

**"Journey of a thousand miles begins  
with a single step."**



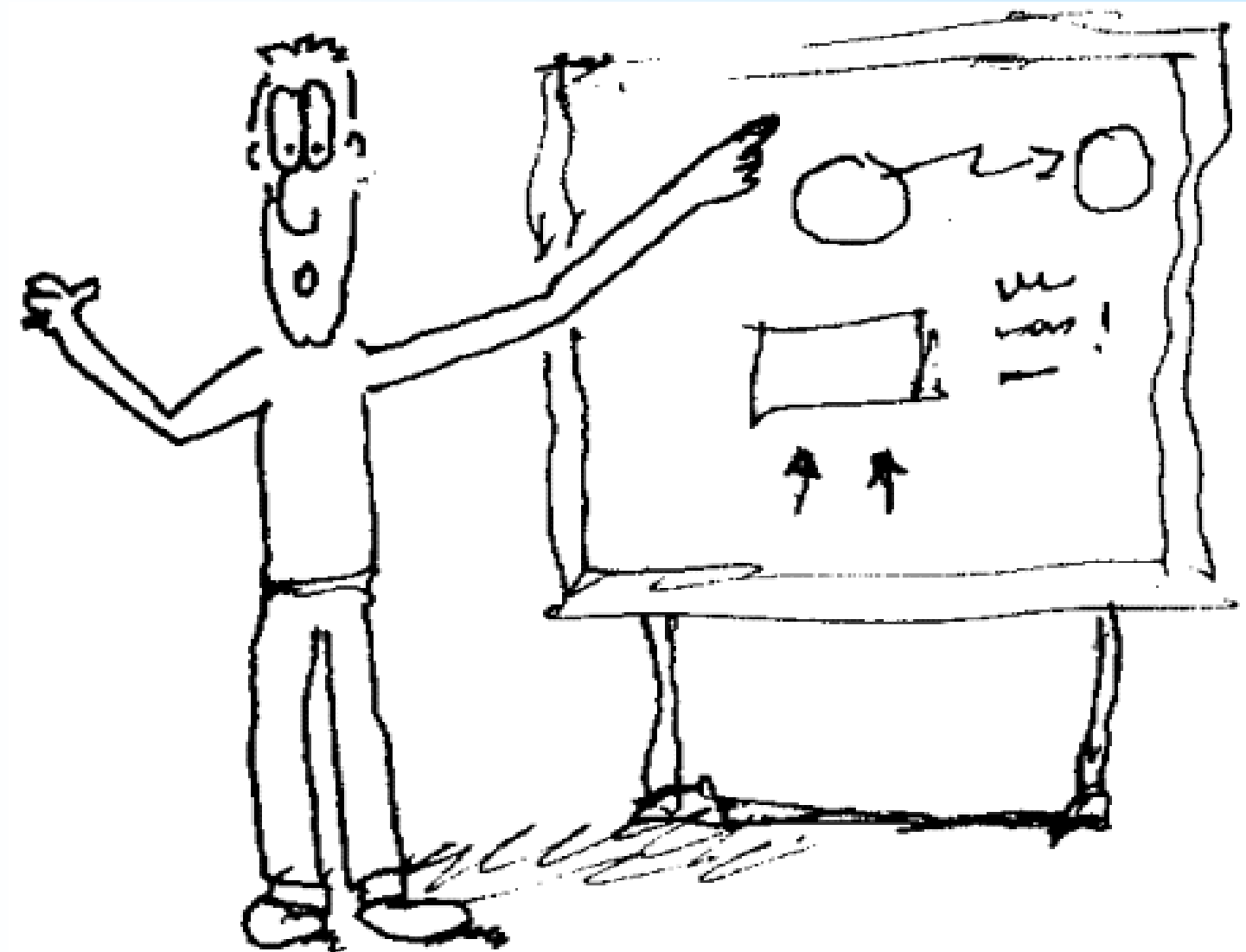
# Object Oriented Programming Java Programming language

Qiuyan Huo 霍秋艳

Software Engineering Institute

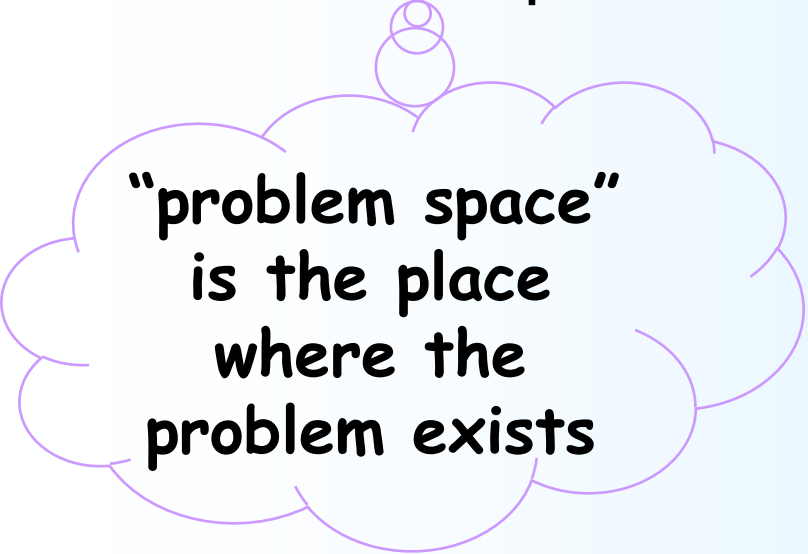
[qyhuo@mail.xidian.edu.cn](mailto:qyhuo@mail.xidian.edu.cn)

# Introduction to Objects

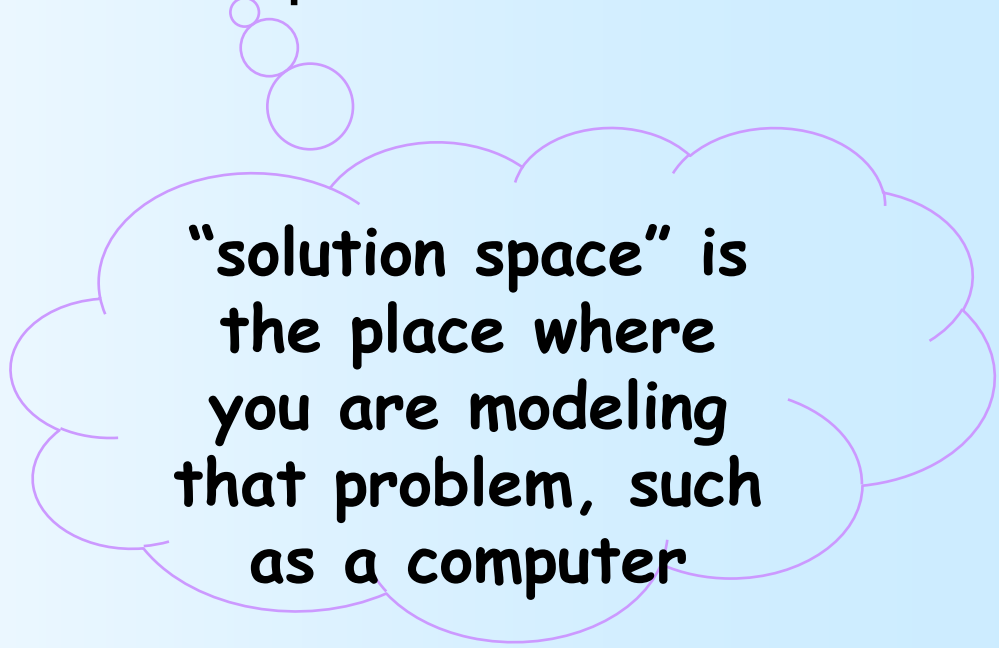


# The Progress of Abstraction

- Problem space & solution space



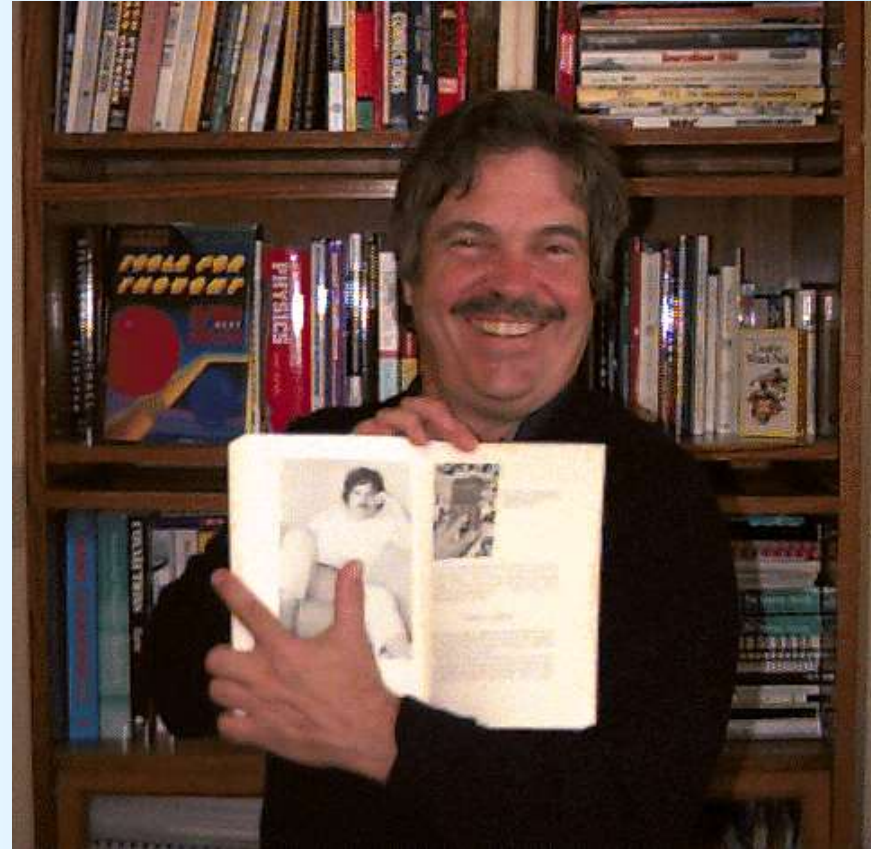
**“problem space”  
is the place  
where the  
problem exists**



**“solution space” is  
the place where  
you are modeling  
that problem, such  
as a computer**

# The Progress of Abstraction

- Alan Kay's 5 rules for a pure OOP language
  - Everything is an object
  - A program is a bunch of objects telling each other what to do by sending messages
  - Each object has its own memory made up of other objects
  - Every objects has type
  - All objects of a particular type can receive the same messages



*An object has state (状态), behavior (行为) and identity (标识) .--Booch*

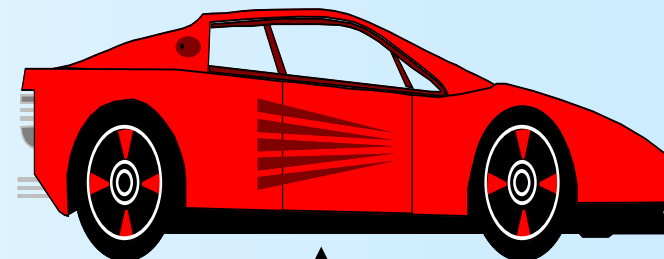
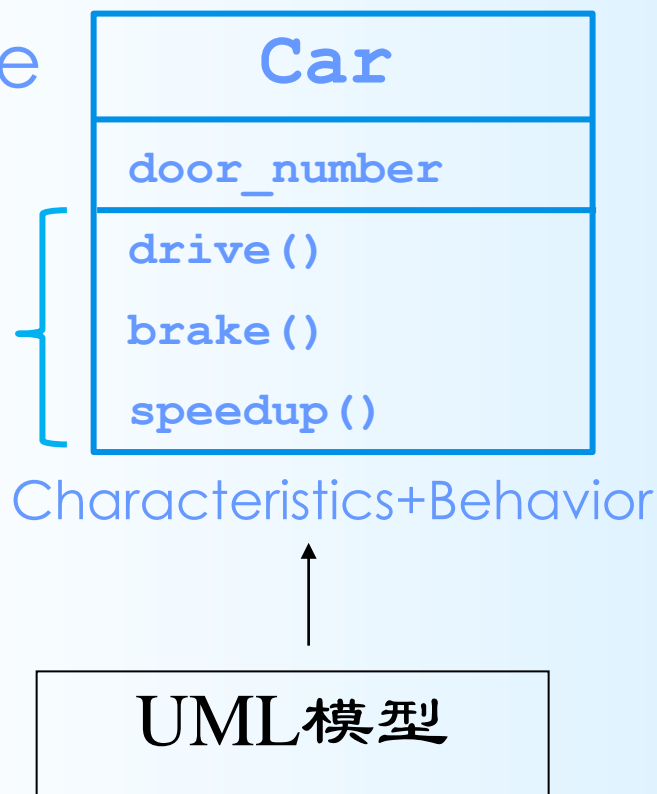
# What is an object?

- State
  - represented by object's fields (属性或字段以及它们当前的值构成的集合)
- Behavior
  - represented by object's methods (方法的集合)
- Identity
  - represented by object itself, and its ability to distinguish itself from similar objects

# An Object has an interface

Type name

interface



现实生活中的  
对象

习惯的改变 —— 把考虑解决问题的方法的方式转换为考虑将问题抽象成对象再去解决它

# Data Types

- **Everything that has a *value* also has a *type*.**
  - anything that can be stored in a variable, passed to a method, returned from a method or operated on by an operator.
- **In Java there are two kinds of types:**
  - Reference Types
    - *Classes*
    - *Arrays*
    - *Interfaces*
  - Primitive Types
    - *Built-in data types.*



# What is a Java class?

- A template for a new **data type** (一种数据类型, 即对象类型)
  - If you write a class, objects of that class's type can be constructed and used.
- A code module (模块) – Class是面向对象程序设计中最基本的程序单元。
  - A module is a unit of program code.
  - Putting code in separate classes separates your program's functionality.
- **Arrays in Java**
  - Not pointers, like C, but first-class objects
  - Checked at run-time for safety
  - Covered later

# Creating new data types: **class**

- **Class** keyword defines new data type
  - `class ATypeName { /* Class body goes here */ }`
  - `ATypeName a = new ATypeName();`

- Fields

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}
```

- Each instance of DataOnly gets its own copy of the fields

# Special case: primitive types

- Built-in: **boolean, char** (Unicode), **byte, short, int, long, float, double, void**
  - not like C/C++: **0 doesn't mean false !**
  - Size of each data type is machine independent!
  - Portability & performance implications
  - In a class, primitives get [default values](#).

- Wrappers: **Boolean, Character, Byte, Short, Integer, Long, Float, Double, Void**. Read only! [Comparative Table](#)

- subclasses of the abstract class `java.lang.Number`

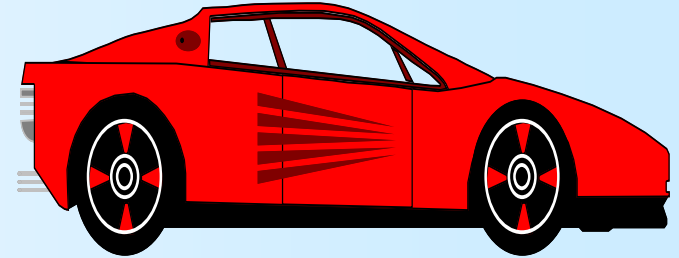
- Example: `char c = 'x';`  
`Character C = new Character(c);`



`Character C = new Character('x');`

# An Object has an interface

```
class Car {  
    int color;  
    int door_number;  
    int speed;  
  
    void drive() { ... }  
    void brake() { ... }  
    void speedUp() { ... }  
    void slowDown() { ... }  
}
```



计算机中  
的对象的原型

- Object                      `Car car = new Car();`
- Request                    `car.drive();`
- Programming with objects!

# Everything is an Object



# manipulate objects with **references**

- `String s; // Reference only`

- `// Normal object creation:`

```
String s = new String("asdf");
```

string Literals

- `// special string initialization:`

```
String s = "asdf";
```

# Literals

- Literals are fixed values found in a program.
  - we don't use the term “constant” for this, we use “constant” for something else...

- Examples:

`x = y + 3;`

`System.out.println("Hello World");`

`finished = false;`

***literals***



# Boolean Literals

- There are only two:

true

false



# Integer Literals

- **Default is decimal (base 10). Examples**

23                  100                  0                  1234567

- **Java also supports hexadecimal (base 16).**

0x23                  0x1                  0xaf3c21                  0xAF3C21

- **and octal (base 8)**

023                  01                  0123724

**any integer literal that starts with a leading 0 is treated by Java as an octal number!**

-2147483648 ( $2^{31}$ ) ~ 2147483647 ( $2^{31}-1$ )

# Long Literals

- Literal has the suffix 'l' or 'L'

**23L      1001      0xABCD01234L**

- Without the trailing 'l' any integral literal is an **integer** type.

-9223372036854775808 ( $2^{63}$ ) ~ 9223372036854775807 ( $2^{63}-1$ )

# Floating Point Literals

- Decimal only (base 10).
- Optional suffix:
  - 'f' or 'F' means **float**
  - 'd' or 'D' means **double**

**12.5**

**6.02E23D**

**1.5e-4**

# Underscores ( \_ ) in Numeric Literals

- any number of underscore characters ( \_ ) can appear anywhere between digits in a numerical literal.

**Java 7+**

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

- cannot place underscores in the following places:
  - At the beginning or end of a number
  - Adjacent to a decimal point in a floating point literal
  - Prior to an F or L suffix
  - In positions where a string of digits is expected

# Binary Literals

- the integral types (**byte**, **short**, **int**, and **long**) can also be expressed using the binary number system.
  - Java SE 7
- add the prefix **0b** or **0B** to the number.

```
// An 8-bit 'byte' value:  
byte aByte = (byte)0b00100001;  
  
// A 16-bit 'short' value:  
short aShort = (short)0b1010000101000101;  
  
// Some 32-bit 'int' values:  
int anInt1 = 0b10100001010001011010000101000101;  
int anInt2 = 0b101;  
int anInt3 = 0B101; // The B can be upper or lower case.  
  
// A 64-bit 'long' value. Note the "L" suffix:  
long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;
```

# Character Literals

- characters are represented using Unicode (*Unicode 4.0 since Java 5.0*), not ASCII.
  - we (programmers) don't have to deal with this issue as much as you might think.
  - character literals look just like they do in C/C++
 

`'a'          'b'          'C'          '\n'          '\t'`
  - anything that starts with `\u` is treated as hex encoding of the Unicode number (U+0000~U+FFFF)
 

`'\u03df'                      '\u003f'`
  - code point (32-bit **int** 0x10000~0x10ffff)
 

$\pi$  (3.1415926)  $\rightarrow$  0x3c0

# Other Literals

- array literals (used only for initialization):

```
int a[] = {1,3,5,7,9};
```

# Variables

- **Storage location (chunk of memory) and an associated type.**
  - type is either a primitive type or a reference type.
- **For primitive type variables, a variable holds the actual value (the memory is used to store the value).**
- **For reference type variables, a variable holds a reference to an object.**



# Variable Names

- Variable names are *identifiers* (so are class names, interface names, method names, ...)
- Identifiers are made of *any length* *unlimited-length sequence* of Java letters, Java digits and the underscore character ''.
  - The character '\$' is also supported, but is generally only used by code generators, not by humans.
- The first character of an identifier must be a Java letter.
- Not:
  - Java keywords or BooleanLiteral or NullLiteral

# Java Keywords

abstract	continue	for	new	switch
assert <sup>***</sup>	default	goto <sup>*</sup>	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum <sup>****</sup>	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp <sup>**</sup>	volatile
const <sup>*</sup>	float	native	super	while

\* not used

\*\* added in 1.2

\*\*\* added in 1.4

\*\*\*\* added in 5.0

*Boolean Literal:*

**true false**

*Null Literal:*

**null**

# Examples of identifiers

**Consider the following examples:**

- **\$Variable**
- **\_Variable**
- **nVariable**
- **200Variable**
- **Vari#able**
- **new**
- **Integer**

# Kinds of variables

- There are lots of kinds of variables, we will bump into the various kinds as we go...

**class variables (static)**

**instance variables**

**parameters (method, constructor, exception handler)**

**local variables**

**array components**

# Scope of Variables

- ***member variable***(成员变量): declared within a class but outside of any method or constructor; Its scope is within the entire class.
- ***local variable***: declared within a block of code; Its scope extends from its declaration to the end of the code block in which it was declared.
- ***method parameter***: its scope is the entire method or constructor for which it is a parameter

# Scoping

```

{
    ← beginning of scope 1
    int x = 12;
    // Only x available
    {
        ← beginning of scope 2
        int q = x + 96;
        // Both x & q available
    }
    ← end of scope 2
    // Only x available
    // q “out of scope”
}
    ← end of scope 1
  
```

operators

```

{
    int x = 12;
    {
        int x = 96; // Illegal
    }
}
  
```

# Operators

- Arithmetic/logical:

- `+, -, *, /, %, ++, --, &, |, ^, ~, <<, >>, >>>`

- Assignment:

- `=, +=, -=, *=, /=, % =, <<= ...`

- boolean/relational (**comparison**)

- `==, !=, <, >, <=, >=, &&, ||, ? :`

- String: `+`

- Common pitfalls

`while (x = y) {...}`

- Casting operators (**...**)

- *narrowing /widening*

- Java has no “sizeof”

Precedence

- `->` lambda expression (covered later)

# Operator Precedence

Operators	Precedence
postfix	<code>expr++ expr--</code>
unary	<code>++expr --expr +expr -expr ~ !</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>
shift	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>
relational	<code>&lt; &gt; &lt;= &gt;= instanceof</code>
equality	<code>== !=</code>
bitwise AND	<code>&amp;</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&amp;&amp;</code>
logical OR	<code>  </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</code>



# Precedence revisited

- “Ulcer Addicts Really Like C A lot.”

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++--
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&&    &   ^
C	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

non-bit operations it works

# You Never Destroy Objects

- Scope of objects

```
{ // ← Beginning of scope  
    String s = new String("a string");  
} //← End of scope
```

Reference has gone “out of scope”  
but the object itself still exists

# Methods, arguments, and return values

- Methods: how you get things done in an object

- Can only be defined inside classes

```
returnType methodName( /* Argument list */ ) {  
    /* Method body */  
}
```

- call a method

```
objectName.methodName( arg1, arg2, arg3 );
```

- Example method call:

```
int x = a.f() ; // For object a
```

# The argument list

- Types of the objects to pass into the method
- Name (identifier) to use for each one
- Whenever you seem to be passing objects in Java, you're actually passing references

```
int storage(String s) {  
    return s.length() * 2;  
}
```

```
boolean flag() { return true; }  
float naturalLogBase() { return 2.718f; }  
void nothing() { return; }  
void nothing2() {}
```

# Building a Java Program

- Before you can build your first Java program you need to understand:
  - Name visibility
    - Preventing name clashed: methods and fields are already nested within classes
    - What about class names?
    - Produce an unambiguous name for each library using your reversed domain name and library path
  - Using other components
    - `import java.util.ArrayList;`
    - `import java.util.*;`
  - The **static** keyword
    - static Data
    - static Method

## Unique

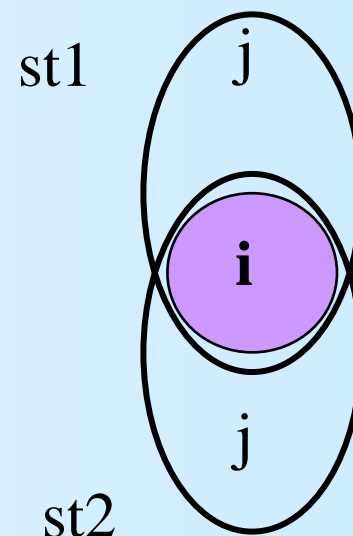
The author's domain: BruceEckel.com  
com.bruceeckel.utility.foibles

# static Data(“class data”)

- Normally each object gets its own data
- What if you want only one piece of data shared between all objects of a class?

```
class StaticTest {  
    static int i = 47;  
    int j;  
}
```

```
StaticTest st1 = new StaticTest();  
StaticTest st2 = new StaticTest();
```



# static Method

- What if you want a method that can be called for the class, without an object? (“class method”)

```
class StaticFun {  
    static void incr() {  
        WithStaticData.x++;  
    }  
}
```

```
StaticFun.incr();
```

# HelloDate.java

```
// HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```



# Comments and embedded documentation

```
/* This is a comment  
 * that continues  
 * across lines  
 */
```

```
/* This is a comment that  
continues across lines */
```

```
// This is a one-line comment
```

## ● Syntax

```
/** A class comment */  
public class DocTest {  
    /** A variable comment */  
    public int i;  
    /** A method comment */  
    public void f() {}  
}
```

## ● Embedded HTML

## ● Example tags

- @see
- {@link  
package.class#member label}
- @author
- @version
- @param
- @return

# Embedded HTML

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

```
/**
 * You can <em>even</em> insert a list:
 * <ol>
 * <li> Item one
 * <li> Item two
 * <li> Item three
 * </ol>
 */
```

[return](#)

# 注释语法 (Annotations)

- `@Override`: 限定重写父类
- `@Deprecated`: 标记已过时
- `@SuppressWarnings`: 抑制编译器警告
- .....

# Execution control

- true and false
- if - else
- return
- Iteration(循环)
- break and continue
- switch
- **NOT** support goto

`for (variable:collection)  
statement`

target (a label in the code)

Syntax is very similar (in most cases identical) to C/C++

# break examples

```
for (int i=0;i<10;i++) {  
    System.out.println("i is " + i);  
    if (i==3) break;  
}
```

```
outer: for (int j=0;j<10;j++) {           labeled break  
    for (int k=0;k<10;k++) {  
        if (k==3) break outer;  
        System.out.println("j,k: " + j + ", " + k);  
    }  
}  
// this statement is executed immediately after the labeled break
```

# continue examples

```
for (int i=0;i<10;i++) {  
    if (i==3) continue;  
    System.out.println("i is " + i);  
}
```

```
outer: for (int j=0;j<10;j++) { labeled continue  
    for (int k=0;k<10;k++) {  
        if (k==3) continue outer;  
        System.out.println("j,k: " + j + "," + k);  
    }  
}
```

# switch

```
switch(selector) {  
    case value1 : statement; break;  
    case value2 : statement; break;  
    case value3 : statement; break;  
    case value4 : statement; break;  
    case value5 : statement; break;  
    // ...  
    default: statement;  
}
```

integral value (整数值)

byte, short, char, int;

Enum ;

Character, Byte, Short, and Integer

**Java 7: String**

# ? switch example

```
int i = 1;  
int j = 2; → final int j = 2;  
switch ( i ) {  
  
    case 1: System.out.println(i);  
            System.out.println(++i);  
  
    case j: System.out.println("j");  
  
    default: System.out.println(++i);  
}
```

OK!



# OOP initialization & Cleanup

# Comparative Table (对照表)

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	$-2^{15}$	$+2^{15}-1$	Short
int	32-bit	$-2^{31}$	$+2^{31}-1$	Integer
long	64-bit	$-2^{63}$	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	—	—	—	Void

# Default values

Primitive type	Default
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d