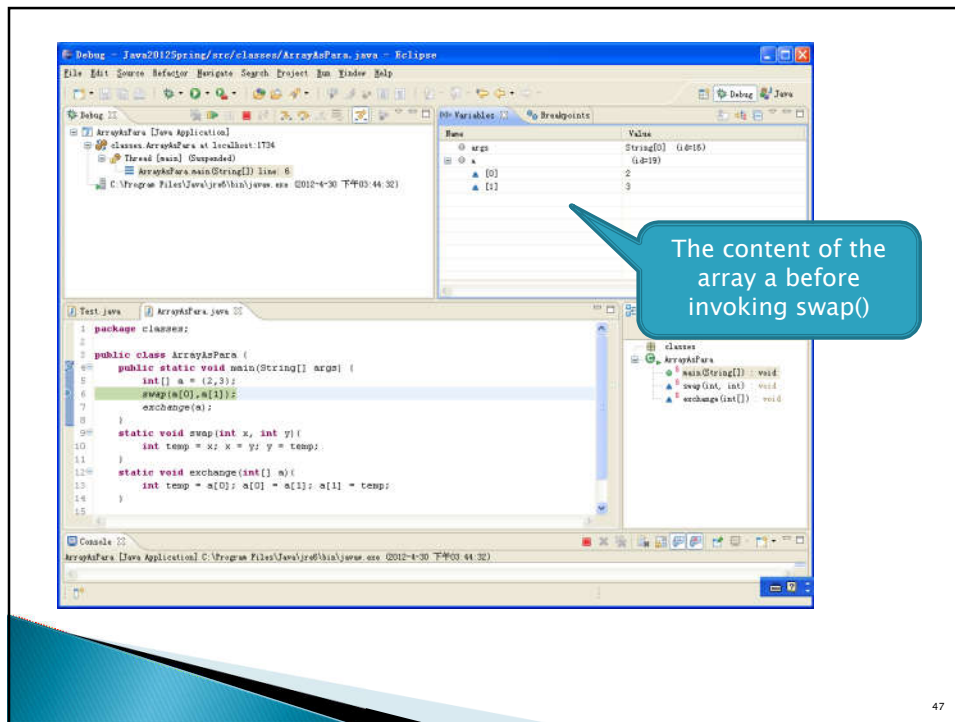
- ▶ For an argument of a primitive type, the argument's value is passed.
- ▶ For an argument of a reference type, the value of the argument contains a reference to an object and this reference is passed to the method.

45

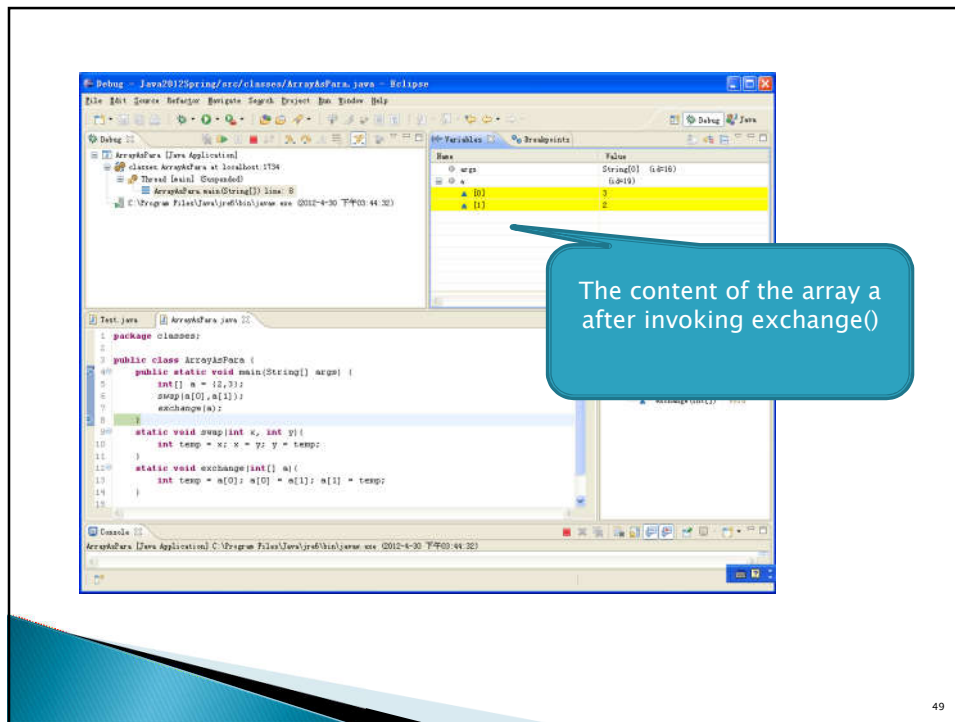
## Example

```
▶ public class ArrayAsPara {
▶ public static void main(String[] args) {
▶ int[] a = {2,3};
▶ swap(a[0],a[1]);
▶ exchange(a);
▶ }
▶ static void swap(int x, int y){
▶ int temp = x; x = y; y = temp;
▶ }
▶ static void exchange(int[] a){
▶ int temp = a[0]; a[0] = a[1]; a[1] = temp;
▶ }
▶ }
```

46







## Returning from methods

- ▶ void methods:
  - return;
- ▶ non-void methods:
  - return <value>;

## Example

```

public class ReturnDemo {
 public static void main(String[] args) {
 A a = new A();
 B b1 = a.getB();
 b1.say();
 }
}
class A {
 public B getB(){
 B b= new B();
 return b;
 }
}
class B{
 public void say(){
 System.out.println("I am B.");
 }
}

```

51

## Method overloading

```

public void setLocation(Point p){
 currentLocation = p;
}
public void setLocation(int x, int y){
 Point p = new Point(x, y);
 currentLocation = p;
}

```

Same name  
Different parameter list

polymorphism

52

## method overloading

- ▶ `java.io.PrintWriter.println()`
- ▶ `java.io.PrintWriter.println(boolean)`
- ▶ `java.io.PrintWriter.println(char)`
- ▶ `java.io.PrintWriter.println(char[])`
- ▶ `java.io.PrintWriter.println(double)`
- ▶ `java.io.PrintWriter.println(float)`
- ▶ `java.io.PrintWriter.println(int)`
- ▶ `java.io.PrintWriter.println(java.lang.Object)`
- ▶ `java.io.PrintWriter.println(java.lang.String)`
- ▶ `java.io.PrintWriter.println(long)`

53

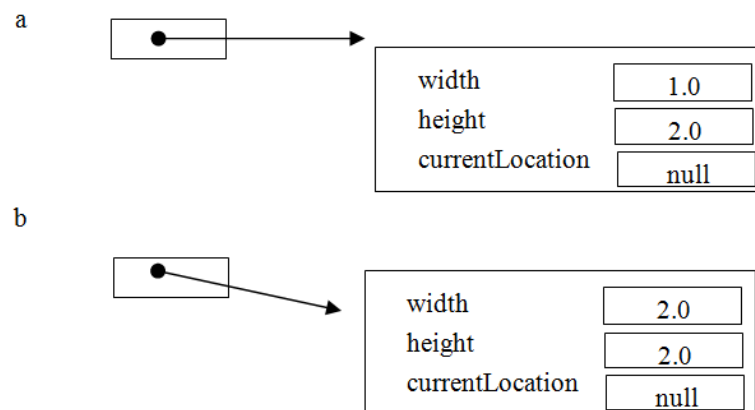
## example

- ▶ To calculate area of a rectangle or a circle using overloading methods
  - Area of a rectangle = width\*length
  - **`double area(double width, double length){...}`**
  - Area of a circle = radius\*radius\* $\pi$
  - **`double area(double radius){...}`**

54

### 3.8 Class Variables and Instance Variables

- ▶ The states of object a and b



55

### class variables and instance variables

- ▶ To provide public resources visited by objects
  - **static** variables and methods
- ▶ A **static** variable or method is also called a **class variable or method**, since it belongs to the class itself rather than to an instance of that class.
- ▶ The non-static member variables are referred to as **instance variables** since the values belong to a unique instance of the class. The instance variable has the same life cycle as the object it belongs to.

56

## Example

```

▶ class Car {
▶ public Car() {
▶ this.width = 3;
▶ this.height = 4;
▶ }
▶ public void notify(String note) {
▶ Car.note = note;
▶ }
▶ public String response() {
▶ return Car.note;
▶ }
▶ public static String note; // shared by the all Car's instances.
▶ double width, height;
▶ }

```

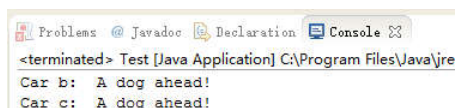
57

## Example

```

▶ public class StaticVariableDemo {
▶ public static void main(String args[]) {
▶ Car a = new Car();
▶ Car b = new Car();
▶ Car c = new Car();
▶ a.notify("A dog ahead!");
▶ System.out.println("Car b: " +
▶ b.response());
▶ System.out.println("Car c: " +
▶ c.response());
▶ }
▶ }

```



```

Problems @ Javadoc Declaration Console
<terminated> Test [Java Application] C:\Program Files\Java\jre
Car b: A dog ahead!
Car c: A dog ahead!

```

58

## access static variables

- ▶ We can access the static variables directly using the name of the class, as in:
- ▶ `Car.note = "Dog!";`
- ▶ The general form is:
- ▶ `<class_name>.<static_variable>`

59

## class variables and instance variables

- ▶ There are three ways to initialize instance variables:
  - Declaration
  - Creation
  - and Invoking a method

60

## class variables and instance variables

- ▶ 1. Declaration
  - ▶ `public class A {`
  - ▶ `public void aMethod() {`
  - ▶ `}`
  - ▶ `private int aMember = 1;`
  - ▶ `}`
- ▶ 2. Constructor
  - ▶ `class A {`
  - ▶ `A(int i) {`
  - ▶ `aMember = i;`
  - ▶ `}`
  - ▶ `private int aMember;`
  - ▶ `}`

61

## class variables and instance variables

- ▶ 3. Method
  - ▶ `class A {`
  - ▶ `void f() {`
  - ▶ `aMember = 0;`
  - ▶ `}`
  - ▶ `int aMember;`
  - ▶ `}`
- ▶ However, static variables **cannot** be initialized by constructors.
- ▶ Initialization can be done by declaration or a regular method.

62

## A static block

- ▶ A static block is a block of statements inside a Java class that will be executed when a class is first loaded into the JVM. A static block helps **to initialize the static data members**, just as constructors help to initialize instance members.
- ▶ Class A{
  - ▶ static {
  - ▶ //Statements here
  - ▶ }
  - ▶ }

63

## 3.9 Class Methods and Instance Methods

- ▶ The typical usage of **static methods** is to do some kind of generic calculation such as methods in Math and java.util.Random. A method is declared as "static" by the **static** modifier.
- ▶ Static methods are invoked with the class name, without the need for creating an instance of the class, as in:
  - ▶ **<class\_name>.<method\_name>(<argument\_list>)**

64



## Example

```
▶ public class Thermograph{
▶ public static int centigradeToFahrenheit(int cent){
▶ return cent * 9 / 5 + 32;
▶ }
▶ }
▶ Thermograph.centigradeToFahrenheit(36);
```

65

## 3.9 Class Methods and Instance Methods

- ▶ The static methods **can only** access static variables, while instance methods can access static and instance variables.

66

```

StaticMember.java
public class StaticMember {
 int i = 8;
 static int k = 5;
 public static void main(String[] args) {

 int j = k;
 aStaticMethod();
 int j = i; // Cannot make a static reference to the non-static field i
 anInstanceMethod(); // Cannot make a static reference to the non-static
 // method anInstanceMethod() from the type
 // StaticMember
 StaticMember sm = new StaticMember();
 sm.anInstanceMethod();
 sm.aStaticMethod();

 }

 public void anInstanceMethod() {
 i = aStaticMethod();
 }

 public static int aStaticMethod() {
 return k;
 }
}

```

67

## Class Methods and Instance Methods

- ▶ In summary,
- ▶ a **static** method is a method which **belongs to the class** and not to the object(instance).
- ▶ A static method can access **only** static data.
- ▶ It cannot access non-static data (instance variables).
- ▶ A static method can call **only** other static methods and cannot call a non-static method.
- ▶ A static method can be accessed directly **by the class name** and does not need any object reference
- ▶ A variable or method that is dependent on a specific instance of the class should be an instance variable or method.
- ▶ A variable or method that does not depend on a specific instance of the class should be a static variable or method.

68



| Modifier | Return Type | Method Name                                                                     | Description                                                                                                       |
|----------|-------------|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| static   | int         | binarySearch(Object[] a, int fromIndex, int toIndex, Object key)                | Searches a range of the specified array for the specified object using the binary search algorithm.               |
| static   | int         | binarySearch(Object[] a, Object key)                                            | Searches the specified array for the specified object using the binary search algorithm.                          |
| static   | int         | binarySearch(short[] a, int fromIndex, int toIndex, short key)                  | Searches a range of the specified array of shorts for the specified value using the binary search algorithm.      |
| static   | int         | binarySearch(short[] a, short key)                                              | Searches the specified array of shorts for the specified value using the binary search algorithm.                 |
| static   | <T> int     | binarySearch(T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c) | Searches a range of the specified array for the specified object using the binary search algorithm.               |
| static   | <T> int     | binarySearch(T[] a, T key, Comparator<? super T> c)                             | Searches the specified array for the specified object using the binary search algorithm.                          |
| static   | boolean[]   | copyOf(boolean[] original, int newLength)                                       | Copies the specified array, truncating or padding with false (if necessary) so the copy has the specified length. |
| static   | byte[]      | copyOf(byte[] original, int newLength)                                          | Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. |

69

## The Scope of Variables

- ▶ local variables,
- ▶ instance variables,
- ▶ and class variables

70

```

public class Car {
 public Car() {
 this(0, 0);
 }
 public Car(double width, double height) { //Parameters
 this.width = width;
 this.height = height;
 }
 public void setLocation(Point p){ //Parameters
 currentLocation = p;
 }
 public void setLocation(int x, int y){
 Point p = new Point(x, y); // Local Variables
 currentLocation = p;
 }
 public static int rectsCount = 0; // Class Variables
 private double width, height; // Instance variables
 private Point currentLocation;
}

```

71

## The Scope of Variables

- ▶ **Local variables** are declared in a method, constructor, or block. When a method is entered, the local variables are created; when the method exits, they disappear.
- ▶ **Parameters** are essentially local variables that are initialized from the actual parameters.
- ▶ Local variables are not visible outside the method.

72

## The Scope of Variables

- ▶ **Instance variables** are declared in a class, but outside a method.
- ▶ They are also called field or member variables.
- ▶ An instance variable is created when an object is created and destroyed when the object is destroyed.

73

## The Scope of Variables

- ▶ **Class/static variables** are declared with the **static** keyword in a class, but outside a method.
- ▶ There is only one copy per class, regardless of how many objects are created from it.
- ▶ Class variables are created when the program starts and destroyed when the program terminates.

74

## The Scope of Variables

- ▶ The variables declared in a block can only be accessed within the block.

```
class BlockDemo {
 public static void main(String[] args) {
 boolean flag = true;
 if (flag) {
 System.out.println("Condition is true.");
 }
 else {
 System.out.println("Condition is false.");
 }
 }
}
```

The diagram illustrates the scope of variables in the provided Java code. It uses curly braces to group lines of code into two distinct blocks. The first block, labeled 'Block 1', encompasses the `if (flag) {` statement and its body `System.out.println("Condition is true.");`. The second block, labeled 'Block 2', encompasses the `else {` statement and its body `System.out.println("Condition is false.");`. This visualizes that any variables declared within these blocks would only be accessible within their respective scopes.

75

## The Scope of Variables

- ▶ Local variables (including formal parameters) are only be accessed in the method, constructor, or block in which they are declared.

76

## The Scope of Variables

- ▶ Instance variables (**methods**) are available to all methods in the class.
- ▶ No other class can access **private** instance variables.
- ▶ If an instance variable is declared as **public**, it can be accessed from any class.
- ▶ By default, if there are no access modifiers (**private**, **public**, **protected**) preceding a variable declaration, the variable can be seen by any class in the same package.

|           | Same class | Same package | Descendant classes in different packages | Different packages |
|-----------|------------|--------------|------------------------------------------|--------------------|
| private   | ✓          |              |                                          |                    |
| default   | ✓          | ✓            |                                          |                    |
| protected | ✓          | ✓            | ✓                                        |                    |
| public    | ✓          | ✓            | ✓                                        | ✓                  |

77

## The Scope of Variables

- ▶ Java does not provide initial values for local variables.
- ▶ They must be assigned a value before their first use.
- ▶ In contrast, instance variables and static variables are initialized by Java automatically: zero for numbers, false for booleans, or null for object references.

78

```

▶ public class ScopeOfVariable {
▶ public static void main(String args[]) {
▶ int i;
▶ for (i = 0; i < 3; i++) {
▶ int x = 0;
▶ // x is a block variable. It is initialized each time block is entered
▶ x = x + 10;
▶ System.out.println("x in the for loop: " + x);
▶ }
▶ System.out.println("i is now: " + i);
▶ System.out.println("x in the class: " + x);
▶ }
▶ static void aMethod(){
▶ int x = 1; // x is a local variable
▶ System.out.println("x in aMethod: " + x);
▶ }
▶ static int x = 100; // static variable
▶ }

```

▶ x in the for loop: 10  
 ▶ x in the for loop: 10  
 ▶ x in the for loop: 10  
 ▶ i is now: 3  
 ▶ x in the class: 100

79

## Garbage Collection

- ▶ Garbage collection is the process of automatically collecting memory blocks that are no longer being used ("garbage"), and making them available for further use.
- ▶ Garbage consists of objects that can't be referenced by anyone.

80



## Garbage Collection

```
▶ public void process(){
▶ Car a = new Car();
▶ // Processing on the Object.
▶ a = null;
▶ }
```

81

## Reflection

```
▶ Reflection is a mechanism that allows a Java program to
 acquire information about a class at runtime.
```

82

## Reflection

```
public class Test {
 public static void main(String[] args) {
 Car a = new Car();
 Class c = a.getClass();
 System.out.println("This is : " + c);
 System.out.println("The class name : " + c.getName());
 }
}
```

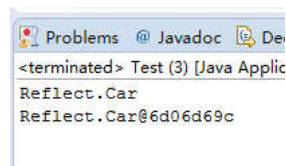
```
class Car{
 public Car(){}
 public Car(double x){}
 public void aMethod() {}
 public void aMethod(int i) {}
 public double aField;
}
```

- ▶ This is : class Car
- ▶ The class name : Car

83

## Reflection

- ▶ Car car1=new Car();
- ▶ System.out.println(car1.getClass().getName());
- ▶ Car car2 =  
(Car)Class.forName(car1.getClass().getName()).newInstance();
- ▶ System.out.println(car2);



```
<terminated> Test (3) [Java Applic
Reflect.Car
Reflect.Car@6d06d69c
```

- ▶ The classes Class, Method and Constructor are all used to find information about a class.

84

## Example

```
public static void about(Object object) {
 Class c = object.getClass();
 System.out.println(c);
 // The constructors in the class c are printed first.
 Constructor[] constructors = c.getConstructors();
 for (Constructor cs : constructors) {
 String line = "C ";
 // Return the name of the constructor as a string
 line += cs.getName() + "(";
 // Return an array of instances of Class representing
 // the parameters to the constructor
 Class[] parameterTypes = cs.getParameterTypes();
 for (int i = 0; i < parameterTypes.length; i++) {
 line += parameterTypes[i].getName();
 if (i != parameterTypes.length - 1)
 line += ", ";
 }
 line += ")";
 System.out.println(line);
 }
}
```

85

## Example

```
// The methods in the class c are listed.
Method[] methods = c.getMethods();
for (Method method : methods) {
 String line = "M ";
 Class returnType = method.getReturnType();
 line += returnType.getName() + " " + method.getName() + "(";
 Class[] parameterTypes = method.getParameterTypes();
 for (int i = 0; i < parameterTypes.length; i++) {
 line += parameterTypes[i].getName();
 if (i != parameterTypes.length - 1)
 line += ", ";
 }
 line += ")";
 System.out.println(line);
}
}
```

86

## Example

```

public static void main(String[] args){
 about(new Car());
}

class Car
C Car()
C Car(double)
M void aMethod(int)
M void aMethod()
M void wait()
M void wait(long, int)
M void wait(long)
M boolean
equals(java.lang.Object)
M java.lang.String toString()
M int hashCode()
M java.lang.Class getClass()
M void notify()
M void notifyAll()

class Car{
public Car(){}
public Car(double x){}
 public void aMethod() {}
 public void aMethod(int i) {}
}

```

87

## Example

```

public static void main(String[] args) throws InstantiationException,
IllegalAccessException, IllegalArgumentException,
InvocationTargetException, SecurityException, NoSuchMethodException{
 Car car = new Car();
 Class c = car.getClass();
 Car reflectionCar = (Car) c.getConstructors()[0].newInstance();
 reflectionCar = (Car) c.getConstructor(double.class).newInstance(1.1);
 c.getMethod("aMethod").invoke(reflectionCar);
 c.getMethod("aMethod", int.class).invoke(reflectionCar, 1);
}

Car car = new Car();
Class c = car.getClass();
for(Field f : c.getFields()){
 System.out.println("F "+f.getName()+" "+f.getType());
}

```

F aField double

## Code Organization

- ▶ A package is a group of related types (classes, interfaces ...).

org.omg.SendingContext  
org.omg.stub.java.rmi  
org.w3c.dom  
org.w3c.dom.bootstrap  
org.w3c.dom.events  
org.w3c.dom.ls  
org.w3c.dom.views  
org.xml.sax  
org.xml.sax.ext  
org.xml.sax.helpers

< >  
DatatypeConverter  
DatatypeConverterInterface  
DatatypeFactory  
Date  
DateFormat  
DateFormat.Field  
DateFormatProvider  
DateFormatSymbols  
DateFormatSymbolsProvider  
DateFormatter

PREV NEXT FRAMES NO FRAMES

### Java™ Platform, Standard Edition 8 API Specification

This document is the API specification for the Java Platform, Standard Edition 8.

See: Description

#### Profiles

- compact1
- compact2
- compact3

#### Packages

Package

early defined manner.  
that  
names.  
e sam

compact1, compact2, compact3  
java.util  
**Class Date**  
java.lang.Object  
java.util.Date

compact2, compact3  
java.sql  
**Class Date**  
java.lang.Object  
java.util.Date  
java.sql.Date

89

## Java API

- ▶ Java provides a rich set of pre-written classes, giving programmers an existing set of application programming interfaces.
- ▶ An API library of code exists to support networking, graphics, and general language routines; each major category being organized by packages.
- ▶ One of the most important packages of all is **java.lang**.

90

## Java API

- Inside `java.lang` are the classes that represent primitive data types (such as `int` & `char`), as well as more complex classes such as `String`, `Math`, `Threads`, and even the `System` class from which we obtain the input and output streams.

91

## Java API

- `java.util` contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes ( `Calendar`, `Date`, `Scanner`, `Stack`, `Vector`, `Set`, `Queue`).

92

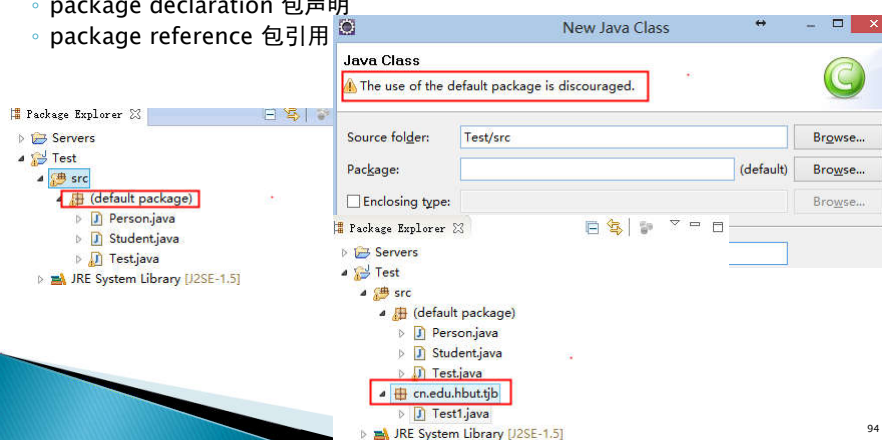
## Java API

- ▶ **java.io** provides for system input and output through data streams, serialization and the file system. For instance, File, InputStream, OutputStream, Reader, Writer.
- ▶ **java.swing** provides a set of "lightweight" GUI components such as JButton, JComboBox.

93

## Package Customs

- ▶ It is good practice to group your java code into different packages.
- ▶ There are two steps to use package:
  - package declaration 包声明
  - package reference 包引用



94

## Declaration

```
package cn.edu.hbut.java;
```

```
public class A {
 ...
}
```

All the classes and interface declared in this file belong to the package. There is only one package statement in a source file. In addition, it must be the first statement.

95

## Declaration

- ▶ The Java compiler creates corresponding folders for the declared package.
- ▶ For example, `cn.edu.hbut.java` has the counterpart `cn\edu\hbut\java` in the disk file system. All the classes and interfaces are stored in these folders.

96



## Declaration

- ▶ Package names are written in all lowercase and start with a reversed Internet domain name.
- ▶ the Internet domain name for our laboratory may be:
  - cloud.hbut.edu.cn
- ▶ Then, the packages should start with:
  - cn.edu.hbut.cloud.

97

## Reference

- ▶ There are three approaches to using package members:
  - refer to the member by its fully qualified name,
  - import the package member,
  - and import the member's entire package.

98

## Reference

- ▶ For example, suppose there is a class E in
- ▶ cn.edu.hbut.packtwo which needs to access class A in cn.edu.hbut.packone:
  - package cn.edu.hbut.packtwo;
  - class E {
  - void accessByOtherClassInOtherPackage() {
  - cn.edu.hbut.packone.A a = new cn.edu.hbut.packone.A();
  - ...
  - }

99

## Reference

- ▶ the second approach imports the member or its package and then uses its name without any prefix.
  - package cn.edu.hbut.packtwo;
  - import cn.edu.hbut.packone.A;
  - class E {
  - void accessByOtherClassInOtherPackage() {
  - A a = new A();
  - ...
  - }
- ▶

100

## Reference

- ▶ To use many types from a package, it is convenient to import the entire package.
  - `import cn.edu.hbut.packone.*;`
- ▶ means import all the classes in package `cn.edu.hbut`.

101

## Reference

- ▶ The Java compiler will import automatically three packages: `java.lang`, the `default anonymous package` and `the current package` which is the package declared in the current source file.
- ▶ The import directive tells the compiler where to look for the class definitions when it comes across a class that it cannot find in the default `java.lang` package.
- ▶ Note that the import statement must appear after the package declaration and before the class declaration.

102

```
package cn.edu.hbut.tjb.entities;

public class employee{

 private String name;
 private double salary;

 public employee(String n , double m){
 name=n; salary=m;
 }
 public String getname(){
 return name;
 }
 public double getsalary(){
 return salary;
 }
}
```

```
package cn.edu.hbut.tjb.business;
import cn.edu.hbut.entities.employee;

public class test{
 public static void main(String args[]){
 employee a = new employee("Wang" ,3000);
 System.out.println(a.getname());
 System.out.println(a.getsalary());
 }
}
```