

JAVA语言程序设计 复习提纲



1、考试时间：12月25日（周三）上午

2、试卷类型：全英文试卷闭卷

3、复习范围：

Chapter 1, 2, 3, 4, 5, 6, 7, 9, 10,
11, 12, 13, 15, 17, 19

注意：不考Chapter 14 JavaFX Basics
Lab1~Lab3

考试题型：

(1) 单选题 15道 30分

(2) 判断题 (T或F) 10道 10分

(3) 简答题3道 15分

(4) 读程序写结果题3道15分

(异常、继承和多态、抽象类、接口.....)

(5) 编程题2道大题30分

(读UML图、类、接口、异常、ArrayList、

Text I/O (包含command-line arguments) 、

BinaryI/O、arraycopy.....)

Chapter 1 Introduction to Computers, Programs, and Java



JVM、JRE and JDK

Java语言规范是对语言的技术定义，包括Java编程语言的语法和语义

☞ JVM

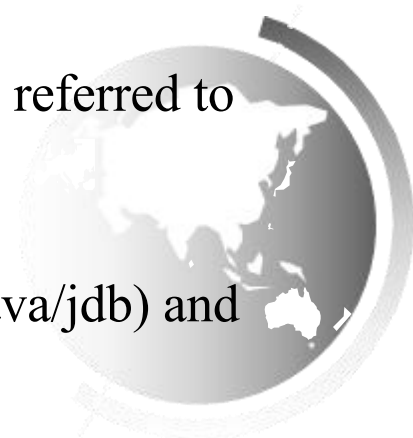
- ☞ **Java Virtual Machine** (JVM) is required on every platform where your programming will run. It is responsible for interpreting Java technology code, loading Java classes, and executing Java technology programs.

☞ JRE

- ☞ A Java technology program also needs a set of standard Java class libraries for the platform. Java class libraries are libraries of prewritten code that can be combined with the code that you write to create robust applications.
- ☞ Combined, the JVM software and Java class libraries are referred to as the **Java runtime environment** (JRE).

☞ JDK

- ☞ **Java Development Kit**, including JRE/ Java tool(javac/java/jdb) and Java API .



- Java SE : J2SE, Java 2 Platform Standard Edition
- Java EE : J2EE, Java 2 Platform, Enterprise Edition
- Java ME : J2ME, Java 2 Platform Micro Edition
- JDK (Java Development Kit)

Anatomy of a Java Program

- ➡ Class name （类名）
- ➡ Main method （主方法）
- ➡ Statements （语句）
 - ➡ Statement terminator （语句结束符）
- ➡ Reserved words （保留字）
- ➡ Comments （注释）
- ➡ Blocks （块）



Chapter 2 Elementary Programming



Identifiers

- An identifier is a sequence of characters that consist of letters, digits, underscores (`_`), and dollar signs (`$`).
- An identifier must start with a letter, an underscore (`_`), or a dollar sign (`$`). It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A, “Java Keywords,” for a list of reserved words).
- An identifier cannot be `true`, `false`, or `null`.
- An identifier can be of any length.

Reading Input from the Console

1. Create a Scanner object

```
Scanner input = new Scanner (System.in) ;
```

声明input是一个
Scanner类型的变量

创建一个Scanner类型
的对象

右边赋值给左边的变
量input

Import java.util.Scanner



Reading Input from the Console

2. Use the method `nextDouble()` to obtain a double value.
For example,

```
System.out.print("Enter a double value: ");  
Scanner input = new Scanner(System.in);  
double radius = input.nextDouble();
```

Method	Description
<code>nextByte()</code>	reads an integer of the <code>byte</code> type.
<code>nextShort()</code>	reads an integer of the <code>short</code> type.
<code>nextInt()</code>	reads an integer of the <code>int</code> type.
<code>nextLong()</code>	reads an integer of the <code>long</code> type.
<code>nextFloat()</code>	reads a number of the <code>float</code> type.
<code>nextDouble()</code>	reads a number of the <code>double</code> type.



Named Constants

```
final datatype CONSTANTNAME = VALUE;
```

```
final double PI = 3.14159;
```

```
final int SIZE = 3;
```



Numerical Data Types

四种类型的整数

两种类型的浮点数

Name	Range	Storage Size
<code>byte</code>	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
<code>short</code>	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
<code>int</code>	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
<code>long</code>	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
<code>float</code>	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
<code>double</code>	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

`float`: 单精度; `double`: 双精度



Literals (字面值)

A *literal* is a constant value that appears directly in the program. For example, 20_10, 34, 1,000,000, 5.0 and false are literals in the following statements:

```
int i = 34;   int i=0b1101;   int i=0x11;
```

```
int k = 20_10;
```

```
long x = 1000000;
```

```
double d = 5.0;
```

```
boolean b = false;
```



Augmented Assignment Operators

(增强赋值操作符)

操作符+、-、*、/、%可以结合赋值操作符形成增强操作符

<i>Operator</i>	<i>Name</i>	<i>Example</i>	<i>Equivalent</i>
<code>+=</code>	Addition assignment 加法赋值操作符	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>



Increment and Decrement Operators

<i>Operator</i>	<i>Name</i>	<i>Description</i>	<i>Example (assume i = 1)</i>
++var	preincrement	Increment var by 1 , and use the new var value in the statement	int j = ++i; // j is 2, i is 2
var++	postincrement	Increment var by 1 , but use the original var value in the statement	int j = i++; // j is 1, i is 2
--var	predecrement	Decrement var by 1 , and use the new var value in the statement	int j = --i; // j is 0, i is 0
var--	postdecrement	Decrement var by 1 , and use the original var value in the statement	int j = i--; // j is 1, i is 0

Increment and Decrement Operators, cont.

```
int i = 10;
```

```
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;  
i = i + 1;
```

```
int i = 10;
```

```
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;  
int newNum = 10 * i;
```

Conversion Rules

When performing a **binary operation** involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is double, the other is converted into double.
2. Otherwise, if one of the operands is float, the other is converted into float.
3. Otherwise, if one of the operands is long, the other is converted into long.
4. Otherwise, both operands are converted into int.

Type Casting

Implicit casting

```
double d = 3; (type widening)
```

Explicit casting

```
int i = (int)3.0; (type narrowing)
```

```
int i = (int)3.9; (Fraction part is truncated)
```

What is wrong? `int x = 5 / 2.0;`

range increases



byte, short, int, long, float, double

Chapter 3 Selections



The boolean Type and Operators

- One-way `if` Statements
- The Two-way `if` Statement
- Multiple Alternative `if` Statements
- Multi-Way `if-else` Statements
- Switch Statements

Logical Operators

Conditional Expressions



3. Logical Operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
	or	logical disjunction
^	exclusive or	logical exclusion

The & and | Operators

Supplement III.B, “The & and | Operators”

P1&P2

P1&&P2

P1 | P2

P1 | | P2

Relational Operators

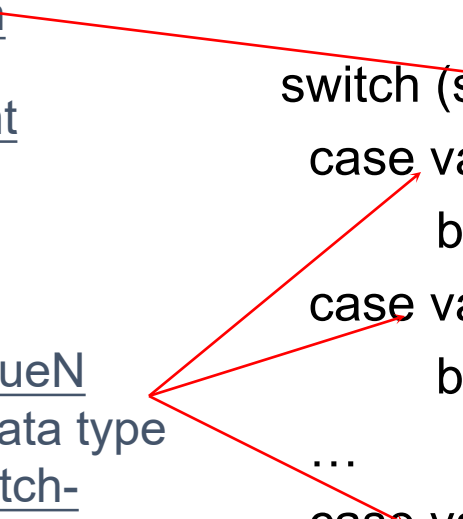
Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	less than	<code>radius < 0</code>	<code>false</code>
<=	≤	less than or equal to	<code>radius <= 0</code>	<code>false</code>
>	>	greater than	<code>radius > 0</code>	<code>true</code>
>=	≥	greater than or equal to	<code>radius >= 0</code>	<code>true</code>
==	=	equal to	<code>radius == 0</code>	<code>false</code>
!=	≠	not equal to	<code>radius != 0</code>	<code>true</code>

switch Statement Rules

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as $1 + x$.

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-  
        default;  
}
```



Chapter 4 Mathematical Functions, Characters, and Strings



Mathematical Functions

The `Math` Class

Character Data Type

The `String` Type



The random Method

☞ `random ()` $0 \leq \text{Math.random}() < 1.0$

Returns a random double value
in the range [0.0, 1.0).

Examples:

`(int) (Math.random () * 10)` → Returns a random integer
between 0 and 9 .

`50 + (int) (Math.random () * 50)` → Returns a random integer
between 50 and 99 .

In general,

`a + (int) (Math.random() * b)`

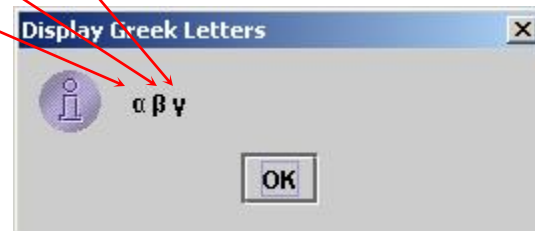
returns a random number between a and a+b-1



Unicode Format

Java characters use *Unicode*, a **16-bit** encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of **written texts** in the world's **diverse languages**. Unicode takes two bytes, preceded by `\u`, expressed in **four hexadecimal numbers** that run from `'\u0000'` to `'\uFFFF'`. So, Unicode can represent $65535 + 1$ characters.

Unicode `\u03b1` `\u03b2` `\u03b3` for three Greek letters



ASCII Code for Commonly Used Characters

!

Characters	Code Value in Decimal	Unicode Value
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

3. The String Type

The char type only represents one character. To represent a string of characters, use the data type called String. For example,

类 对象
Scanner input = new Scanner(System.in)

String message = "Welcome to Java";

String is actually a predefined class in the Java library just like the System class and Scanner class. **The String type is not a primitive type. It is known as a *reference type*.** Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 9, “Objects and Classes.” For the time being, you just need to know how to declare a String variable, how to assign a string to the variable, how to concatenate strings, and to perform simple operations for strings.

Simple Methods for String Objects

Method	Description
<code>length()</code>	Returns the number of characters in this string.
<code>charAt(index)</code>	Returns the character at the specified index from this string.
<code>concat(s1)</code>	Returns a new string that concatenates this string with string <code>s1</code> .
<code>toUpperCase()</code>	Returns a new string with all letters in uppercase.
<code>toLowerCase()</code>	Returns a new string with all letters in lowercase.
<code>trim()</code>	Returns a new string with whitespace characters trimmed on both sides.



instance method and static method

Strings are objects in Java. The methods in the preceding table can only be invoked from a specific string instance. For this reason, these methods are called *instance methods*.

A non-instance method is called a *static method*.

A static method can be invoked without using an object. All the methods defined in the **Math** class are static methods. They are not tied to a specific object instance.



实例方法调用和静态方法调用的比较：

(1) The syntax to invoke an instance method is :

Reference-Variable.methodName(arguments).

eg. String message = “Hello Java”

System.out.println(message.length())

(2) The syntax to invoke an instance method is :

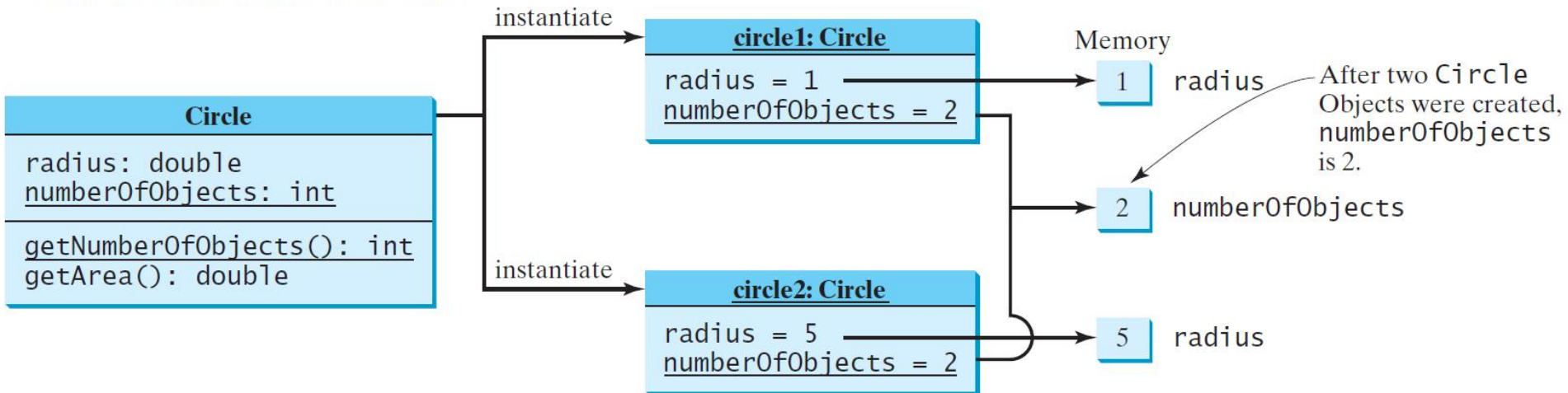
ClassName.methodName(arguments).

eg. Math.pow(2,3)



Static Variables, Constants, and Methods, cont.

UML Notation:
underline: static variables or methods



```
public class CircleWithStaticMembers {  
  double radius;  
  static int numberOfObjects = 0;  
  CircleWithStaticMembers() {  
    radius = 1.0; numberOfObjects++; }  
  CircleWithStaticMembers(double newRadius) {  
    radius = newRadius; numberOfObjects++; }  
  static int getNumberOfObjects() {  
    return numberOfObjects; }  
  double getArea() {  
    return radius * radius * Math.PI; } }  
}
```

Comparing Strings

```
if (string1== string2)
```

```
System.out.println ("string1 and string2 are the same  
object");
```

```
else
```

```
System.out.println("string1 and string2 are different objects");  
if (string1.equals(string2))
```

```
S1="ABC",S2="ABG", s1.compareTo(s2)
```

```
return -4
```

Conversion between Strings and Numbers

数值型字符串转换为数值方法：要将字符串转换为int值，使用Integer.parseInt方法；要将字符串转换为 double值，使用Double.parseDouble方法；Integer的intValue()。

```
int intValue = Integer.parseInt( “ 123 ” );
```

```
double doubleValue = Double.parseDouble( “45.67 ” );
```

将数值转换为字符串，只需要使用字符串的连接操作符

```
String s = 23 + "";
```

```
String s1 = String.valueOf(s);
```

```
String s2 = Integer.toString(15);
```



Chapter 5 Loops



while Loops

do-while Loop

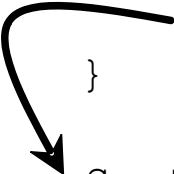
for Loop

Using break and continue



break

```
public class TestBreak {  
    public static void main (String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
  
            sum += number;  
            if (sum >= 100)  
                break;  
        }  
  
        System.out.println ("The number is " + number);  
        System.out.println ("The sum is " + sum);  
    }  
}
```



关键字break跳出整个循环



continue

```
public class TestContinue {  
    public static void main (String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
  
            if (number == 10 || number == 11)  
                continue;  
  
            sum += number;  
        }  
  
        System.out.println ("The sum is " + sum);  
    }  
}
```

Continue跳出一次迭代



Chapter 6 Methods



Defining Methods

Calling Methods

void Method

Passing Parameters

Overloading Methods

Scope of Local Variables



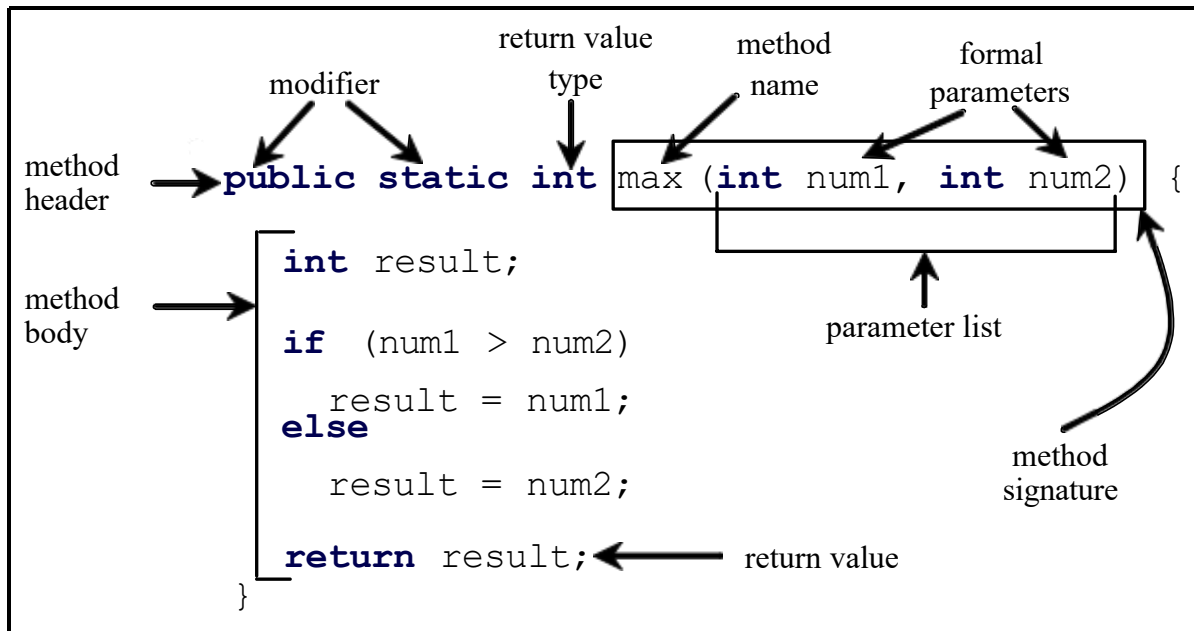
Defining Methods

方法的定义由方法名称、参数、返回值类型、修饰符以及方法体组成。定义方法的语法如下所示：

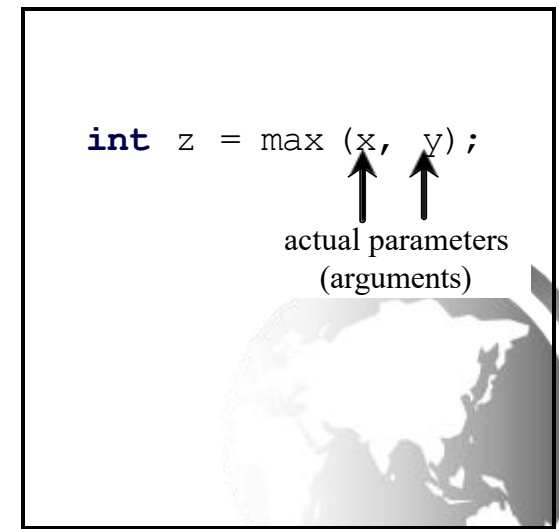
修饰符 返回值类型 方法名(参数列表)

{ // 方法体 ; }

Define a method



Invoke a method



Calling Methods, cont.

main方法由Java 虚拟机调用

TestMax.java

```
public static void main (String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
    System.out.println (  
        "The maximum between " + i + "  
        " and " + j + " is " + k);  
}
```

```
public static int max (int num1, int num2) {  
    int result;  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
    return result;  
}
```

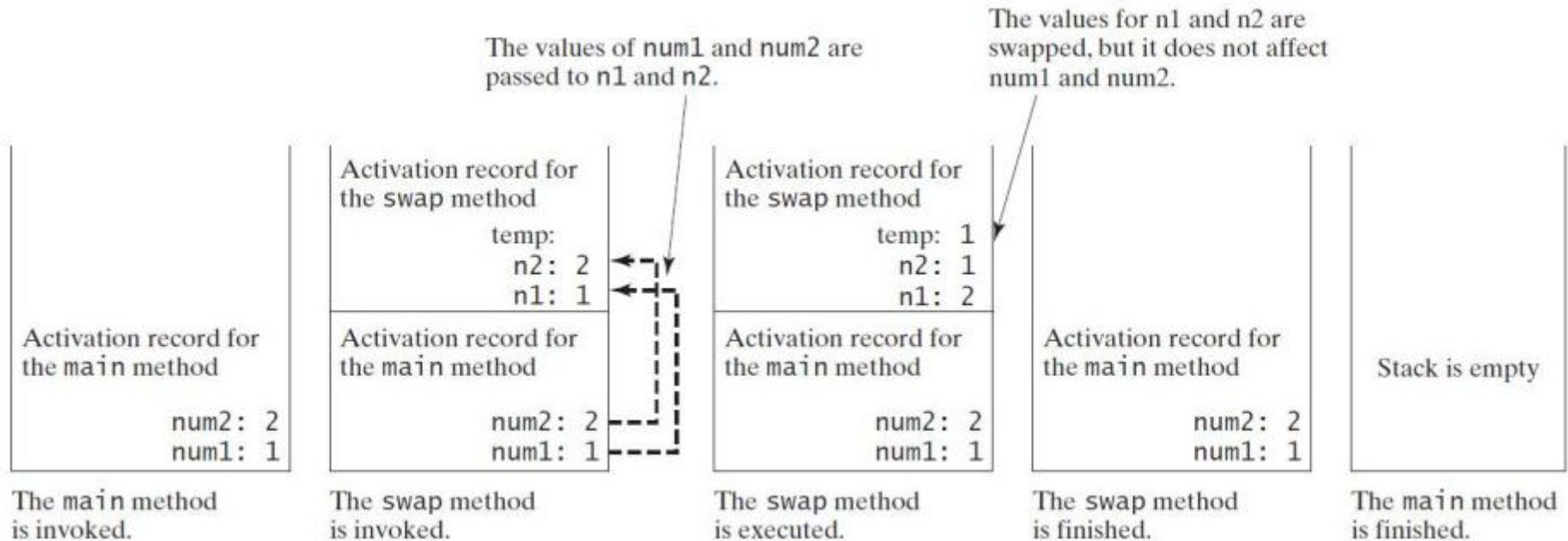
pass the value of i

pass the value of j



Pass by Value, cont.

注意：虽然实参num1和num2把值传递给形参n1和n2，但n1和n2有自己独立于num1和num2的存储空间，因此，n1和n2的改变不影响num1和num2的内容



Overloading Methods

重载方法使得可以使用同样的名字来定义不同的方法，只要它们的参数列表不同。例如：重载max方法。Java编译器根据方法签名决定使用哪个方法。

```
public static int max (int num1, int num2) {  
    if (num1 > num2)  
        return num1;  
    else  
        return num2; }
```

```
public static double max (double num1, double num2) {  
    if (num1 > num2)  
        return num1;  
    else  
        return num2; }
```



Chapter 7 Single-Dimensional Arrays



Declaring and Creating in One Step

- **`datatype[] arrayRefVar = new
datatype[arraySize];`**

```
double[] myList = new double[10];
```

- `datatype arrayRefVar[] = new
datatype[arraySize];`

```
double myList[] = new double[10];
```

Enhanced for Loop (for-each loop)

JDK 1.5 introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable. For example, the following code displays all elements in the array myList:

```
for (double value: myList)
    System.out.println(value);
```

In general, the syntax is

```
for (elementType value: arrayRefVar) {
    // Process the value
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

The Arrays.sort Method

Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
```

```
java.util.Arrays.sort(numbers); // 对整个数组进行排序
```

结果: [1.9, 2.9, 3.4, 3.5, 4.4, 6.0]

```
char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
```

```
java.util.Arrays.sort(chars,0,3); // 对从chars[0]到 chars [3-1] 的部分排序
```

结果: [4, A, a, F, D, P]



Copying Arrays

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};  
int[] targetArray = new  
    int[sourceArray.length];  
  
for (int i = 0; i < sourceArray.length; i++)  
    targetArray[i] = sourceArray[i];
```

The `arraycopy` Utility

```
arraycopy(sourceArray, src_pos,  
          targetArray, tar_pos, length);
```

Example:

```
System.arraycopy(sourceArray, 0,  
                 targetArray, 0, sourceArray.length);
```

The `arraycopy` Utility

- 数组复制的常用方法有4种
- `for`循环，效率最低
- `System.arraycopy()` 效率最高
- `Arrays.copyOf()` 效率次于第二种方法
- `Object.clone()` 效率次于第二种和第三种

Command-Line Parameters

此外，main方法还可以从命令行传送参数。

```
class TestMain {  
    public static void main (String[] args) {  
        ...  
    }  
}
```

下面的命令行用三个字符串 **arg0**、**arg1**、**arg2**
启动程序 **TestMain**

```
java TestMain arg0 arg1 arg2
```



Chapter 9 Objects and Classes



Constructors

A constructor with no parameters is referred to as a *no-arg constructor*.

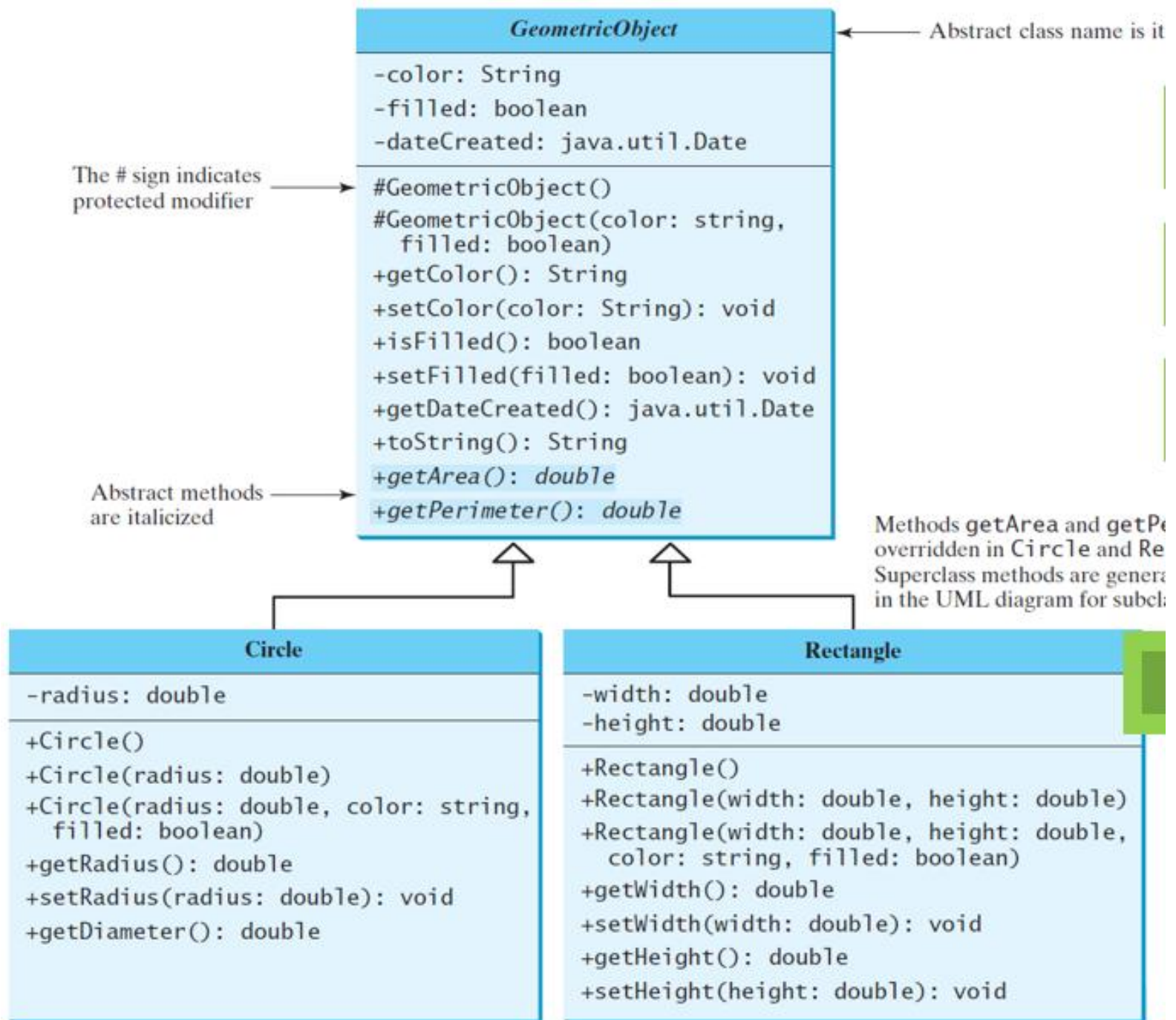
- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked **using the new operator** when an object is created. Constructors play the role of initializing objects.

Default Constructor

A class may be defined without constructors.

In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called *a default constructor*, is provided automatically ***only if no constructors are explicitly defined in the class.***

UML Class



实例方法和静态方法区别

- 实例方法可以调用实例方法和静态方法，以及访问实例数据域和静态数据域；
- 静态方法可以调用静态方法以及访问静态数据域。不能调用实例方法或者访问实例数据域



6. Array of Objects （对象数组）

```
Circle [] circleArray = new Circle [10];
```

An array of objects is actually an *array of reference variables*.

So invoking `circleArray[1].getArea()` involves two **levels of referencing** as shown in the next figure. `circleArray` references to the entire array. `circleArray[1]` references to a `Circle` object.



7. Immutable Objects and Classes

(不可变对象和类)

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* and its class is called an *immutable class*. Eg. String class.



What Class is Immutable?

For a class to be immutable, it must mark all data fields private and provide no mutator methods and **【no accessor methods that would return a reference to a mutable data field object】**.



注: 为了能够访问私有数据域, 可以提供一个 get 方法返回数据域的值。为了能够更新一个数据域, 可以提供一个 set 方法为数据域设置新值。get 方法也被称为访问器 (accessor methods), 而 set 方法称为修改器 (mutator methods)。



8. The this Keyword

- ❑ The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class's *hidden data fields*.
- ❑ Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

Calling Overloaded Constructor

```
public class Circle {  
    private double radius;
```

```
    public Circle(double radius) {  
        this.radius = radius;  
    }
```

→ this must be explicitly used to reference the data field radius of the object being constructed

```
    public Circle() {  
        this(1.0);  
    }
```

→ this is used to invoke another constructor

```
    public double getArea() {  
        return this.radius * this.radius * Math.PI;  
    }  
}
```

↓ ↓
Every instance variable belongs to an instance represented by this, which is normally omitted

Chapter 10 Thinking in Objects



3. Wrapper Classes(包装类)

- ✓ Java中的许多方法需要将对象作为参数，可使用Java API中的**包装类**将基本数据类型包装成对象。例如，将int包装成Integer类、double包装成Double类，char包装成Character类。Boolean类包装了布尔值true和false。
- ✓ Java在java.lang包中为基本数据类型提供了如下包装类：

- | | |
|------------------------------------|----------------------------------|
| <input type="checkbox"/> Boolean | <input type="checkbox"/> Integer |
| <input type="checkbox"/> Character | <input type="checkbox"/> Long |
| <input type="checkbox"/> Short | <input type="checkbox"/> Float |
| <input type="checkbox"/> Byte | <input type="checkbox"/> Double |

优势：使用包装类
可将基本数据类型
作为对象处理



The Integer and Double Classes

java.lang.Integer

-value: int

+MAX_VALUE: int

+MIN_VALUE: int

+Integer(value: int) **new Integer (3)**

+Integer(s: String) **new Integer("3")**

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Integer): int

+toString(): String

+valueOf(s: String): Integer

+valueOf(s: String, radix: int): Integer

+parseInt(s: String): int

+parseInt(s: String, radix: int): int

java.lang.Double

-value: double

+MAX_VALUE: double

+MIN_VALUE: double

+Double(value: double)

+Double(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Double): int

+toString(): String

+valueOf(s: String): Double

+valueOf(s: String, radix: int): Double

+parseDouble(s: String): double

+parseDouble(s: String, radix: int): double

The Static valueOf Methods

`valueOf(String s)` method creates a new object initialized to the value represented by the specified string.

For example:

```
Double doubleObject = Double.valueOf(" 12.4");
```

```
Integer integerObject = Integer.valueOf(" 12");
```



Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

Integer I = Integer.valueOf(2);

Boxing : Integer I = 2;

```
Integer[] intArray = {new Integer(2),  
                      new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 boxing

int i = I.intValue();

Unboxing int i = I;

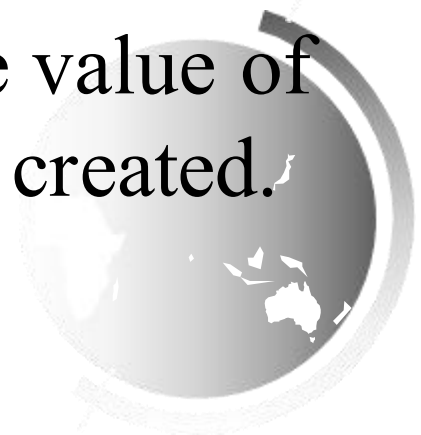
```
Integer[] intArray = {1, 2, 3};  
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

6. `StringBuilder` class and `StringBuffer` class

The `StringBuilder/StringBuffer` class is an alternative to the `String` class. In general, **a `StringBuilder/StringBuffer` can be used wherever a string is used.**

`StringBuilder/StringBuffer` is more flexible than `String`. You can **add, insert, or append new contents into a string buffer**, whereas the value of a `String` object is fixed once the string is created.



Chapter 11 Inheritance and Polymorphism



Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- ➡ To call a superclass constructor
- ➡ To call a superclass method



Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    @Override  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

10.0

10.0

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

10

20.0

Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues.

The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.

A method may be implemented in several subclasses.

The Java Virtual Machine dynamically binds the implementation of the method at runtime.



```

class TestVirtualInvoke{
    public static void main( String [] args ){
        Geometric s = new Geometric();
        s.draw();
        Geometric c1 = new Circle();
        c1.draw();
        Circle c2 = new Circle();
        c2.draw();
    }
}

class Geometric
{
    void draw(){ System.out.println("Geometric Drawing"); }
}

class Circle extends Geometric
{
    void draw(){ System.out.println("Draw Circle"); }
}

```

Geometric Drawing
Draw Circle
Draw Circle

```

class TestVirtualInvoke{
    public static void main( String [] args ){
        Geometric s = new Geometric();
        s.draw();
        Geometric c1 = new Circle();
        c1.draw();
        Circle c2 = new Circle();
        c2.draw();
    }
}

class Geometric
{
    static void draw(){ System.out.println("Geometric Drawing"); }
}

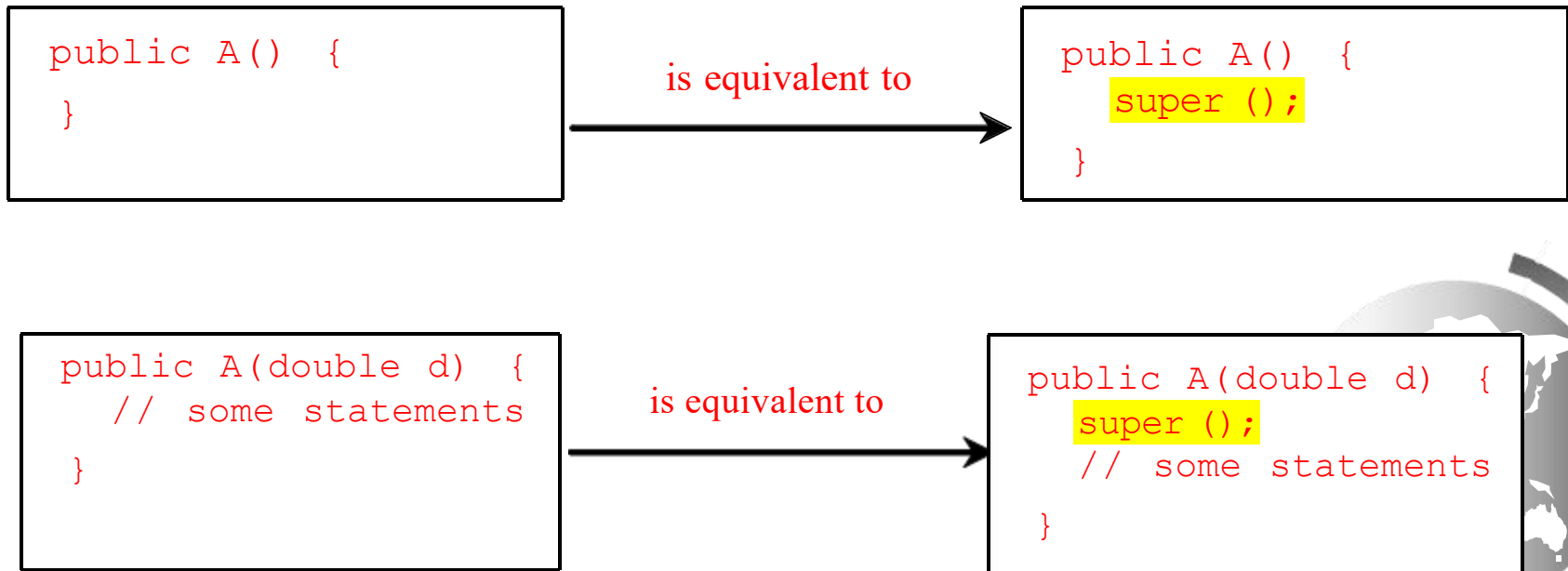
class Circle extends Geometric
{
    static void draw(){ System.out.println("Draw Circle"); }
}

```

Geometric Drawing
Geometric Drawing
Draw Circle

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,



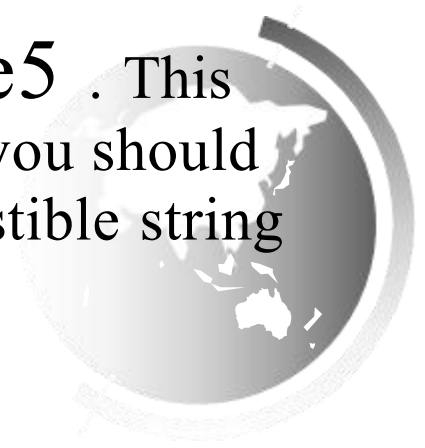
The toString() method in Object

The **toString()** method returns a string representation of the **object**. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (**@**), and a number representing this object.

```
Loan loan = new Loan();
```

```
System.out.println(loan.toString());
```

The code displays something like **Loan@15037e5** . This message is not very helpful or informative. Usually you should override the **toString** method so that it returns a digestible string representation of the object.



2) Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting can also be used to convert an object of one class type to another within an inheritance hierarchy.*

```
m(new Student());  
  
public static void m(Object x) {  
    System.out.println(x.toString());  
}
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting  
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

由于student的实例也是object的实例，所以语句合法,它称为隐式转换。



Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass.

考类一个水果、苹果、橘子之间关系的例子。Fruit类是Apple类和Orange类的父类

```
Apple x = (Apple) fruit;
```

```
Orange x = (Orange) fruit;
```



3) The instanceof Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object o = new Circle ();  
... // Some lines of code  
/** Perform casting if o is an instance of Circle */  
if (o instanceof Circle) {  
    System.out.println ("The circle diameter is " +  
        ((Circle) o).getDiameter ());  
    ...  
}
```



4. The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the **ArrayList class** that can be used to store an unlimited number of objects.

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

implements List<E>

Creates an empty list.

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element *o* from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.

Generic Type

ArrayList is known as a generic class with a generic type **E**. You can specify a concrete type to replace **E** when creating an **ArrayList**. For example, the following statement creates an **ArrayList** and assigns its reference to variable **cities**. This **ArrayList** object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

```
ArrayList<String> cities = new ArrayList<>();
```

[TestArrayList](#)

Differences and Similarities between Arrays and ArrayList

<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList<String> list = new ArrayList<>();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>

`java.util.Arrays.sort (array)`

`java.util.Collections.sort(arrayList)`

[DistinctNumbers](#)

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-



The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {  
    ...  
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

Chapter 12 Exception Handling and Text IO



Exception-Handling Overview

Throwing Exceptions

Text IO



Checked Exceptions vs. Unchecked Exceptions

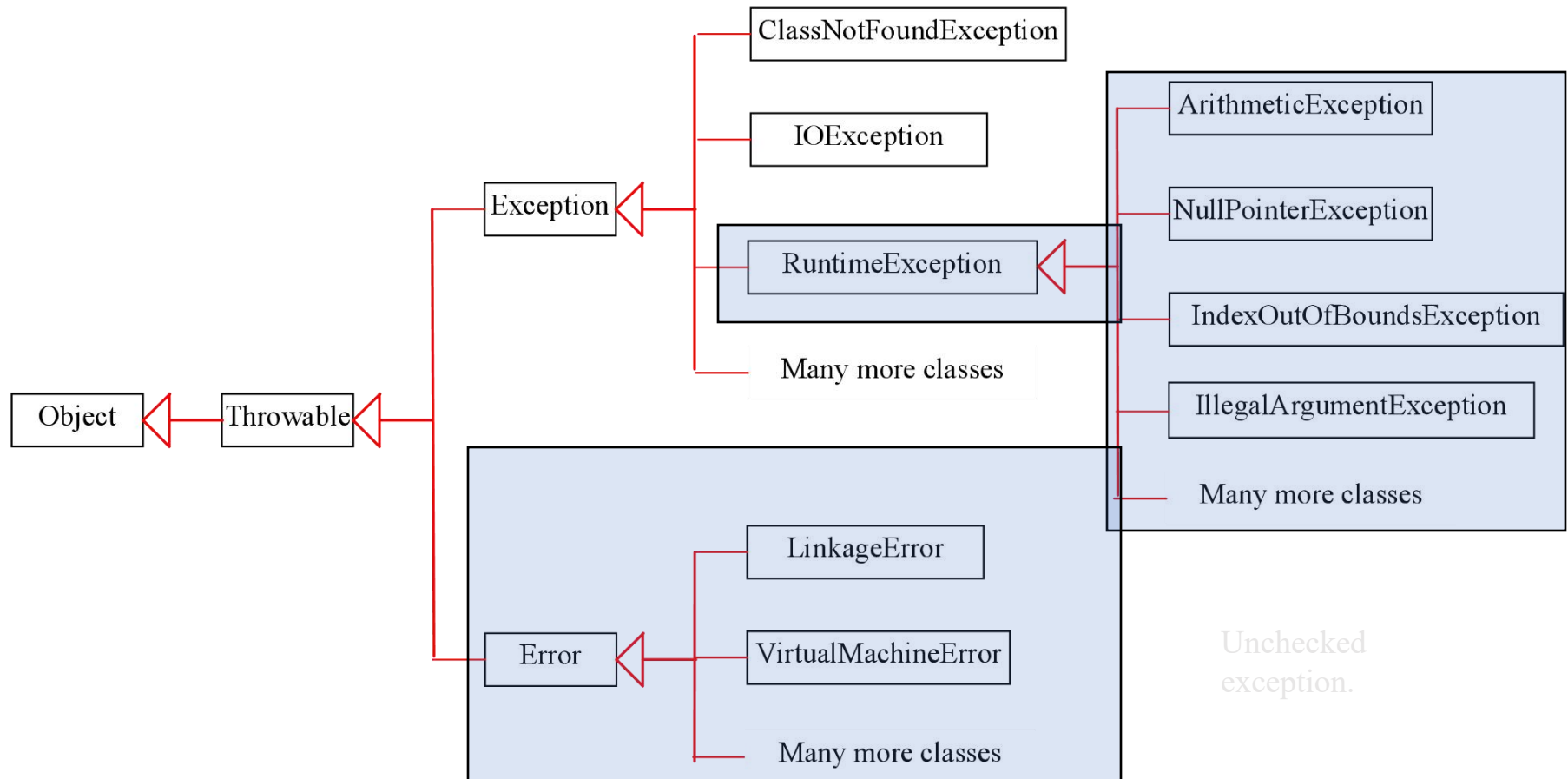
RuntimeException, Error and their subclasses are known as *unchecked exceptions* (免检异常) .

All other exceptions are known as *checked exceptions*, (必检异常) meaning that the compiler forces the programmer to check and deal with the exceptions.

***必检异常意味着编译器会强制程序员检查并通过try-catch块处理它们，或者在方法头进行声明。

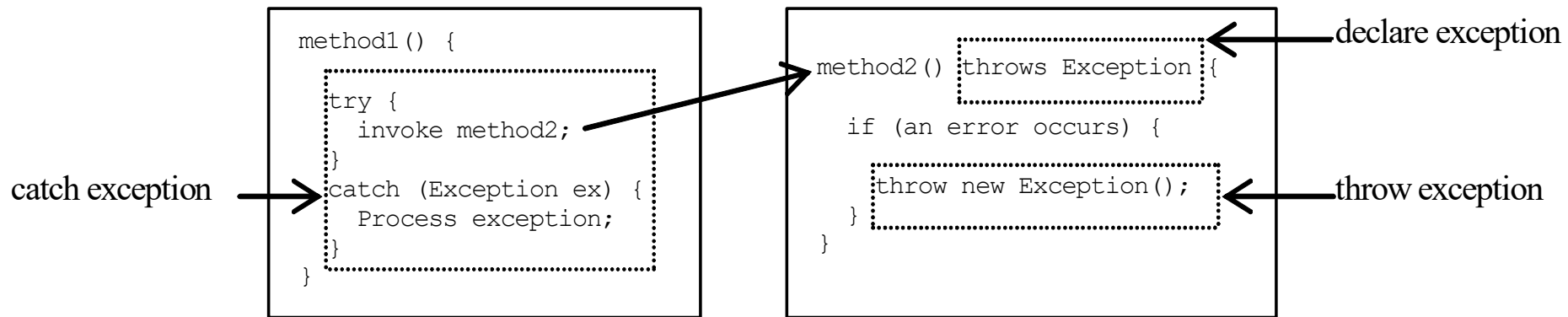


Unchecked Exceptions



Unchecked
exception.

3.Declaring, Throwing, and Catching Exceptions



```

public class CircleWithException {
    .....
    public CircleWithException(double newRadius) {
        setRadius(newRadius);
        numberOfObjects++;
    }
    public void setRadius(double newRadius) throws IllegalArgumentException {
        if (newRadius >= 0)
            radius = newRadius;
        else
            throw new IllegalArgumentException("Radius cannot be negative");
    }.....
}

```

```

public class TestCircleWithException {
    public static void main(String[] args) {
        try {
            CircleWithException c1 = new CircleWithException(5);
            CircleWithException c2 = new CircleWithException(-5);
            CircleWithException c3 = new CircleWithException(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }
        System.out.println("Number of objects created: " +
            CircleWithException.getNumberOfObjects());
    }
}

```

```

... java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

```

Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {  
    try {  
        p2();  
    }  
    catch (IOException ex) {  
        ...  
    }  
}
```

(a)

```
void p1() throws IOException {  
    p2();  
}
```

(b)

Catching Exceptions

```
try {
    statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

```
try {
    statements;
}
catch (Exception ex) {
    .....;
}
catch (RuntimeException ex) {
    .....;
}
```

RuntimeException是**Exception**的子类，
所以应该写在前面

The `finally` Clause

无论异常是否产生，`finally`子句总会被执行

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```



5. Defining Custom Exception Classes

- ➡ Use the exception classes in the API whenever possible.
- ➡ Define custom exception classes if the predefined classes are not sufficient.
- ➡ Define custom exception classes by extending Exception or a subclass of Exception.

Custom Exception Class Example

The setRadius method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler, you have to create a custom exception class.

```
public class InvalidRadiusException extends Exception {
    private double radius;

    /** Construct an exception */
    public InvalidRadiusException(double radius) {
        super("Invalid radius " + radius);
        this.radius = radius;
    }

    /** Return the radius */
    public double getRadius() {
        return radius;
    }
}
```

java.lang.Throwable

+getMessage(): String
+toString(): String

+printStackTrace(): void

+getStackTrace():
StackTraceElement[]

Returns the message of this object.

Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the `getMessage()` method.

Prints the `Throwable` object and its call stack trace information on the console.

Returns an array of stack trace elements representing the stack trace pertaining to this throwable.

Custom Exception Class Example

```
class CircleWithCustomException {  
    .....  
    public CircleWithCustomException() throws InvalidRadiusException {  
        this(1.0);  
    }  
    public CircleWithCustomException(double newRadius)  
        throws InvalidRadiusException {  
        setRadius(newRadius);  
        numberOfObjects++;  
    }  
    .....  
    /** Set a new radius */  
    public void setRadius(double newRadius)  
        throws InvalidRadiusException {  
        if (newRadius >= 0)  
            radius = newRadius;  
        else  
            throw new InvalidRadiusException(newRadius);  
    }  
    .....  
}
```

Text I/O

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.

In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.

This section introduces how to read/write strings and numeric values from/to a text file using the Scanner (读数据) and PrintWriter classes (写数据) .



Reading Data Using Scanner

Scanner input = new Scanner(System.in);

Scanner input = new Scanner(new File(filename));

java.util.Scanner

+Scanner(source: File)

Creates a Scanner object to read data from the specified file.

+Scanner(source: String)

Creates a Scanner object to read data from the specified string.

+close()

Closes this scanner.

+hasNext(): boolean

Returns true if this scanner has another token in its input.

+next(): **String**

Returns next token as a string.

+nextByte(): **byte**

Returns next token as a byte.

+nextShort(): **short**

Returns next token as a short.

+nextInt(): int

Returns next token as an int.

+nextLong(): long

Returns next token as a long.

+nextFloat(): float

Returns next token as a float.

+nextDouble(): double

Returns next token as a double.

+useDelimiter(pattern: String):

Sets this scanner's delimiting pattern.

Scanner

Writing Data Using PrintWriter

java.io.PrintWriter

+PrintWriter(filename: String)

Creates a PrintWriter for the specified file.

+print(s: String): void

Writes a string.

+print(c: char): void

Writes a character.

+print(cArray: char[]): void

Writes an array of character.

+print(i: int): void

Writes an int value.

+print(l: long): void

Writes a long value.

+print(f: float): void

Writes a float value.

+print(d: double): void

Writes a double value.

+print(b: boolean): void

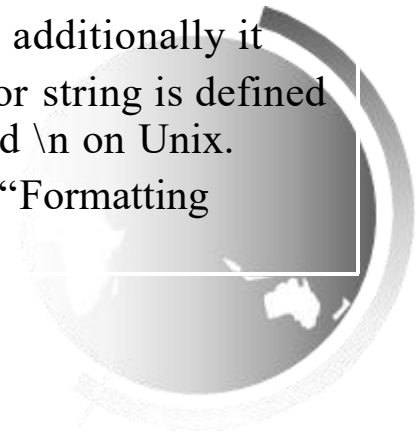
Writes a boolean value.

Also contains the overloaded
println methods.

A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.

Also contains the overloaded
printf methods.

The printf method was introduced in §3.6, “Formatting Console Output and Strings.”



```
public class WriteData {  
    public static void main(String[] args) throws Exception {  
        java.io.File file = new java.io.File("scores.txt");  
        if (file.exists()) { false  
            System.out.println("File already exists");  
            System.exit(0);  
        }  
  
        // Create a file  
        java.io.PrintWriter output = new java.io.PrintWriter(file);  
        System.out.println(file.exists()); true  
        // Write formatted output to the file  
        output.print("John T Smith ");  
        output.println(90);  
        output.print("Eric K Jones ");  
        output.println(85);  
  
        // Close the file  
        output.close();  
    }  
}
```

必须要为写入的文件创建一个PrintWriter对象，其可能会抛出I/O异常，这里强制书写。

调用PrintWriter对象上的方法 print, println 和 printf向文件写入数据。

必须要用close()方法关闭文件，否则数据不能正确保存。

scores.txt

```
John T Smith 90  
Eric K Jones 85
```

```
java.io. File file = new java.io. File("c:\\book\\scores.txt");
```


Problem: Replacing Text

Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder  
StringBuffer
```

replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.



ReplaceText

Run

```
import java.io.*;
import java.util.*;
```

```
public class ReplaceText {
    public static void main(String[] args) throws Exception {
        // Check command line parameter usage
        if (args.length != 4) {
            System.out.println(
                "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
            System.exit(1);
        }

        // Check if source file exists
        File sourceFile = new File(args[0]);
        if (!sourceFile.exists()) {
            System.out.println("Source file " + args[0] + " does not exist");
            System.exit(2);
        }

        // Check if target file exists
        File targetFile = new File(args[1]);
        if (targetFile.exists()) {
            System.out.println("Target file " + args[1] + " already exists");
            System.exit(3);
        }

        try (
            // Create input and output files
            Scanner input = new Scanner(sourceFile);
            PrintWriter output = new PrintWriter(targetFile);
        ) {
            while (input.hasNext()) {
                String s1 = input.nextLine();
                String s2 = s1.replaceAll(args[2], args[3]);
                output.println(s2);
            }
        }
    }
}
```

检查传进main方法的参数个数.

检查源文件和目标文件是否存在.

创建Scanner和PrintWriter对象用于读取和写入数据.

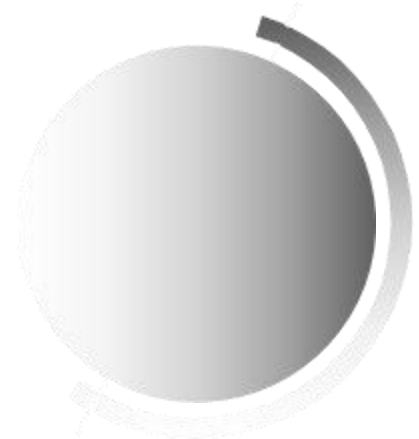
重复读入一行, 替换文本后写入目标文件.

用于表明程序异常终止.

Try-with-resources

Programmers often forget to close the file. JDK 7 provides the followings new **try-with-resources** syntax that automatically closes the files.

```
try (declare and create resources) {  
    Use the resource to process the file;  
}
```



Chapter 13 Abstract Classes and Interfaces



5. Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

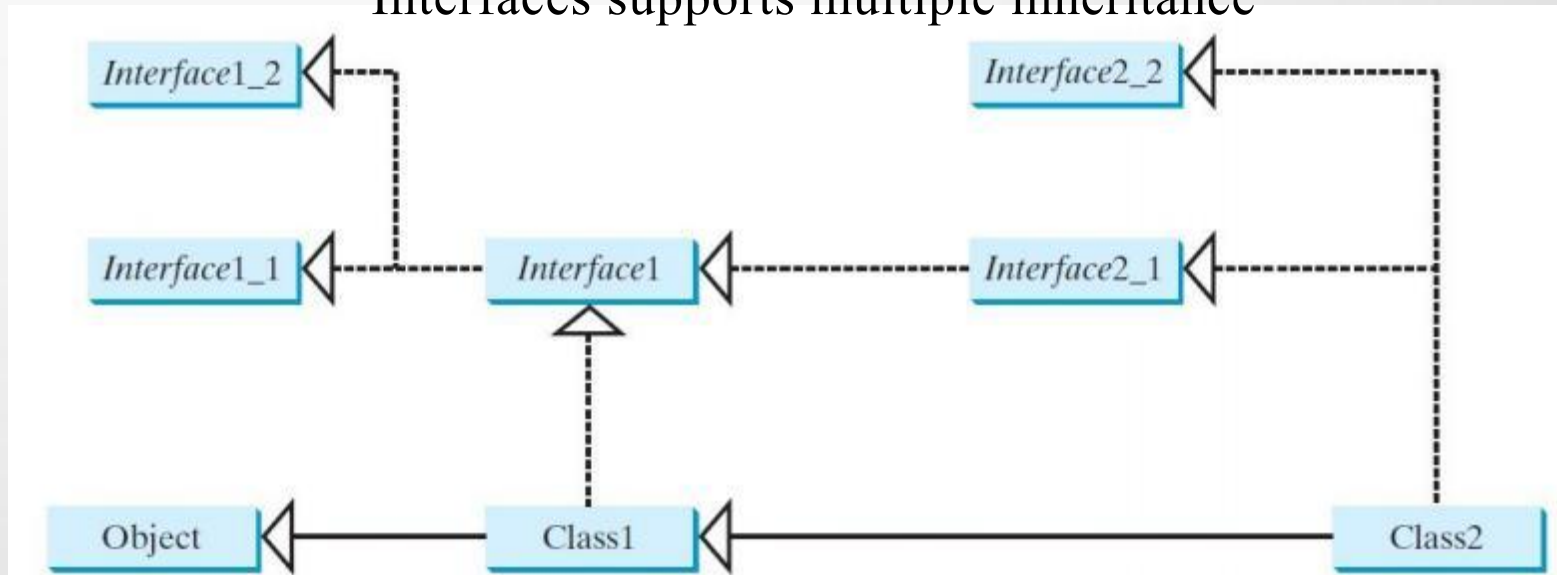
	<i>Variables</i>	<i>Constructors</i>	<i>Methods</i>
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final .	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods



Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type of an interface type to its subclass, and vice versa. and cast a variable

Interfaces supports multiple inheritance



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class?

In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person.

A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.

In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {  
    constant declarations;  
    abstract method signatures;  
}
```

- ☞ 一个接口可以继承多个接口；例如：interface a extends cls1,cls2;
- ☞ 一个类可以实现多个接口；例如：class b implements face1 , face2;
- ☞ 一个类只能继承一个类，这就是JAVA的继承特点；



3. Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
    // class body omitted

    @Override
    public int compareTo(Integer o) {
        // Implementation omitted
    }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
    // class body omitted

    @Override
    public int compareTo(BigInteger o) {
        // Implementation omitted
    }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
    // class body omitted

    @Override
    public int compareTo(String o) {
        // Implementation omitted
    }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
    // class body omitted

    @Override
    public int compareTo(Date o) {
        // Implementation omitted
    }
}
```

4. The Cloneable Interfaces

Marker Interface: An empty interface.

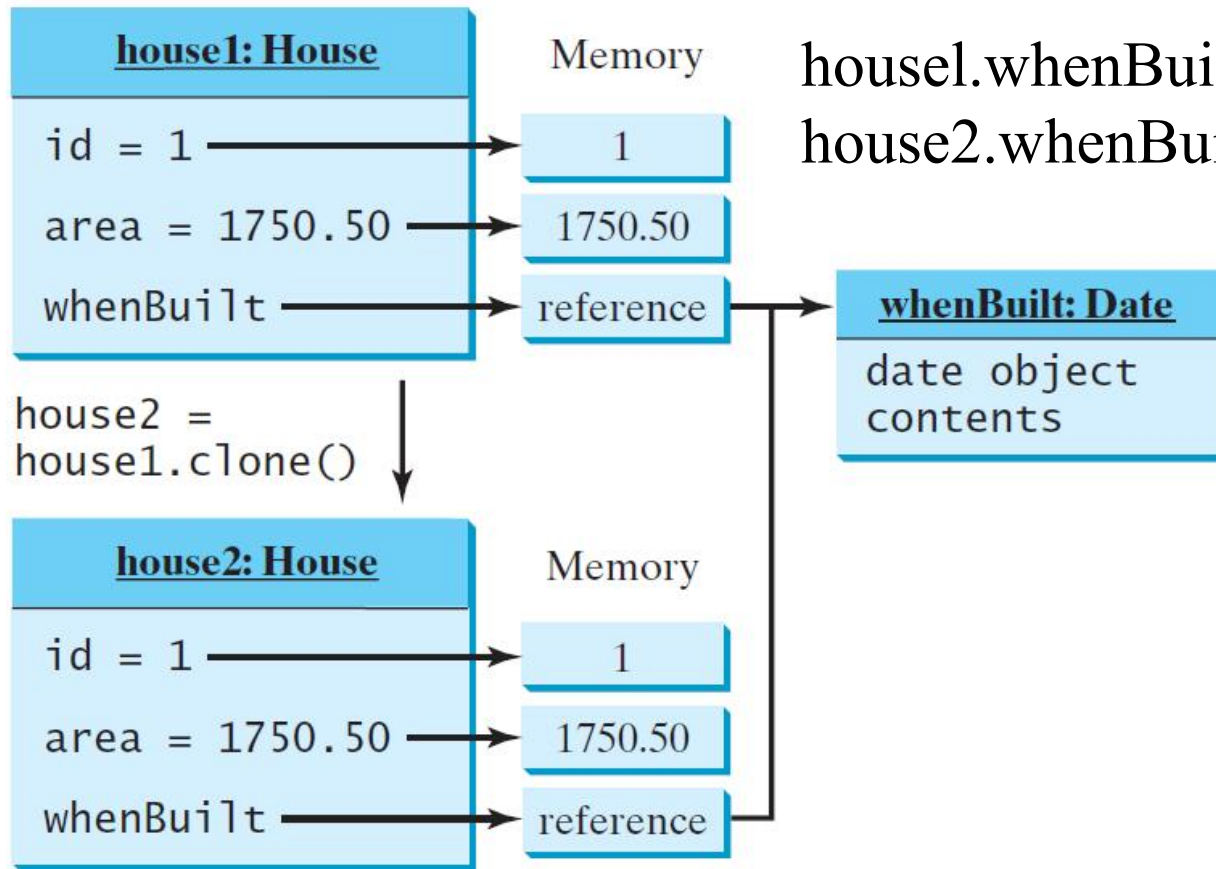
A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

```
package java.lang;  
public interface Cloneable {  
}
```

Shallow vs. Deep Copy

```
public Object clone() throws  
CloneNotSupportedException {  
    return super.clone();  
}
```

Shallow Copy



house1==house2 is false

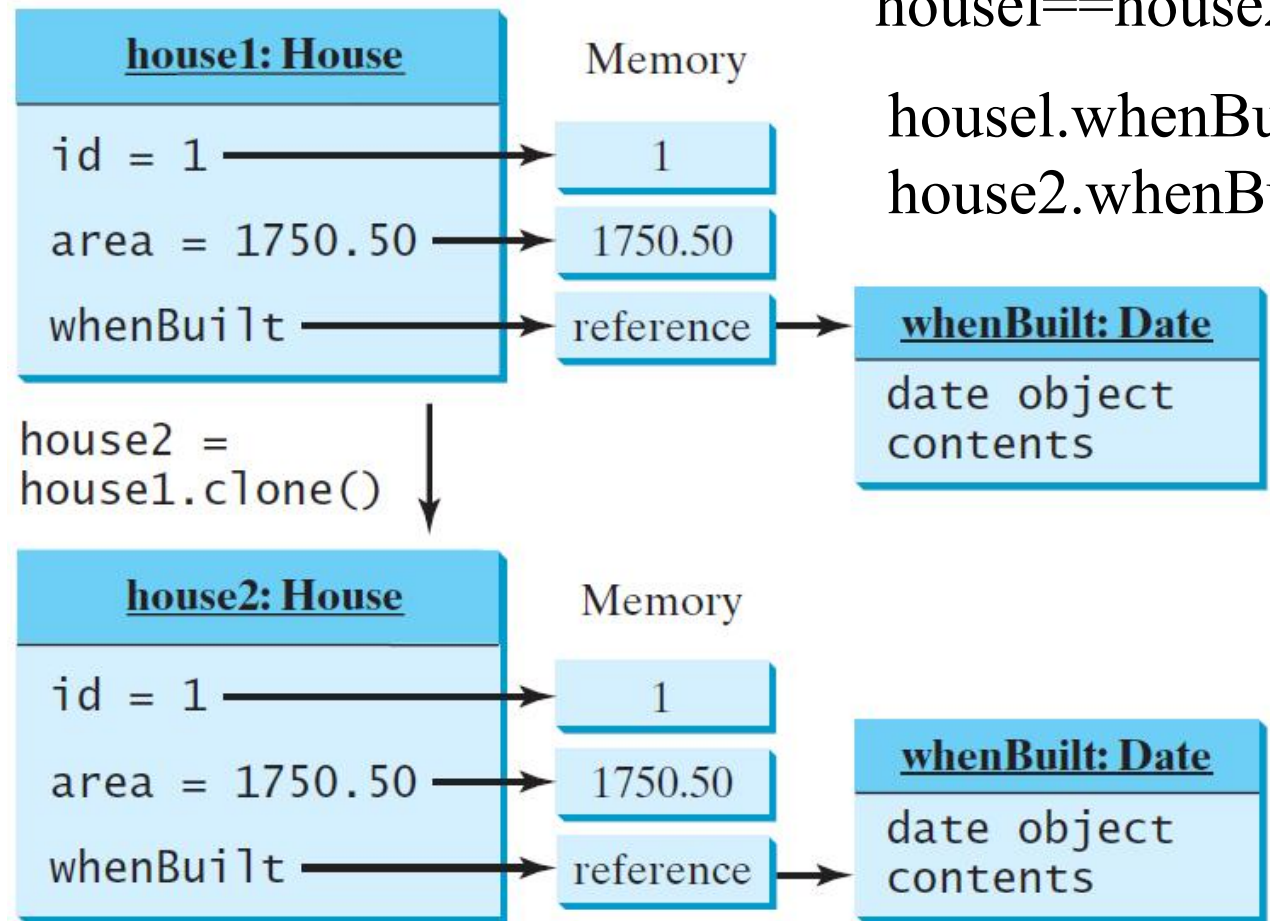
house1.whenBuilt==
house2.whenBuilt is true

```

public Object clone() throws CloneNotSupportedException {
    House houseClone = (House) super.clone();
    houseClone.whenBuilt = (java.util.Date) (whenBuilt
.clone());
    return houseClone ;
}

```

Deep Copy



house1==house2 ?

house1.whenBuilt=
house2.whenBuilt ?

Chapter 15 Event-Driven Programming and Animations



Java Event Delegation Model

⊕ Event/Event source /Event handler(Listener)

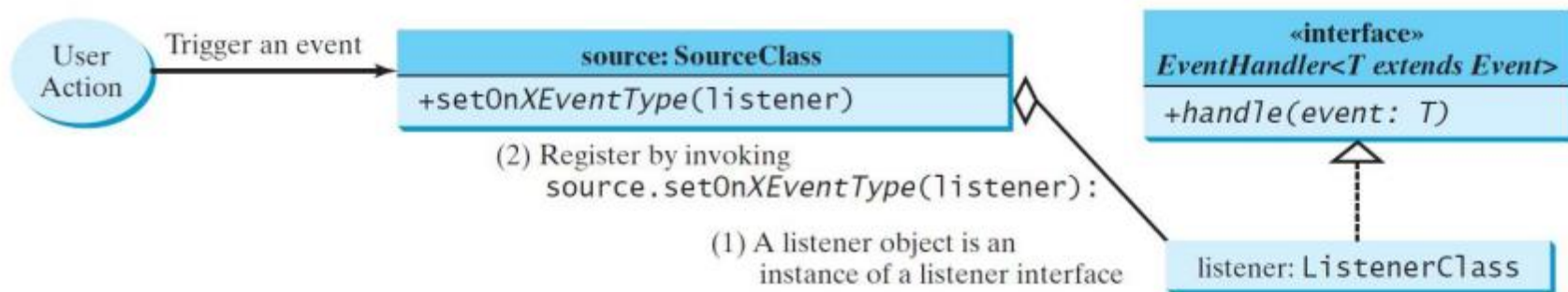
Inner Class

⊕ Anonymous Inner Class / Lambda expression



The Delegation Model

- ◆ Java采用基于委派的模型（the delegation model）进行事件的处理：源对象触发一个事件，然后一个对该事件感兴趣的对象处理它。后者称为事件处理器或事件监听器。
- ◆ 下图给出一个通用源对象和一个通用的事件T。如果一个对象要成为源对象上事件处理器，需要满足如下3个条件：
 - (1) 事件源对象，事件，事件监听器
 - (2) 一个监听器对象是对应的事件处理接口的实例。JavaFX为事件T定义了一个统一的处理器接口 `EventHandler<T extends Event>`，该处理器接口包含 `handle(T e)` 方法用于处理事件。
 - (3) 通过调用 **`source.setOnEventType(listener)`** 进行注册



Inner Classes

Inner class: A class is a member of another class.

Advantages: In some applications, you **can use an inner class to make programs simple.**

An inner class can reference the data and methods defined in the outer class in which it nests, so you **do not need to pass the reference of the outer class to the constructor of the inner class.**



Inner Classes (cont.)

Inner classes can make programs simple and concise.

An inner class supports the work of its containing outer class and is compiled into a class named

OuterClassName\$InnerClassName.class.

For example, the inner class InnerClass in OuterClass is compiled into

OuterClass\$InnerClass.class .



Anonymous Inner Classes

Inner class listeners can be shortened using anonymous inner classes. *An anonymous inner class is an inner class without a name.* It combines declaring an inner class and creating an instance of the class in one step.

An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {  
    // Implement or override methods in superclass or interface  
    // Other methods if necessary  
}
```



Anonymous Inner Classes

- ❑ An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.
- ❑ An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- ❑ An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is `Object()`.
- ❑ An anonymous inner class is compiled into a class named `OuterClassName$n.class`. For example, if the outer class `Test` has two anonymous inner classes, these two classes are compiled into `Test$1.class` and `Test$2.class`.

Simplifying Event Handling Using Lambda Expressions

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an **anonymous method with a concise syntax**. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

```
btEnlarge.setOnAction(  
    new EventHandler<ActionEvent>() {  
        @Override  
        public void handle(ActionEvent e)  
        {  
            // Code for processing event e  
        }  
    })
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {  
    // Code for processing  
    event e  
});
```

(b) Lambda expression event handler

Basic Syntax for a Lambda Expression

The basic syntax for a lambda expression is either
(type1 param1, type2 param2, ...) -> expression or
(type1 param1, type2 param2, ...) -> { statements; }

(参数) -> 结果

如 (String s) -> s.length()

如 x -> x*x

如 () -> { System.out.println("aaa"); }

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type.

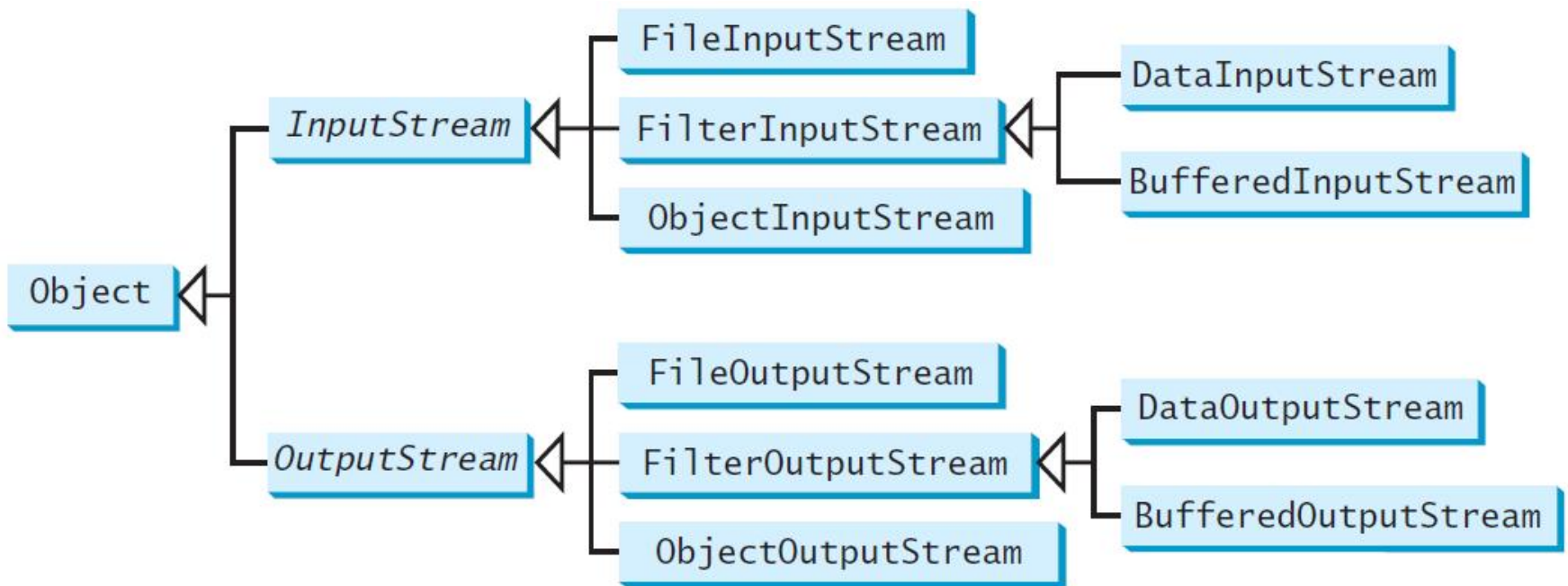
e -> { // Code for processing event e }

Single Abstract Method Interface (SAM)

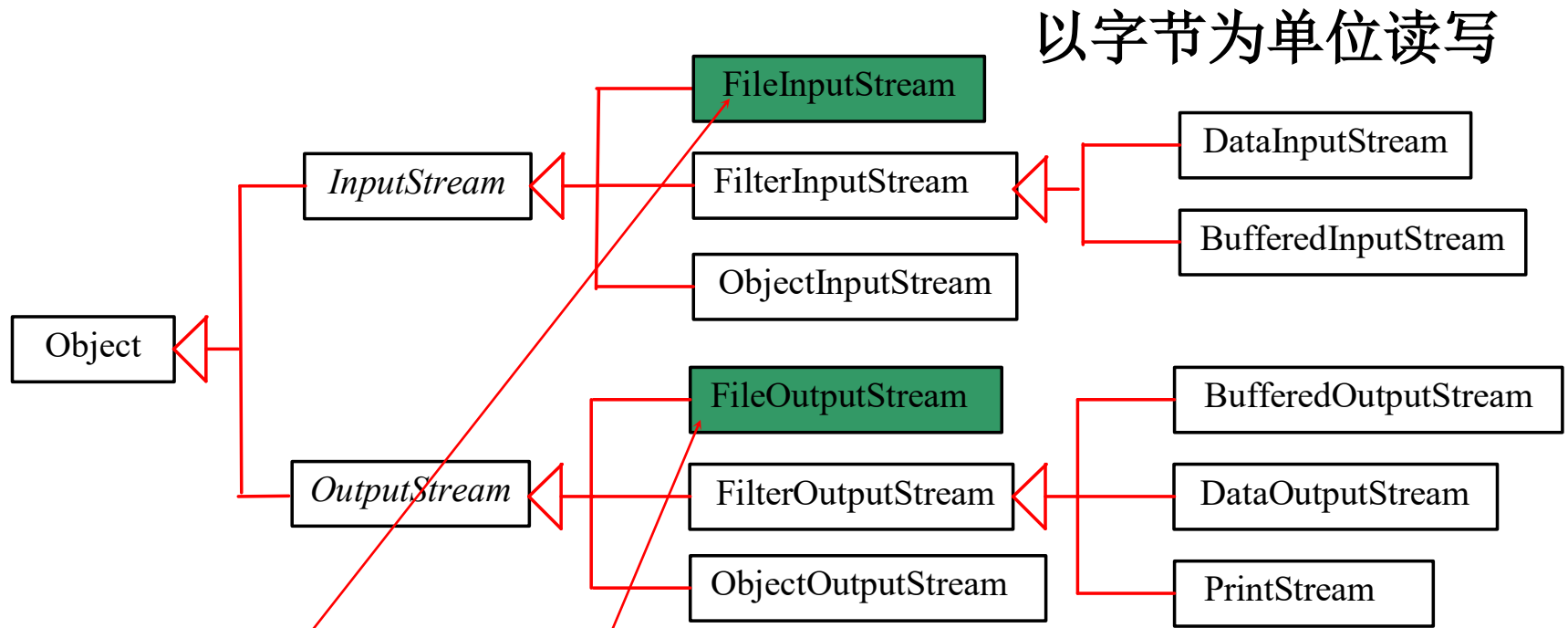
The statements in the lambda expression is all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. So, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *functional interface*, or a *Single Abstract Method (SAM)* interface.

Chapter 17 Binary I/O

2. Binary I/O Classes



FileInputStream/FileOutputStream



FileInputStream/FileOutputStream associates a binary input/output stream with an external file. All the methods in **FileInputStream/FileOuptputStream** are inherited from its superclasses.

FileInputStream

To construct a `FileInputStream`, use the following constructors:

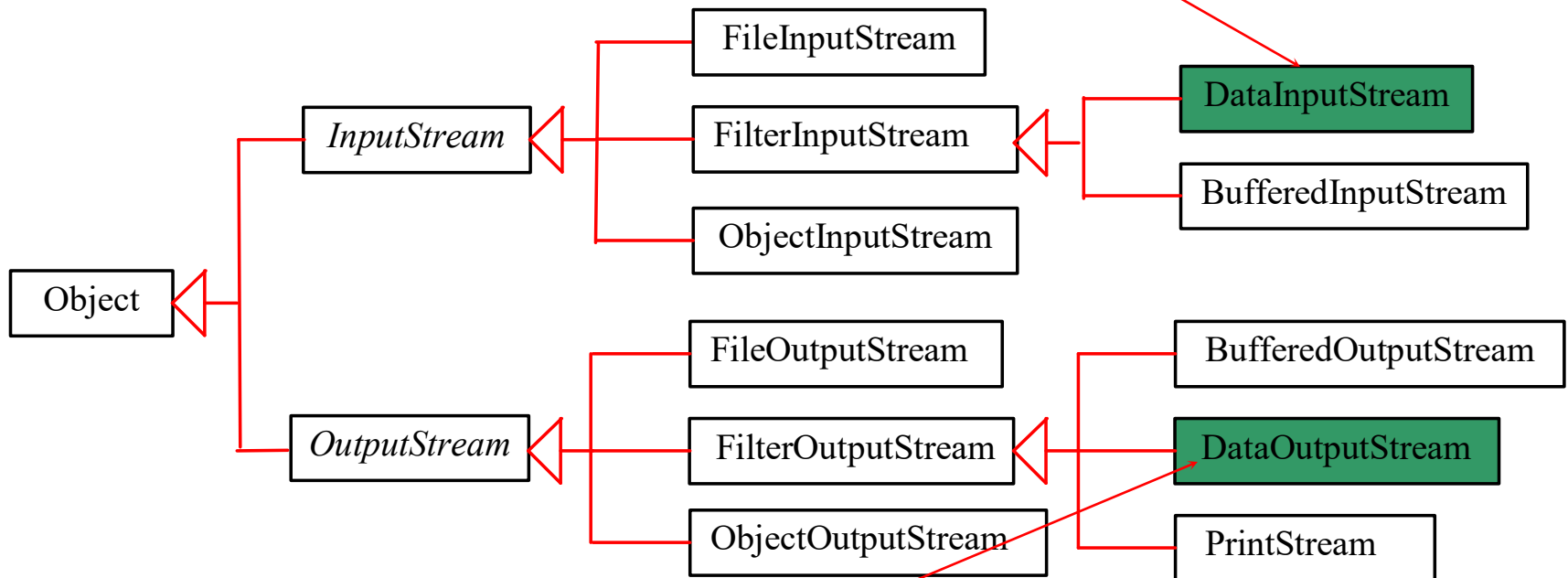
```
public FileInputStream(String filename)
```

```
public FileInputStream(File file)
```

A `java.io.FileNotFoundException` would occur if you attempt to create a `FileInputStream` with a nonexistent file.

DataInputStream/DataOutputStream

DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

Case Studies: Copy File

This case study develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

java Copy source target



```
Command Prompt
C:\book>java Copy Welcome.java Temp.java
Target file Temp.java already exists

C:\book>del Temp.java

C:\book>java Copy Welcome.java Temp.java
The file Welcome.java has 119 bytes
Copy done!

C:\book>java Copy TTT Temp.java
Source file TTT not exist

C:\book>
```

The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, tell the user the file is not found. If the target file already exists, tell the user the file already exists.



```

public static void main(String[] args) throws IOException {
    if (args.length != 2) {                // Check command-line parameter
usage
        System.out.println(
            "Usage: java Copy sourceFile targetfile");
        System.exit(1);    }
    File sourceFile = new File(args[0]); // Check if source file exists
    if (!sourceFile.exists()) {
        System.out.println("Source file " + args[0]    + " does not exist");
        System.exit(2);    }
    File targetFile = new File(args[1]); // Check if target file exists
    if (targetFile.exists()) {
        System.out.println("Target file " + args[1]    + " already exists");
        System.exit(3);    }
    try ( BufferedInputStream input =    // Create an input stream
        new BufferedInputStream(new FileInputStream(sourceFile));
        BufferedOutputStream output = // Create an output stream
        new BufferedOutputStream(new FileOutputStream(targetFile));
    ) { // Continuously read a byte from input and write it to output
        int r, numberOfBytesCopied = 0;
        while ((r = input.read()) != -1) {
            output.write((byte)r);    numberOfBytesCopied++;    }
        System.out.println(numberOfBytesCopied + " bytes copied");
    }
}

```

Chapter 19 Generics



What is Generics?

Generics is the capability to parameterize types. With this capability, you can define a class or a method with generic types that can be substituted using concrete types by the compiler. For example, you may define a generic stack class that stores the elements of a generic type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

Why Generics?

The key benefit of generics is to enable errors to be detected at compile time rather than at runtime. A generic class or method permits you to specify allowable types of objects that the class or method may work with. If you attempt to use the class or method with an incompatible object, a compile error occurs.

Erasure and Restrictions on Generics

Generics are implemented using an approach called *type erasure*. The compiler uses the generic type information to compile the code, but erases it afterwards. So the generic information is not available at run time. This approach enables the generic code to be backward-compatible with the legacy code that uses raw types.

Bounded Generic Type

```
public static void main(String[] args ) {  
    Rectangle rectangle = new Rectangle(2, 2);  
    Circle circle = new Circle (2);  
    System.out.println("Same area? " + equalArea(rectangle, circle));  
}
```

```
public static <E extends GeometricObject> boolean  
    equalArea(E object1, E object2) {  
    return object1.getArea() == object2.getArea();  
}
```

```
public static Boolean  equalArea(GeometricObject object1,  
    GeometricObject object2) {  
    return object1.getArea() == object2.getArea();  
}
```

Wildcards

Why wildcards are necessary? See this example.

WildCardNeedDemo

?	unbounded wildcard
? extends T	bounded wildcard
? super T	lower bound wildcard

AnyWildCardDemo

SuperWildCardDemo

Restrictions on Generics

Generics are limited to reference types.

❑ Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., `new E()`).

❑ Restriction 2: Generic Array Creation is Not Allowed. (i.e., `new E[100]`).

❑ `E[] elements = new E[capacity]` ❑F

```
E[] elements = (E[])new Object[capacity];
```

 ❑T

```
ArrayList<String>[] list = new ArrayList<String>[10];
```

 ❑F

```
ArrayList<String>[] list = (ArrayList<String>[])new ArrayList[10];
```

❑T

Restrictions on Generics

- ❑ Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context.

```
public class Test<E>{  
    public static void m(E o1){  
        }// 静态方法  
    public static E o1;// 数据域  
    static {  
        E o2;  
        }// 初始化语句  
}
```

- ❑ Restriction 4: Exception Classes Cannot be Generic.

```
public class MyException<T> extends Exception{  
}
```

- ❑ Try{...} catch(MyException ex){...}

Thank you and Good luck !

