# 10 Thinking in Objects

Motivations

You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.
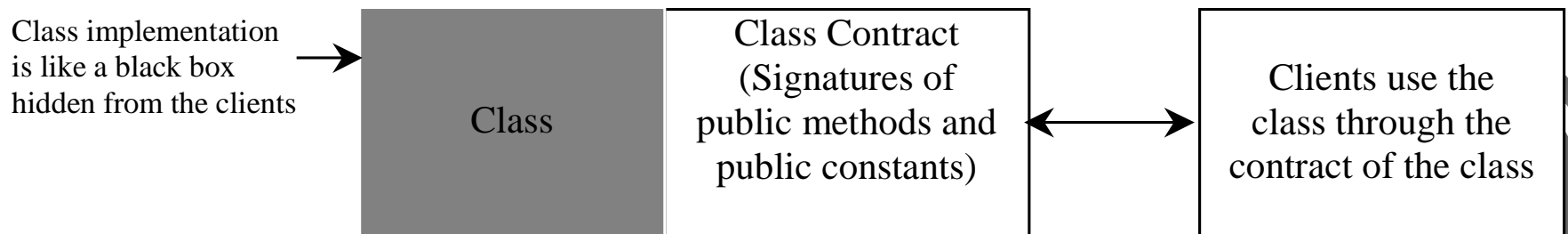
Objectives

❑ To apply class abstraction to develop software (§10.2).

❑ To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).

❑ To discover the relationships between classes (§10.4).

❑ To design programs using the object-oriented paradigm (§§10.5–10.6).

❑ To create objects for primitive values using the wrapper classes (**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, and **Boolean**) (§10.7).

❑ To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).

❑ To use the **BigInteger** and **BigDecimal** classes for computing very large numbers with arbitrary precisions (§10.9).

❑ To use the **String** class to process immutable strings (§10.10).

❑ To use the **StringBuilder** and **StringBuffer** classes to process mutable strings (§10.11).
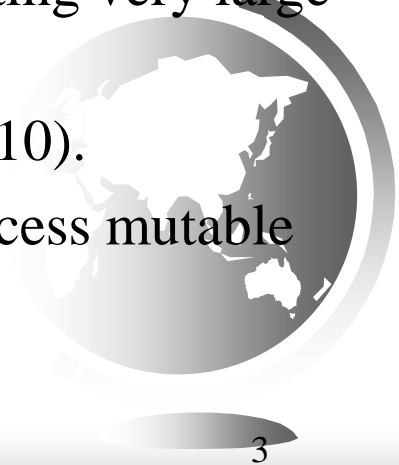
Class Abstraction and Encapsulation

Class abstraction means to <u>separate class implementation from the use of the class</u>. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.

Class implementation is like a black box hidden from the clients →

| Class | Class Contract (Signatures of public methods and public constants) | ↔ | Clients use the class through the contract of the class |

Designing the Loan Class

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double

The annual interest rate of the loan (default: 2.5).

The number of years for the loan (default: 1)

The loan amount (default: 1000).

The date this loan was created.

Constructs a default Loan object.

Constructs a loan with specified interest rate, years, and loan amount.

Returns the annual interest rate of this loan.

Returns the number of the years of this loan.

Returns the amount of this loan.

Returns the date of the creation of this loan.

Sets a new annual interest rate to this loan.

Sets a new number of years to this loan.

Sets a new amount to this loan.

Returns the monthly payment of this loan.

Returns the total payment of this loan.

Loan   TestLoanClass   Run

Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

The BMI Class
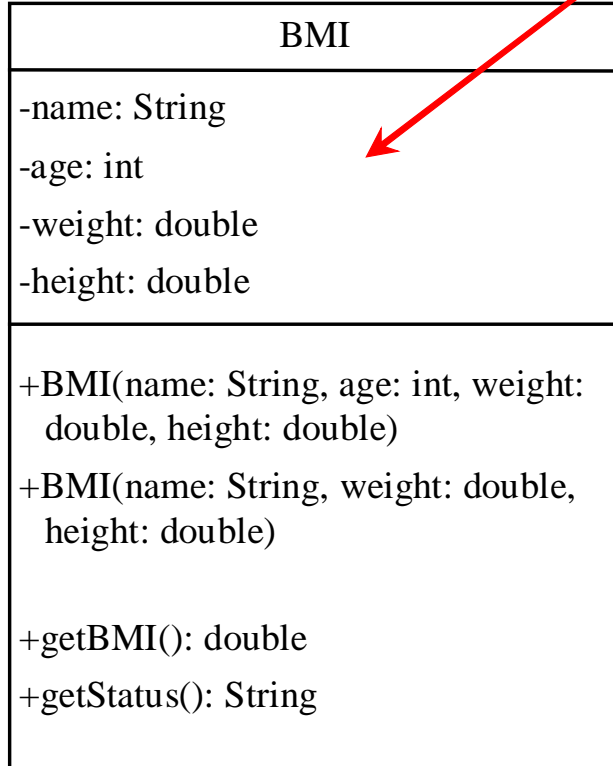
The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

BMI   UseBMIClass   Run

Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

Take ▶

Teach ◀

| Student | 5..60 ─── * | Course | 0..3 ─── 1 | Faculty |

Teacher
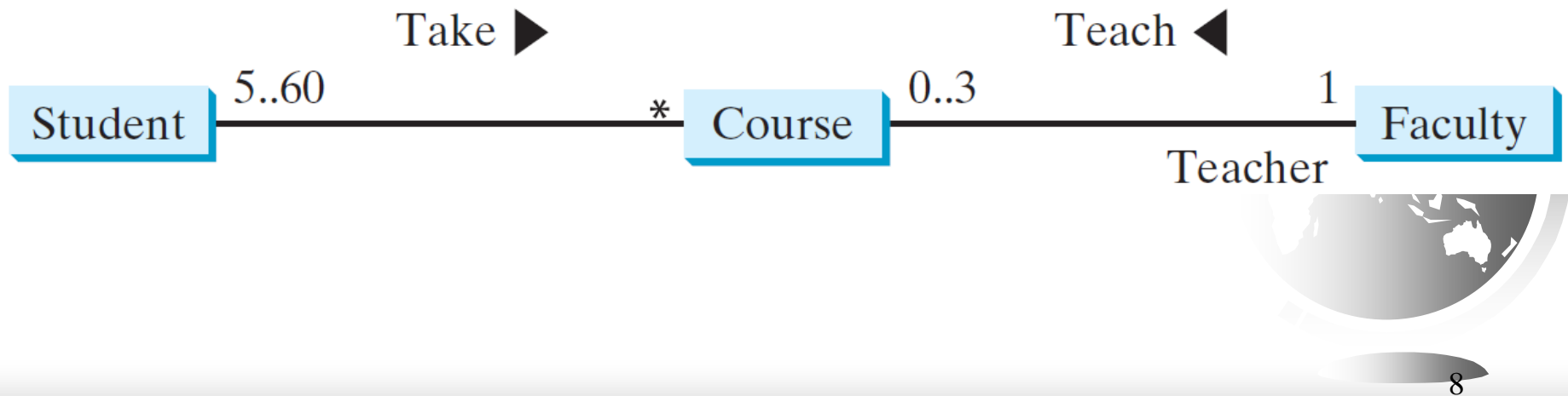
Object Composition

```
Public class Student{
    private Course[]
        courseList;
    public void addCourse(
        Course s){…}
}
```

```
Public class Course{
    private Student[]
        studentList;
    private Faculty faculty;
    public void
    addStudent(
        Student s){…}
            public void
            setFaculty(
                Faculty
            f){…}
}
```

```
Public class Faculty{
    private Course[]
        courseList;
    public void addCourse(
        Course s){…}
}
```

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

Name ——1——1◆ Student ◇——1...3——1 Address

Composition   Aggression
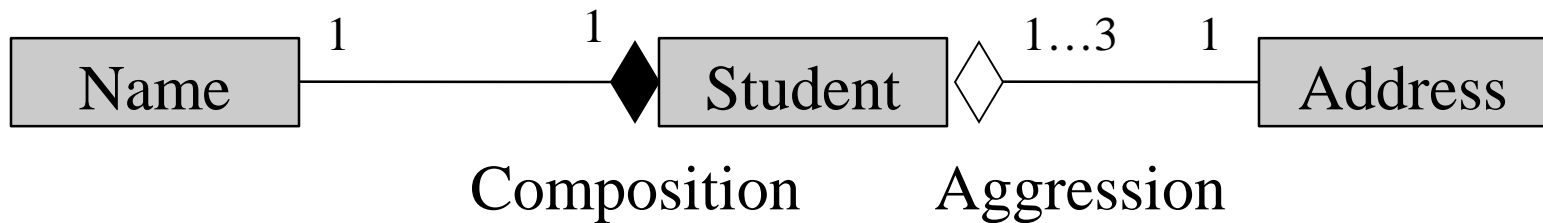
```
public class Name {
    ...
}
```
Aggregated class

```
public class Student {
    private Name name;
    private Address address;

    ...
}
```
Aggregating class

```
public class Address {
    ...
}
```
Aggregated class

Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

类之间的关系：Association, Aggregation, Composition

□ **Association**

  – Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. **Aggregation is a special form of association. Composition is a special form of aggregation.**

Example: A Student and a Faculty are having an association.

类之间的关系：Association, Aggregation, Composition

□ **Aggregation**

– Aggregation is a special case of association. A directional association between objects. When an object 'has-a' another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a "Has-a" relationship.

类之间的关系：Association, Aggregation, Composition

Composition

– Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

Example: A class contains students. A student cannot exist without a class. There exists composition between class and students.

类之间的关系：Association, Aggregation, Composition

- **Difference between aggregation and composition**
  - Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation.
  - In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.
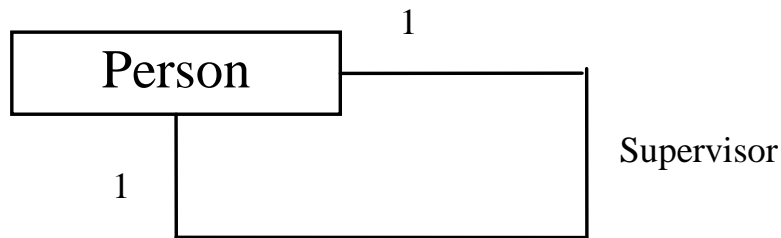
类之间的关系：Association, Aggregation, Composition
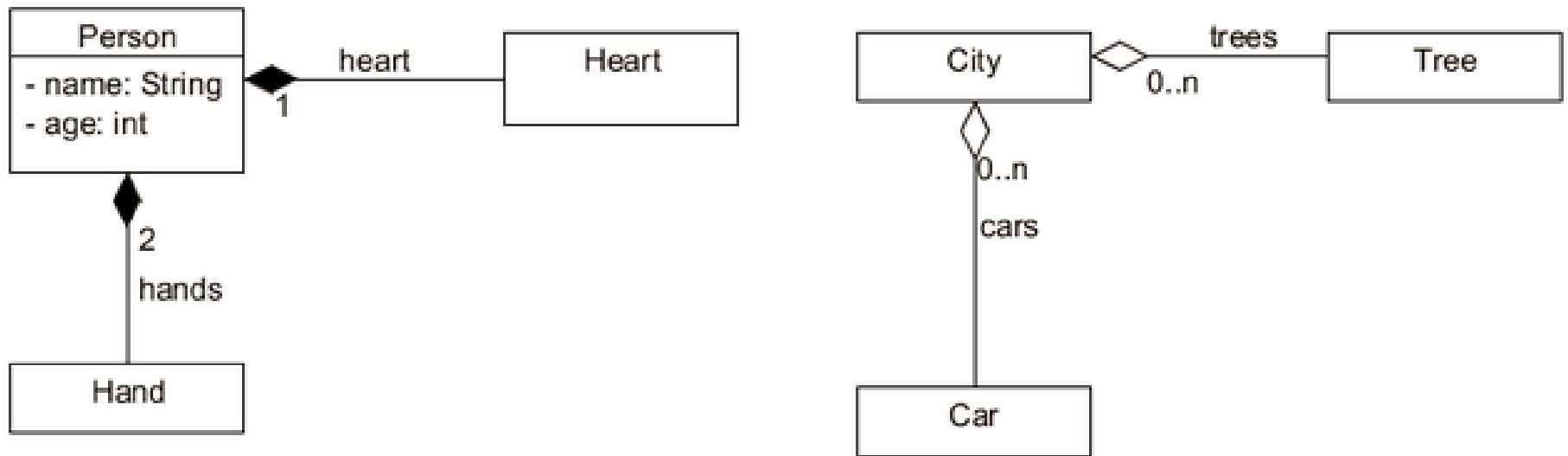
Aggregation Between Same Class

Aggregation may exist between objects of the same class.
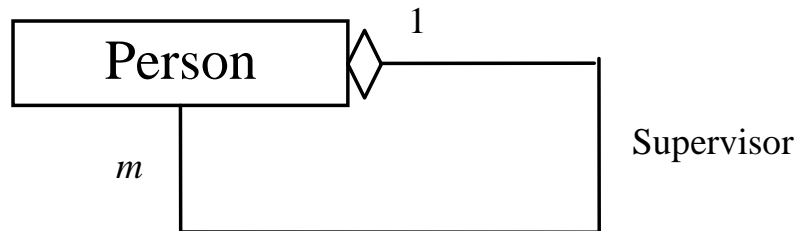For example, a person may have a supervisor.

```
Person
                    1
                         Supervisor
    1
```

```java
public class Person {
    // The type for the data is the class itself
    private Person supervisor;
    ...
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?

Person — 1 — Supervisor, m

```
public class Person {
    ...
    private Person[] supervisors;
}
```

Example: The Course Class

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.
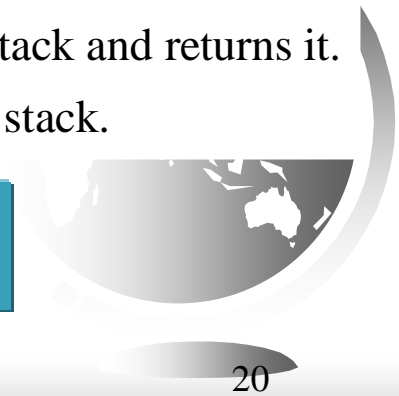
Course   TestCourse   Run

Example: The `StackOfIntegers` Class

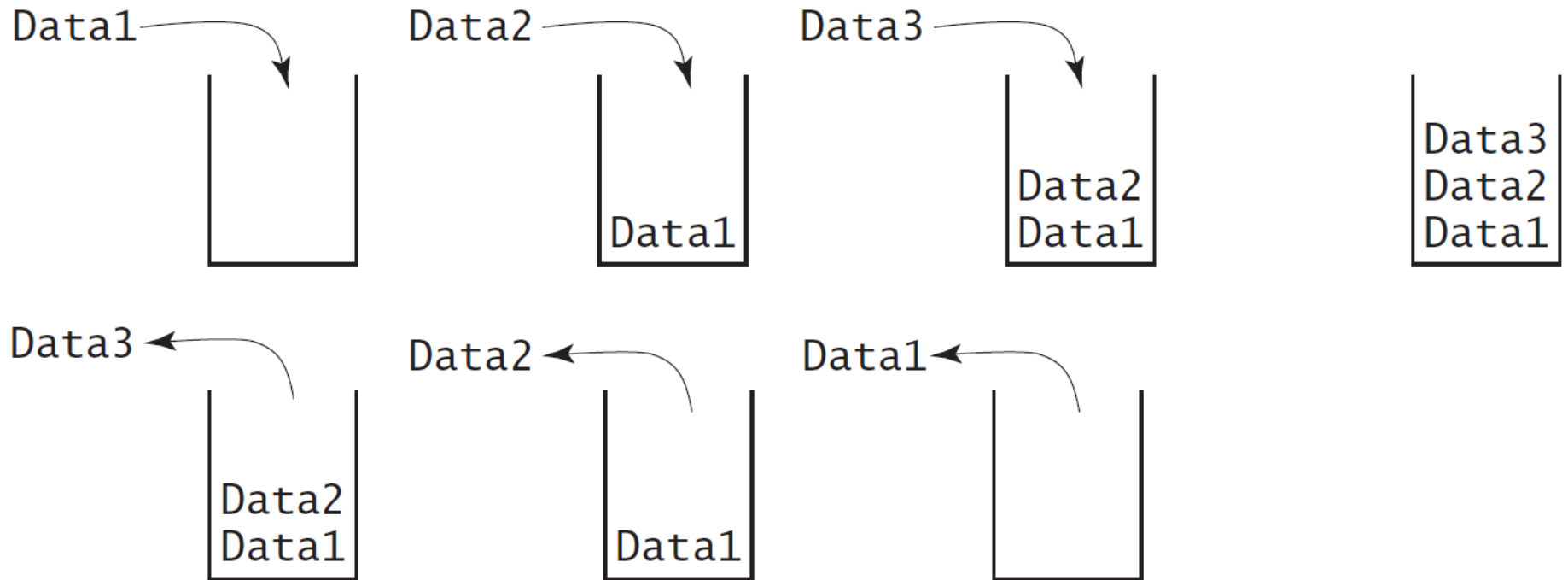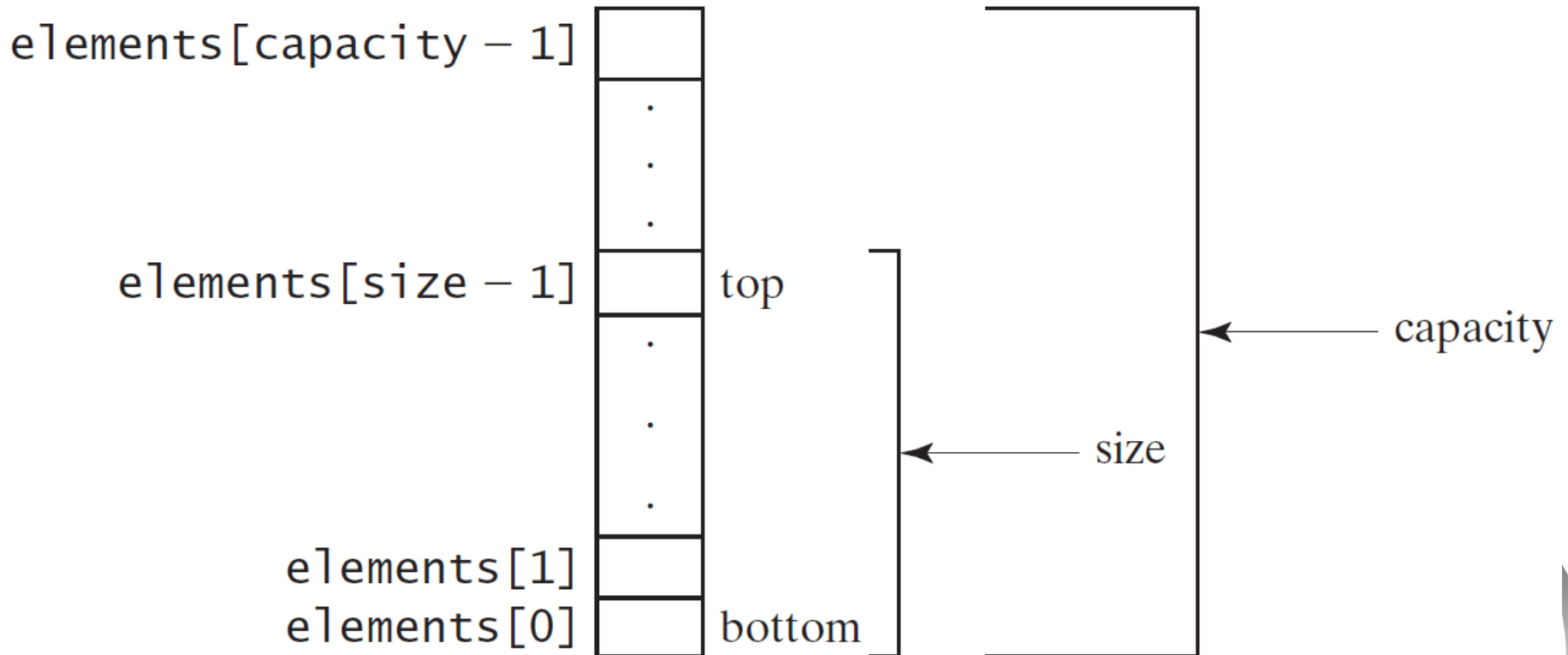| StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): int	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

TestStackOfIntegers   Run

Designing the StackOfIntegers Class

Implementing `StackOfIntegers` Class

elements[capacity - 1]

elements[size - 1]   top

elements[1]

elements[0]   bottom

capacity

size

StackOfIntegers

Wrapper Classes

- Boolean
- Character
- Short
- Byte

- Integer
- Long
- Float
- Double

NOTE:

1) The wrapper classes do not have no-arg constructors.

(2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

The `Integer` and `Double` Classes

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+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 `Integer` Class
and the `Double` Class

❑ Constructors

❑ Class Constants `MAX_VALUE`, `MIN_VALUE`

❑ Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).
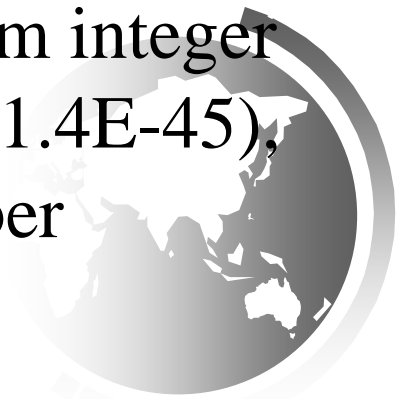
Conversion Methods

Each numeric wrapper class implements the abstract methods <u>doubleValue</u>, <u>floatValue</u>, <u>intValue</u>, <u>longValue</u>, and <u>shortValue</u>, which are defined in the <u>Number</u> class. These methods "convert" objects into primitive type values.

The Static <u>valueOf</u> Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This 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");
```
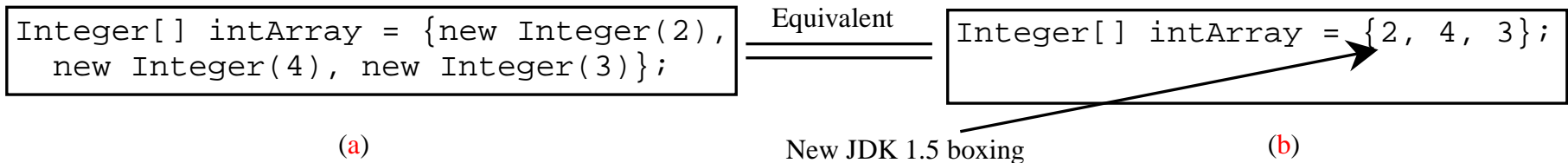
The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

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

Equivalent

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

(a)

New JDK 1.5 boxing

(b)

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

Unboxing

BigInteger and BigDecimal

If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package. Both are *immutable*. Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

```java
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

```java
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

自动装箱（autoboxing）的问题：

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

这段程序答案是正确的，但比实际情况更慢一些（6.8秒->43秒)。

The `String` Class

❑ Constructing a String:

```
String message = "Welcome to Java";
String message = new String("Welcome to Java");
String s = new String();
```

❑ Obtaining String length and Retrieving Individual Characters in a string

❑ String Concatenation (concat)

❑ Substrings (substring(index), substring(start, end))

❑ Comparisons (equals, compareTo)

❑ String Conversions

❑ Finding a Character or a Substring in a String

❑ Conversions between Strings and Arrays

❑ Converting Characters and Numeric Values to Strings

Constructing Strings

String newString = new String(stringLiteral);

String message = new String("Welcome to Java");

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

String message = "Welcome to Java";

Strings Are Immutable

A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?
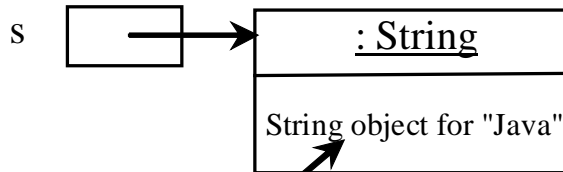
```
String s = "Java";
s = "HTML";
```

Trace Code

String s = "Java";

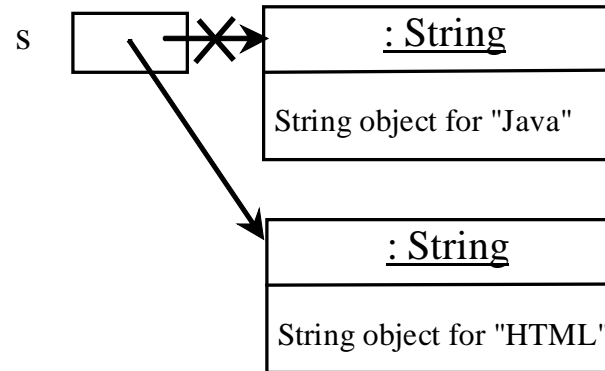s = "HTML";

After executing `String s = "Java";`

s : String

String object for "Java"

Contents cannot be changed

After executing `s = "HTML";`

s : String

String object for "Java"

This string object is now unreferenced

: String

String object for "HTML"

Trace Code

String s = "Java";

s = "HTML";

After executing `String s = "Java";`

s

: String
String object for "Java"

Contents cannot be changed

After executing `s = "HTML";`

s
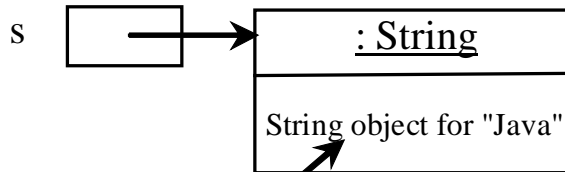
: String
String object for "Java"

This string object is now unreferenced
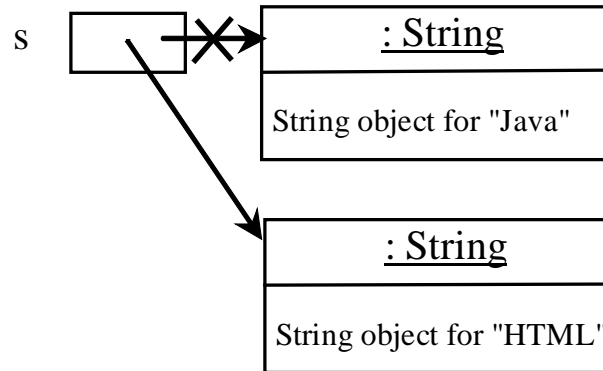
: String
String object for "HTML"

Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, **the JVM uses a unique instance for string literals with the same character sequence**. Such an instance is called *interned*. For example, the following statements:
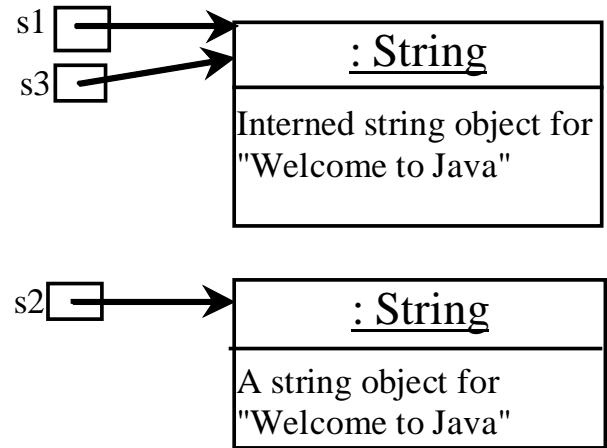
Examples

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

s1 →
s3 →

: String
Interned string object for "Welcome to Java"

s2 →

: String
A string object for "Welcome to Java"

display

  s1 == s is false

  s1 == s3 is true

A new object is created if you use the new operator.

If you use the string initializer, no new object is created if the interned object is already created.
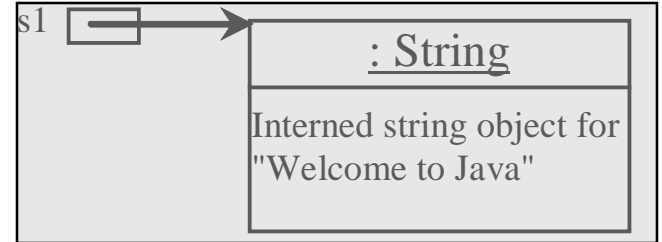
Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```

s1

<table>
<tr><td align="center">: String</td></tr>
<tr><td>Interned string object for "Welcome to Java"</td></tr>
</table>

Trace Code

```java
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```

s1 →

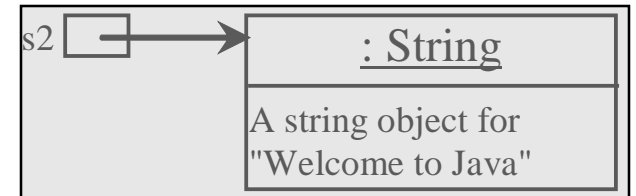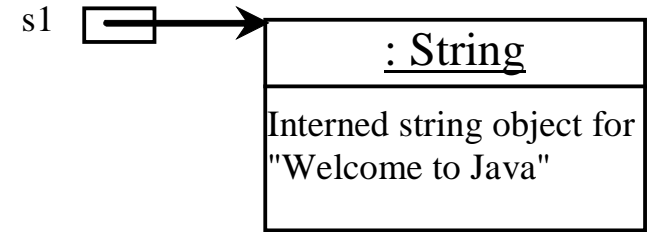| : String |
|---|
| Interned string object for "Welcome to Java" |

s2 →

: String
A string object for "Welcome to Java"

Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```

s1

s3

: String
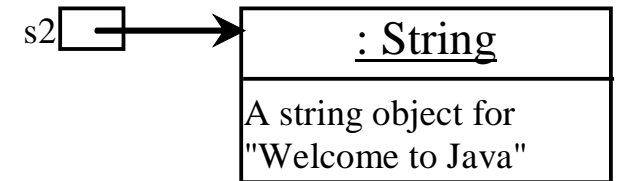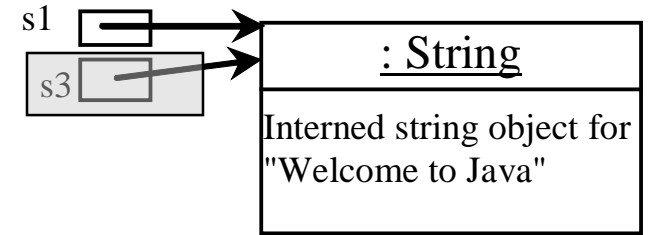
Interned string object for "Welcome to Java"

s2

: String

A string object for "Welcome to Java"

Constant Pool (Java常量池技术)

- java中的常量池技术，是为了方便快捷地创建某些对象而出现的，当需要一个对象时，就可以从池中取一个出来（如果池中没有则创建一个），则在需要重复创建相等变量时节省了很多时间。常量池其实也就是一个内存空间，常量池存在于方法区中。

- JVM的编译器将源程序编译成class文件后，会用一部分字节分类存储常量。这些常量集中在class中的一个区域存放，一个紧接着一个，就称为"常量池"。其中包括了关于类，方法，接口等中的常量，也包括字符串常量，如String s ="java"这种申明方式；对于String s = "java"，在编译成.class时能够识别为同一字符串的,自动优化成常量,所以如果有多个字符串"java"，则它们都会引用自同一String对象。也就是说String s = "java"其中"java"值在JAVA程序编译期就确定下来了的。

Constant Pool (Java常量池技术)

```java
package test;

public class ConstantPool2 {

    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Hel" + "lo";
        String s4 = "Hel" + new String("lo");
        String s5 = new String("Hello");
        String s6 = s5.intern();
        String s7 = "H";
        String s8 = "ello";
        String s9 = s7 + s8;

        System.out.println(s1 == s2);
        System.out.println(s1 == s3);
        System.out.println(s1 == s4);
        System.out.println(s1 == s9);
        System.out.println(s4 == s5);
        System.out.println(s1 == s6);

    }
}
```

Constant Pool (Java常量池技术)

Constant Pool (Java常量池技术)

```java
package test;

public class ConstantPool2 {

    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Hel" + "lo";
        String s4 = "Hel" + new String("lo");
        String s5 = new String("Hello");
        String s6 = s5.intern();
        String s7 = "H";
        String s8 = "ello";
        String s9 = s7 + s8;

        System.out.println(s1 == s2);
        System.out.println(s1 == s3);
        System.out.println(s1 == s4);
        System.out.println(s1 == s9);
        System.out.println(s4 == s5);
        System.out.println(s1 == s6);

    }
}
```

true
true
false
false
false
true

s9果然是这样接出来

s1 == s6这两个相等完全归功于intern方法，s5在堆中，内容为Hello，intern方法会尝试将Hello字符串添加到常量池中，并返回其在常量池中的地址，因为常量池中已经有了Hello字符串，所以intern方法直接返回地址；而s1在编译期就已经指向常量池了，因此s1和s6指向同一地址，相等。

池中的s1地址相同。

48

反编译结果：

```java
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Hello";
String s4 = (new StringBuilder("Hel")).append(new String("lo")).toString();
String s5 = new String("Hello");
String s6 = s5.intern();
String s7 = "H";
String s8 = "ello";
String s9 = (new StringBuilder(String.valueOf(s7))).append(s8).toString();
System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1 == s4);
System.out.println(s1 == s9);
System.out.println(s4 == s5);
System.out.println(s1 == s6);
```

true
true
false
false
false
true

Constant Pool (Java常量池技术)

- java中基本类型的包装类的大部分都实现了常量池技术，这些类是Byte,Short,Integer,Long,Character,Boolean

- 两种浮点数类型的包装类则没有实现。

- Byte,Short,Integer,Long,Character这5种整型的包装类也只是对从-128 到 127的对象使用对象池，也即对象不负责创建和管理大于127和小于-128的这些类的对象。

- 利用缓存机制实现常量池：为了减少不必要的内存消耗和内存开辟次数，Integer 里做了一个缓存，缓存了从 -128 到 127 之间的 Integer 对象，总共是256个对象。

```java
Integer i11 = 126;
Integer i12 = 126;
Integer i13 = new Integer(126);
Integer i14 = 125+1;
Integer i15 = 125 + new Integer(1);
Integer i21 = 128;
Integer i22 = 128;
Integer i23 = new Integer(128);
Integer i24 = 127+1;
Integer i25 = 127+new Integer(1);

System.out.println(i11==i12);
System.out.println(i11==i13);
System.out.println(i11==i14);
System.out.println(i11==i15);
System.out.println("-----------");
System.out.println(i21==i22);
System.out.println(i21==i23);
System.out.println(i21==i24);
System.out.println(i21==i25);
```

先是intValue，然后+，再valueOf

自动装箱的过程，调用了Integer.valueOf(…)

true
true
----
false
false
false
false

如何分析？单步调试（看调用哪个函数） 或反编译

```java
Integer i11 = Integer.valueOf(126);
Integer i12 = Integer.valueOf(126);
Integer i13 = new Integer(126);
Integer i14 = Integer.valueOf(126);
Integer i15 = Integer.valueOf(125 + (new Integer(1)).intValue());
Integer i21 = Integer.valueOf(128);
Integer i22 = Integer.valueOf(128);
Integer i23 = new Integer(128);
Integer i24 = Integer.valueOf(128);
Integer i25 = Integer.valueOf(127 + (new Integer(1)).intValue());
System.out.println(i11 == i12);
System.out.println(i11 == i13);
System.out.println(i11 == i14);
System.out.println(i11 == i15);
System.out.println("------------");
System.out.println(i21 == i22);
System.out.println(i21 == i23);
System.out.println(i21 == i24);
System.out.println(i21 == i25);
```

```java
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
```

```java
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

```java
int i31 = 126;
int i32 = 126;
int i33 = new Integer(126);
int i34 = 125+new Integer(1);
int i41 = 128;
int i42 = 128;
int i43 = new Integer(128);
int i44 = 127+new Integer(1);
System.out.println(i31==i32);
System.out.println(i31==i33);
System.out.println(i31==i34);
System.out.println("-----------");
System.out.println(i41==i42);
System.out.println(i41==i43);
System.out.println(i41==i44);
```

true
true
true
----
true
true
true

```java
int i31 = 126;
int i32 = 126;
int i33 = (new Integer(126)).intValue();
int i34 = 125 + (new Integer(1)).intValue();
int i41 = 128;
int i42 = 128;
int i43 = (new Integer(128)).intValue();
int i44 = 127 + (new Integer(1)).intValue();
System.out.println(i31 == i32);
System.out.println(i31 == i33);
System.out.println(i31 == i34);
System.out.println("------------");
System.out.println(i41 == i42);
System.out.println(i41 == i43);
System.out.println(i41 == i44);
```

```
class Test {
    public static void main(String[] args) {
        Integer a = new Integer(3);
        Integer b = 3;              ← valueOf(3)
        int c = 3;
        System.out.println(a == b);
        System.out.println(a == c);          false
}}                                            true
                 intValue(a)
```

尽量使用 Integer.valueOf(int) 方法来获取一个 Integer 对象，而不使用 new Integer(int) 来构造 Integer 对象。

```java
Double d1 = 1.1;
Double d2 = 1.1;
Double d3 = new Double(1.1);
Double d4 = new Double(1.1);
System.out.println(d1==d2);
System.out.println(d1==d3);
System.out.println(d3==d4);
System.out.println(d1.equals(d2));
System.out.println(d1.equals(d3));
System.out.println(d3.equals(d4));
```

false
false
false
true
true
true

```java
public static Double valueOf(double d) {
    return new Double(d);
}
```

```java
Double d1 = Double.valueOf(1.1000000000000001D);
Double d2 = Double.valueOf(1.1000000000000001D);
Double d3 = new Double(1.1000000000000001D);
Double d4 = new Double(1.1000000000000001D);
System.out.println(d1 == d2);
System.out.println(d1 == d3);
System.out.println(d3 == d4);
System.out.println(d1.equals(d2));
System.out.println(d1.equals(d3));
System.out.println(d3.equals(d4));
```

String的intern方法*

- 了解即可   Jdk版本为1.7以上

```java
public static void main(String[] args) {
    String str1 = new String("ZJU")+ new String("LWM");
    System.out.println(str1.intern() == str1);
    System.out.println(str1 == "ZJULWM");
}
```

```
<terminated> TestStringIntern [Java App
true
true
```

```java
public static void main(String[] args) {
    String str2 = "ZJULWM";//新加的一行代码，其余不变

    String str1 = new String("ZJU")+ new String("LWM");
    System.out.println(str1.intern() == str1);
    System.out.println(str1 == "ZJULWM");
}
```

```
<terminated> TestStringIntern [Java App
false
false
```

**涉及到常量池的位置
与intern的实现方法**
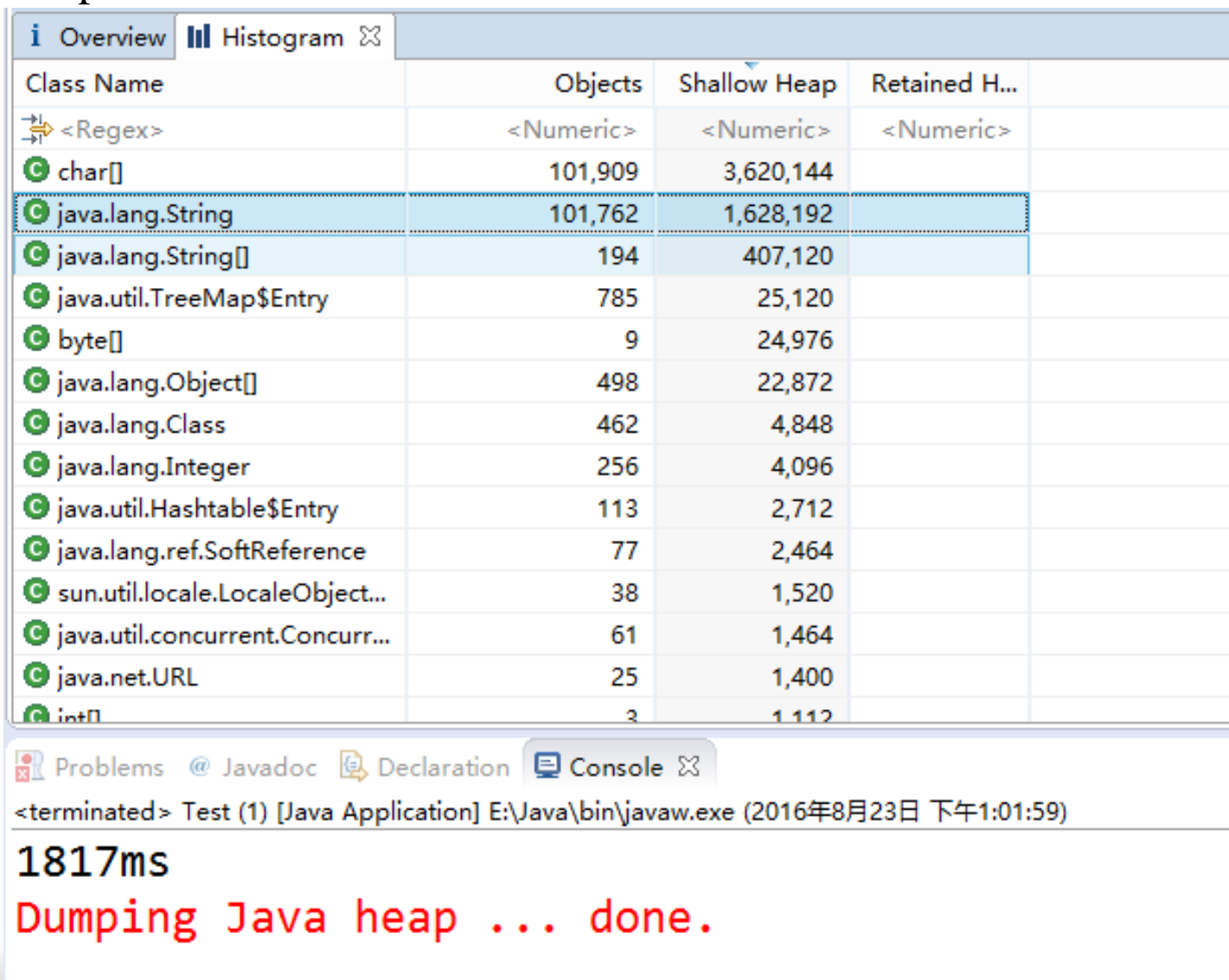
String的intern方法*

- intern()方法设计的初衷，就是重用String对象，以节省内存消耗。

```java
static final int MAX = 100000;
static final String[] arr = new String[MAX];

public static void main(String[] args) throws Exception {
    //为长度为10的Integer数组随机赋值
    Integer[] sample = new Integer[10];
    Random random = new Random(1000);
    for (int i = 0; i < sample.length; i++) {
        sample[i] = random.nextInt();
    }
    //记录程序开始时间
    long t = System.currentTimeMillis();
    //使用/不使用intern方法为10万个String赋值，值来自于Integer数组的10个数
        for (int i = 0; i < MAX; i++) {
            arr[i] = new String(String.valueOf(sample[i % sample.length]));
            //arr[i] = new String(String.valueOf(sample[i % sample.length])).intern();
        }
        System.out.println((System.currentTimeMillis() - t) + "ms");
        System.gc();
}
```

- 两种情况分别运行，可通过Window ---> Preferences --> Java --> Installed JREs设置JVM启动参数为-agentlib:hprof=heap=dump,format=b，将程序运行完后的hprof置于工程目录下。再通过MAT插件(Memory Analyzer)查看该hprof文件。

Class Name	Objects	Shallow Heap	Retained H...
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	101,909	3,620,144	
java.lang.String	101,762	1,628,192	
java.lang.String[]	194	407,120	
java.util.TreeMap$Entry	785	25,120	
byte[]	9	24,976	
java.lang.Object[]	498	22,872	
java.lang.Class	462	4,848	
java.lang.Integer	256	4,096	
java.util.Hashtable$Entry	113	2,712	
java.lang.ref.SoftReference	77	2,464	
sun.util.locale.LocaleObject...	38	1,520	
java.util.concurrent.Concurr...	61	1,464	
java.net.URL	25	1,400	
int[]	3	1,112	

Problems @ Javadoc Declaration Console ⌗

\<terminated> Test (1) [Java Application] E:\Java\bin\javaw.exe (2016年8月23日 下午1:01:59)

1817ms

Dumping Java heap ... done.

Class Name	Objects	Shallow Heap	Retained H
<Regex>	<Numeric>	<Numeric>	<Numeri
java.lang.String[]	194	407,120	
char[]	1,919	260,496	
java.lang.String	1,772	28,352	
java.util.TreeMap$Entry	785	25,120	
byte[]	9	24,976	
java.lang.Object[]	498	22,872	
java.lang.Class	462	4,848	
java.lang.Integer	256	4,096	
java.util.Hashtable$Entry	113	2,712	
java.lang.ref.SoftReference	77	2,464	
sun.util.locale.LocaleObjectC...	38	1,520	
java.util.concurrent.Concurre...	61	1,464	
java.net.URL	25	1,400	
int[]	3	1,112	

用intern只生成1772个String对象，否则将生成10万多的对象

时间会慢些，因为程序中每次都是用了new String后又进行intern()操作的耗时时间，但是不使用intern()占用内存空间导致GC的时间是要远远大于这点时间的。

Problems @ Javadoc Declaration Console

<terminated> Hello [Java Application] E:\Java\bin\javaw.exe (2016年8月

1858ms

Dumping Java heap ... done.

String的intern方法*

JVM **Runtime Data Area**
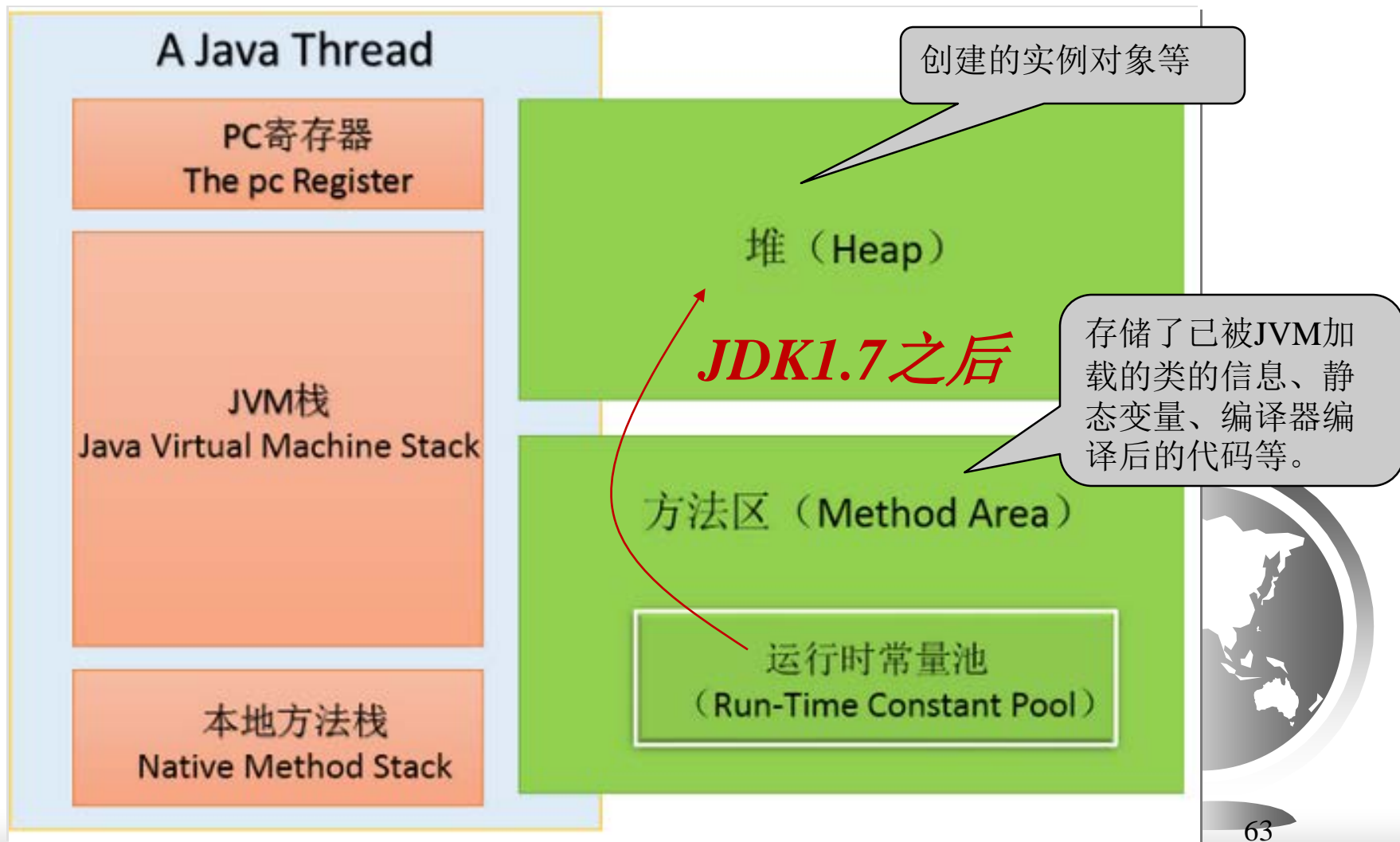
String的intern方法*

- JDK1.7后，常量池被放入到堆空间中，这导致intern()函数的功能不同。

- intern()方法在JDK1.6中的作用是：比如String s = new String("ZJULWM")，再调用s.intern()，此时返回值还是字符串"ZJULWM"，表面上看起来好像这个方法没什么用处。但实际上，在JDK1.6中它做了个小动作：检查常量池里是否存在"ZJULWM"这么一个字符串，如果存在，就返回池里的字符串；如果不存在，该方法会把"ZJULWM"添加到常量池中，然后再返回它的引用。

- 在JDK1.7中却不是这样的，后面会讨论。

String的intern方法*

```
01.    String s = new String("1");
02.    s.intern();
03.    String s2 = "1";
04.    System.out.println(s == s2);
05.
06.    String s3 = new String("1") + new String("1");
07.    s3.intern();
08.    String s4 = "11";
09.    System.out.println(s3 == s4);
```

输出结果为：

```
[java]
01.    JDK1.6以及以下: false false
02.    JDK1.7以及以上: false true
```

String的intern方法*

- 再分别调整上面代码2.3行、7.8行的顺序：

```java
01.  String s = new String("1");
02.  String s2 = "1";
03.  s.intern();
04.  System.out.println(s == s2);
05.
06.  String s3 = new String("1") + new String("1");
07.  String s4 = "11";
08.  s3.intern();
09.  System.out.println(s3 == s4);
```

输出结果为：

```
[java]
01.  JDK1.6以及以下: false false
02.  JDK1.7以及以上: false false
```

□ JDK1.6

堆空间

Perm Space

String obj

s3.intern()

"11"

Java Heap

栈

s4引用

s3引用

s2引用

s引用

String Pool

"1"

String obj

s.intern

jdk6 代码图

```
String s = new String("1");
s.intern();
String s2 = "1";
System.out.println(s == s2);

String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3 == s4);
```

1.6中，s2和s4是常量池中的地址，而s和s3是堆空间中的地址，因此是不一样的。

什么都不做（"1"已在常量池中了）

会将"11"放入常量池（"11"不在常量池中）

JDK1.7 (第一次)

堆空间

Java Heap — String Pool — Perm space

1. 生成了常量池中的 "1" 和堆空间中的字符串对象
2. s对象去常量池中寻找后发现"1"已经存在于常量池中了。
3. 生成一个s2的引用指向常量池中的 "1"对象。
4. s 和 s2 的引用地址明显不同

1. 生成了常量池中的 "1" 和堆空间中的字符串对象"11"，常量池中没有 "11"
2. s3对象去常量池中寻找后发现"11"不存在于常量池中。JDK1.6中直接在常量池中生成一个 "11" 的对象；JDK1.7中常量池中不需要再存储一份对象了，直接存储堆中的引用。所以s3.intern()==s3 返回true
3. 直接去常量池中创建，但是发现已经有这个对象了，此时也就是指向 s3 引用对象的一个引用。
4. s3== s4 返回true

```
01.  String s = new String("1");
02.  s.intern();
03.  String s2 = "1";
04.  System.out.println(s == s2);
05.
06.  String s3 = new String("1") + new Stri
07.  s3.intern();
08.  String s4 = "11";
09.  System.out.println(s3 == s4);
```

JDK1.7 (第二次)
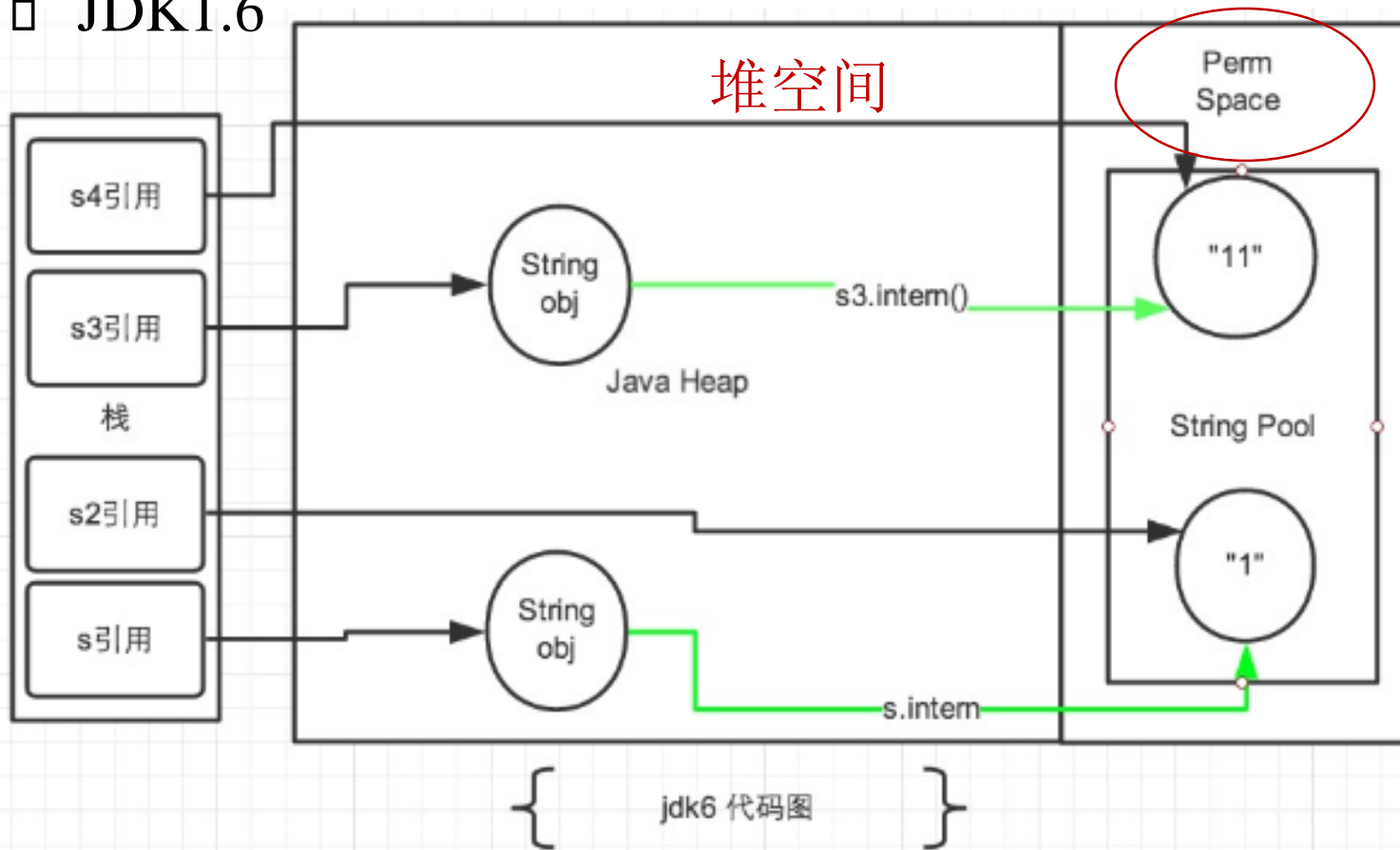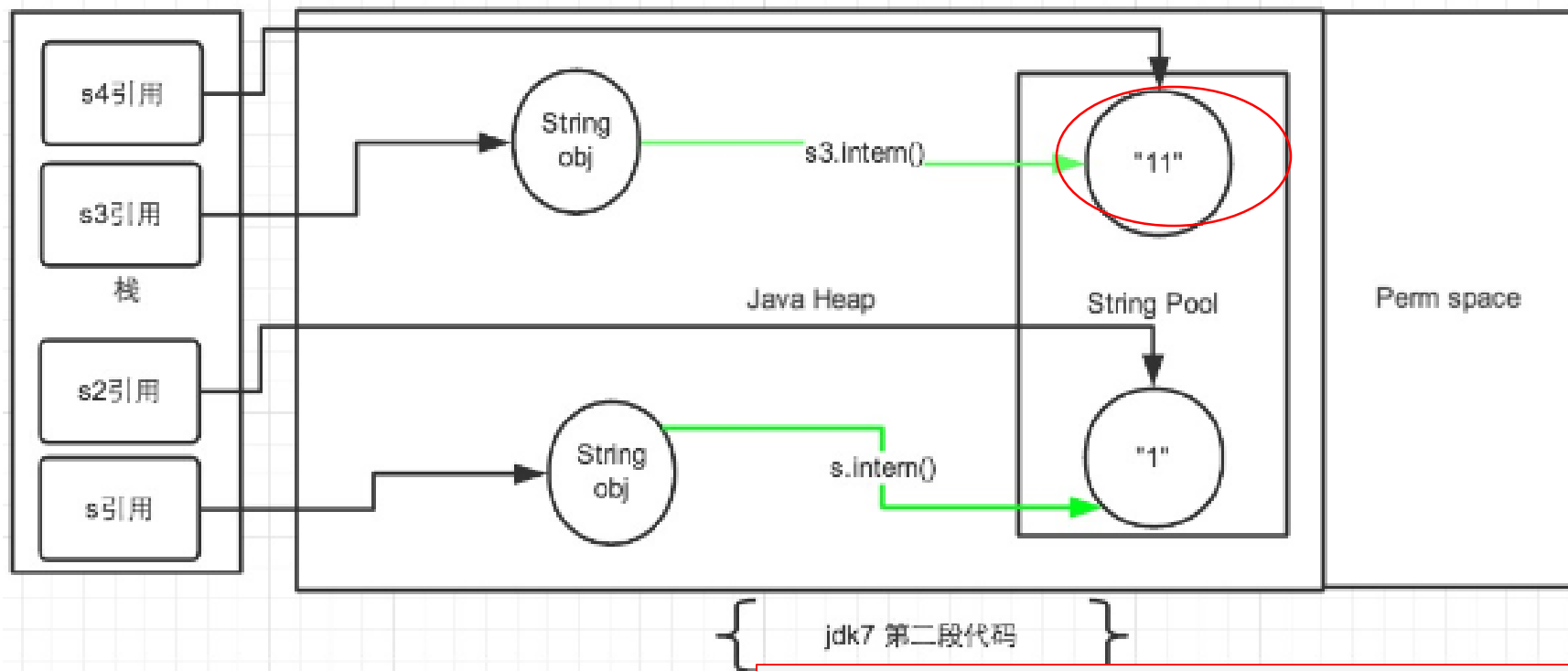


```
String s = new String("1");
String s2 = "1";
s.intern();
System.out.println(s == s2);
```

```
String s3 = new String("1") + new String("1");
String s4 = "11";
s3.intern();
System.out.println(s3 == s4);
```

1.生成了常量池中的"1"和堆空间中的字符串对象
2.生成一个s2的引用指向常量池中的"1"对象，但是发现已经存在了，那么就直接指向它。
3.这一行在这里就没什么实际作用了。因为"1"已经存在了。
4.s 和 s2 的引用地址明显不同

1.在字符串常量池中生成"1"，并在堆空间中生成s3引用指向的对象（内容为"11"）。
2.直接去生成常量池中的"11"
3.这一行在这里就没什么实际作用了。因为"11"已经存在了。
4.s3 和 s4 的引用地址明显不同。因此返回了false

```java
public static void main(String[] args) {
    String str1 = new String("ZJU")+ new String("LWM");
    System.out.println(str1.intern() == str1);
    System.out.println(str1 == "ZJULWM");
}
```

JDK1.7

```
<terminated> TestStringIntern [Java Ap
true
true
```

生成常量池中的"ZJU"和"LWM"，并生成堆空间的"ZJULWM"；
str1是指向堆空间的"ZJULWM"，str1.intern()会查找常量池中是否有"ZJULWM"，
　　发现没有则在常量池中生成"ZJULWM"，内容是堆空间中的"ZJULWM"的引用
　　。
最后"ZJULWM"返回的也是常量池的地址，也就是堆空间中的"ZJULWM"的引用。
所以都是true。

```java
public static void main(String[] args) {
    String str2 = "ZJULWM";//新加的一行代码，其余不变

    String str1 = new String("ZJU")+ new String("LWM");
    System.out.println(str1.intern() == str1);
    System.out.println(str1 == "ZJULWM");

}
```

```
<terminated> TestStringIntern [Java App
false
false
```

生成常量池中的"ZJULWM"； str1是指向堆空间的"ZJULWM"； str1.intern()会查找
　　常量池中是否有"ZJULWM"，发现有则返回常量池中的地址（而这个地址不是
　　堆空间的地址），所以与str1是不一样的。
最后"ZJULWM"返回的也是常量池的地址，这也与堆空间中的"ZJULWM"的引用不
　　一样；所以都是false。

Replacing and Splitting Strings

java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

Examples

"Welcome".replace('e', 'A') returns a new string, WAlcomA.

"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replace("e", "AB") returns a new string, WABlcomAB.

"Welcome".replace("el", "AB") returns a new string, WABcome.

Splitting a String

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

Java HTML Perl

Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as *regular expression*. Regular expression is complex to beginning students. For this reason, two simple patterns are used in this section. Please refer to Supplement III.F, "Regular Expressions," for further studies.

"Java".matches("Java");

"Java".equals("Java");

"Java is fun".matches("Java.*");

"Java is cool".matches("Java.*");

Matching, Replacing and Splitting by Patterns

The replaceAll, replaceFirst, and split methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

String s = "a+b$#c".replaceAll("[$+#]", "NNN");

System.out.println(s);

Here the regular expression [$+#] specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNNc.

Matching, Replacing and Splitting by Patterns

The following statement splits the string into an array of strings delimited by some punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

Convert Character and Numbers to Strings

The String class provides several **static** valueOf methods for converting a character, an array of characters, and numeric values to strings. These methods have the same name valueOf with different argument types char, char[], double, long, int, and float. For example, to convert a double value to a string, use String.valueOf(5.44). The return value is string consists of characters '5', '.', '4', and '4'.

```java
public static String valueOf(int i) {
    return Integer.toString(i);
}
```

```java
public static String valueOf(long l) {
    return Long.toString(l);
}
```

```java
public static String valueOf(char data[]) {
    return new String(data);
}
```

```java
public static String valueOf(float f) {
    return Float.toString(f);
}
```

```java
public static String valueOf(double d) {
    return Double.toString(d);
}
```

```java
public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}
```

```java
public static String valueOf(boolean b) {
    return b ? "true" : "false";
}
```

```java
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

String format

☐ String.format(format, item1, item2,…,itemk)

☐ String s = String.format("%7.2f%6d-4s",45.556, 14, "AB");

		45.56				14	AB		

10 Things Every Java Programmer Should Know about String

☐ 1. Strings are not null terminated in Java

☐ 2. Strings are immutable and final in Java

☐ 3. Strings are maintained in String Pool

☐ 4. Use equals methods for comparing

☐ 5. Use indexOf() and lastIndexOf() or matches(String regex) method to search inside String

☐ 6. Use subString to get part of String in Java

☐ 7. "+" is overloaded for String concatenation

☐ 8. Use trim() to remove white spaces from String

☐ 9. Use split() for splitting String using Regular expression

☐ 10. Don't store sensitive data in String

`StringBuilder` and `StringBuffer`

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.

StringBuilder Constructors

java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

Modifying Strings in the Builder

java.lang.StringBuilder
+append(data: char[]): StringBuilder
+append(data: char[], offset: int, len: int): StringBuilder
+append(v: *aPrimitiveType*): StringBuilder
+append(s: String): StringBuilder
+delete(startIndex: int, endIndex: int): StringBuilder
+deleteCharAt(index: int): StringBuilder
+insert(index: int, data: char[], offset: int, len: int): StringBuilder
+insert(offset: int, data: char[]): StringBuilder
+insert(offset: int, b: *aPrimitiveType*): StringBuilder
+insert(offset: int, s: String): StringBuilder
+replace(startIndex: int, endIndex: int, s: String): StringBuilder
+reverse(): StringBuilder
+setCharAt(index: int, ch: char): void

Appends a char array into this string builder.

Appends a subarray in data into this string builder.

Appends a primitive type value as a string to this builder.

Appends a string to this string builder.

Deletes characters from startIndex to endIndex.

Deletes a character at the specified index.

Inserts a subarray of the data in the array to the builder at the specified index.

Inserts data into this builder at the position offset.

Inserts a value converted to a string into this builder.

Inserts a string into this builder at the position offset.

Replaces the characters in this builder from startIndex to endIndex with the specified string.

Reverses the characters in the builder.

Sets a new character at the specified index in this builder.

Examples

stringBuilder.append("Java");

stringBuilder.insert(11, "HTML and ");

stringBuilder.delete(8, 11) changes the builder to Welcome Java.

stringBuilder.deleteCharAt(8) changes the builder to Welcome o Java.

stringBuilder.reverse() changes the builder to avaJ ot emocleW.

stringBuilder.replace(11, 15, "HTML")
    changes the builder to Welcome to HTML.

stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java.

The toString, capacity, length, setLength, and charAt Methods

java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex-1.
+trimToSize(): void	Reduces the storage size used for the string builder.
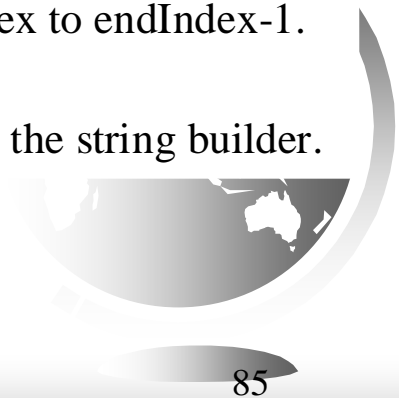
Problem: Checking Palindromes Ignoring Non-alphanumeric Characters

This example gives a program that counts the number of occurrence of each letter in a string. Assume the letters are not case-sensitive.

PalindromeIgnoreNonAlphanumeric   Run

String VS. StringBuilder VS. StringBuffer

□ *1*.三者在执行速度方面的比较：
StringBuilder > StringBuffer > String

```
1 String s = "abcd";
2 s = s+1;
3 System.out.print(s);// result : abcd1
```

明明就是改变了String型的变量s的，为什么说是没有改变呢？
其实这是一种欺骗，JVM是这样解析这段代码的：首先创建对象s，赋予一个abcd，然后再创建一个新的对象s用来执行第二行代码，也就是说我们之前对象s并没有变化，所以我们说String类型是不可改变的对象了，由于这种机制，每当用String操作字符串时，实际上是在不断的创建新的对象，而原来的对象就会变为垃圾被GC回收掉，可想而知这样执行效率会有多底。

```java
    public static String s1() {
        String result = "";

        result += "A";
        result += "B";
        result += "C";

        return result;
    }

    public static String s2() {
        String result = "";
        result = "A" + "B" + "C";

        return result;
    }

    public static String s3() {
        StringBuilder result = new StringBuilder();
        result.append("A").append("B").append("C");
        return result.toString();
    }
```

性能上：s2 > s3 > s1

```
public static java.lang.String s1();
  Code:
   0:   ldc       #2; //String
   2:   astore_0
   3:   new       #3; //class java/lang/StringBuilder
   6:   dup
   7:   invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
   10:  aload_0
   11:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   14:  ldc       #6; //String A
   16:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   19:  invokevirtual   #7; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   22:  astore_0
   23:  new       #3; //class java/lang/StringBuilder
   26:  dup
   27:  invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
   30:  aload_0
   31:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   34:  ldc       #8; //String B
   36:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   39:  invokevirtual   #7; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   42:  astore_0
   43:  new       #3; //class java/lang/StringBuilder
   46:  dup
   47:  invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
   50:  aload_0
   51:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   54:  ldc       #9; //String C
   56:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   59:  invokevirtual   #7; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   62:  astore_0
   63:  aload_0
   64:  areturn
```

用javap –c xxx.class反编译

```
public static java.lang.String s2();
  Code:
   0:   ldc        #2; //String
   2:   astore_0
   3:   ldc        #10; //String ABC
   5:   astore_0
   6:   aload_0
   7:   areturn

public static java.lang.String s3();
  Code:
   0:   new        #3; //class java/lang/StringBuilder
   3:   dup
   4:   invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
   7:   astore_0
   8:   aload_0
   9:   ldc        #6; //String A
  11:   invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
  14:   ldc        #8; //String B
  16:   invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
  19:   ldc        #9; //String C
  21:   invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
  24:   pop
  25:   aload_0
  26:   invokevirtual   #7; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
  29:   areturn
```

String VS. StringBuilder VS. StringBuffer

- *2*．StringBuilder：线程非安全的; StringBuffer：线程安全的

- 当我们在字符串缓冲去被多个线程使用是，JVM不能保证StringBuilder的操作是安全的，虽然他的速度最快，但是可以保证StringBuffer是可以正确操作的。因为StringBuffer修改缓冲区的方式是同步的。

- 当然大多数情况下就是我们是在单线程下进行的操作，所以大多数情况下是建议用StringBuilder而不用StringBuffer的，就是速度的原因。

String VS. StringBuilder VS. StringBuffer

- 对于三者使用的总结：
  - 1.如果要操作少量的数据用＝String
  - 2.单线程操作字符串缓冲区下操作大量数据＝StringBuilder
  - 3.多线程操作字符串缓冲区下操作大量数据＝StringBuffer

Regular Expressions

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

Matching Strings

"Java".matches("Java");

"Java".equals("Java");

"Java is fun".matches("Java.*")

"Java is cool".matches("Java.*")

"Java is powerful".matches("Java.*")

Regular Expression Syntax

Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\w]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
p*	zero or more occurrences of pattern p	aaaabb matches "a*bb" ababab matches "(ab)*"
p+	one or more occurrences of pattern p	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern p	Java matches "J?Java" Java matches "J?ava"
p{n}	exactly n occurrences of pattern p	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern p	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"

Replacing and Splitting Strings

java.lang.String	
+matches(regex: String): boolean	Returns true if this string matches the pattern.
+replaceAll(regex: String, replacement: String): String	Returns a new string that replaces all matching substrings with the replacement.
+replaceFirst(regex: String, replacement: String): String	Returns a new string that replaces the first matching substring with the replacement.
+split(regex: String): String[]	Returns an array of strings consisting of the substrings split by the matches.

Examples

String s = "Java Java Java".replaceAll("v\\w", "wi") ;

String s = "Java Java Java".replaceFirst("v\\w", "wi") ;

String[] s = "Java1HTML2Perl".split("\\d");

枚举类(Enum)

- Java 中的枚举类型采用关键字enum 来定义，从jdk1.5才有的新类型，所有的枚举类型都是继承自Enum 类型。

通常定义常量方法：用public static final

```java
package com.csdn.myEnum;

public class Light {

    /* 红灯 */

    public final static int RED =1;

    /* 绿灯 */

    public final static int GREEN =3;

    /* 黄灯 */

    public final static int YELLOW =2;

}
```

枚举类型定义常量方法

```java
public enum Light {

    RED , GREEN , YELLOW ;

}
```

- 表面上看，枚举类型与其他语言的没什么两样，但实际上Java的枚举类型是功能十分齐全的类，功能比其他语言的对等物要强大的多，Java的枚举本质上是int值。

- 它通过公有的静态final域为每个枚举常量导出实例的类。

- 由于没有可访问的构造器，枚举类型是真正的final。枚举类型是实例受控的，是单例（Singleton ）的泛型化。

- 提供编译时的类型安全。若声明参数的类型为Light，则传到该参数上的任何非null对象引用一定属于Light的三个值之一。

□ >javap Light

```
警告: 二进制文件Light包含test.Light
Compiled from "Light.java"
public final class test.Light extends java.lang.Enum<test.Light> {
  public static final test.Light RED;
  public static final test.Light GREEN;
  public static final test.Light YELLOW;
  static {};
  public static test.Light[] values();
  public static test.Light valueOf(java.lang.String);
}
```

java编译器在对enum关键字进行处理时，实际上是将enum转换成为了java.lang.Enum类的一个子类来完成，而这个子类中含有values()静态方法。

enum类型是无法被继承的，它是final class

用jad反编译Light.class

```java
public final class Light extends Enum
{
    private Light(String s, int i)
    {
        super(s, i);
    }
    public static Light[] values()
    {
        Light alight[];
        int i;
        Light alight1[];
        System.arraycopy(alight = ENUM$VALUES, 0, alight1 = new Light[i = alight.length], 0, i);
        return alight1;
    }
    public static Light valueOf(String s)
    {
        return (Light)Enum.valueOf(test/Light, s);
    }
    public static final Light RED;
    public static final Light GREEN;
    public static final Light YELLOW;
    private static final Light ENUM$VALUES[];
    static
    {
        RED = new Light("RED", 0);
        GREEN = new Light("GREEN", 1);
        YELLOW = new Light("YELLOW", 2);
        ENUM$VALUES = (new Light[] {
            RED, GREEN, YELLOW
        });
    }
}
```

Jad下载：https://varaneckas.com/jad/

```
 * @param <E> The enum type subclass
 * @author  Josh Bloch
 * @author  Neal Gafter
 * @see     Class#getEnumConstants()
 * @see     java.util.EnumSet
 * @see     java.util.EnumMap
 * @since   1.5
 */
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
```

- ⊿ Ⓖ^A Enum<E extends Enum<E>>
  - ▫ ^F name : String
  - ● ^F name() : String
  - ▫ ^F ordinal : int
  - ● ^F ordinal() : int
  - ◇ ^C Enum(String, int)
  - ● toString() : String
  - ● equals(Object) : boolean
  - ● hashCode() : int
  - ◇ clone() : Object
  - ● compareTo(E) : int
  - ● ^F getDeclaringClass() : Class<E>
  - ● ^S valueOf(Class<T>, String) <T extends Enum<T>> : T
  - ◇ finalize() : void
  - ▪ readObject(ObjectInputStream) : void
  - ▪ readObjectNoData() : void

```java
enum EnumTry {
    MON, TUE, WED, THU, FRI;
    public static void main(String[] args)  {
    for (EnumTry e : EnumTry.values()) {

System.out.println(e +":"+ e.toString() +":"+ e.ordinal() +":"+ e.name());
}}}
```

- enum还有一个oridinal的方法，这个方法返回枚举值在枚举类种的顺序，这个顺序根据枚举值声明的顺序而定（从0开始）

```java
System.out.println("ordinal: " + Light.RED.ordinal());
```

- 这里Light.RED.ordinal()返回0。

注：在实际应用中，最好用实数域来代替序数。

```java
//WRONG
public enum Fruit{
    APPLE, PEAR, ORANGE;
    public int numberOfFruit(){
        return ordinal() + 1;
    }
}


//RIGHT
public enum Fruit{
    APPLE(1), PEAR(2), ORANGE(3);
    private final int number;
    Fruit(int num) {number  = num;}
    public int numberOfFruit(){
        return number;
    }
}
```

□ enum不支持clone，确保单例模式

```java
/**
 * Throws CloneNotSupportedException.  This guarantees that enums
 * are never cloned, which is necessary to preserve their "singleton"
 * status.
 *
 * @return (never returns)
 */
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

enum实例按ordinal进行排序

```java
/**
 * Compares this enum with the specified object for order.  Returns a
 * negative integer, zero, or a positive integer as this object is less
 * than, equal to, or greater than the specified object.
 *
 * Enum constants are only comparable to other enum constants of the
 * same enum type.  The natural order implemented by this
 * method is the order in which the constants are declared.
 */
public final int compareTo(E o) {
    Enum<?> other = (Enum<?>)o;
    Enum<E> self = this;
    if (self.getClass() != other.getClass() && // optimization
        self.getDeclaringClass() != other.getDeclaringClass())
        throw new ClassCastException();
    return self.ordinal - other.ordinal;
}
```

❑ 可以用valueOf得到枚举实例

```java
public static <T extends Enum<T>> T valueOf(Class<T> enumType,
                                             String name) {
    T result = enumType.enumConstantDirectory().get(name);
    if (result != null)
        return result;
    if (name == null)
        throw new NullPointerException("Name is null");
    throw new IllegalArgumentException(
        "No enum constant " + enumType.getCanonicalName() + "." + name);
}
```

```java
public class TestLight {

    public static void main(String[] args) {
        Light l = Light.valueOf("RED");
        System.out.println("selected: " + l);
    }

}
```

可以用toString将枚举转化为可打印的字符串

枚举类型支持switch

```java
public static void main(String[] args) {
    Light l = Light.valueOf("RED");
    System.out.println("selected: " + l);
    System.out.println("ordinal: " + Light.RED.ordinal());
    switch(l){
    case RED: System.out.println("red!"); break;
    case GREEN: System.out.println("green!"); break;
    case YELLOW: System.out.println("yellow!"); break;
    }
}
```

- enum类型可以关联不同的数据，也可以添加任意的方法和域，来增强枚举类型。

```java
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6), VENUS(4.869e+24, 6.0518e6), EARTH(5.976e+24,
            6.37814e6), MARS(6.421e+23, 3.3972e6), JUPITER(1.9e+27, 7.1492e7), SATURN(
            5.688e+26, 6.0268e7), URANUS(8.686e+25, 2.5559e7), NEPTUNE(
            1.024e+26, 2.4746e7), PLUTO(1.27e+22, 1.137e6);

    private final double mass; // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() {
        return mass;
    }
    private double radius() {
        return radius;
    }
    // universal gravitational constant (m^3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

□ 枚举类型有一个静态的values方法，可以按照声明顺序返回它的值数组。

```java
public class TestPlanet {
    public static void main(String[] args) {
        double earthWeight = 100;
        double mass = earthWeight/Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Your weight on %s is %f%n",
                              p, p.surfaceWeight(mass));
    }
}
```

```
Your weight on MERCURY is 37.775762
Your weight on VENUS is 90.499910
Your weight on EARTH is 100.000000
Your weight on MARS is 37.873718
Your weight on JUPITER is 253.055753
Your weight on SATURN is 106.601554
Your weight on URANUS is 90.512720
Your weight on NEPTUNE is 113.832807
Your weight on PLUTO is 6.687446
```

- 在枚举类中，可以将不同的行为与每个常量关联起来。例如，对于Operation中的PLUS, MINUS, TIMES和DIVIDE，有不同的操作。

```java
public enum OperationUseSwitch {
    PLUS, MINUS, TIMES, DIVIDE;

    double apply(double x, double y) {
        switch (this) {
            case PLUS:
                return x + y;
            case MINUS:
                return x + y;
            case TIMES:
                return x + y;
            case DIVIDE:
                return x + y;
        }
        // 如果this不属于上面四种操作符，抛出异常
        throw new AssertionError("Unknown operation: " + this);
    }
}
```

若要实现，不同的常量有不同的行为，利用switch可以实现：

```java
public enum OperationUseSwitch {
    PLUS, MINUS, TIMES, DIVIDE;

    double apply(double x, double y) {
        switch (this) {
            case PLUS:
                return x + y;
            case MINUS:
                return x + y;
            case TIMES:
                return x + y;
            case DIVIDE:
                return x + y;
        }
        // 如果this不属于上面四种操作符，抛出异常
        throw new AssertionError("Unknown operation: " + this);
    }
}
```

首先是我们不得不在最后抛出异常或者在switch里加上default，不然无法编译通过，但是很明显，程序的分支是不会进入异常或者default的。

其次，这段代码非常脆弱，如果我们添加了新的操作类型，却忘了在switch里添加相应的处理逻辑，执行新的运算操作时，就会出现问题。

- Java枚举提供了一种功能，叫做*特定于常量的方法实现*。

```java
public enum Operation {
    PLUS {
        double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS {
        double apply(double x, double y) {
            return x - y;
        }
    },
    TIMES {
        double apply(double x, double y) {
            return x * y;
        }
    },
    DIVIDE {
        double apply(double x, double y) {
            return x / y;
        }
    };

    abstract double apply(double x, double y);
}
```

不会出现添加新操作符后忘记添加对应的处理逻辑的情况了，
因为编译器就会提示我们必须覆盖apply方法。

- *特定于常量的方法实现* 有一个缺点，那就是很难在枚举常量之间共享代码。
- 以星期X的枚举为例，周一到周五是工作日，执行一种逻辑，周六周日，休息日，执行另一种逻辑。

```java
public enum DayUseAbstractMethod {
    MONDAY {
        @Override
        void apply() {
            dealWithWeekDays();//伪代码
        }
    },
    TUESDAY {
        @Override
        void apply() {
            dealWithWeekDays();//伪代码
        }
    },
    WEDNESDAY {
        @Override
        void apply() {
            dealWithWeekDays();//伪代码
        }
    },
    THURSDAY {
        @Override
        void apply() {
            dealWithWeekDays();//伪代码
        }
    },
```

```java
    FRIDAY {
        @Override
        void apply() {
            dealWithWeekDays();//伪代码
        }
    },
    SATURDAY {
        @Override
        void apply() {
            dealWithWeekEnds();//伪代码
        }
    },
    SUNDAY {
        @Override
        void apply() {
            dealWithWeekEnds();//伪代码
        }
    };

    abstract void apply();
}
```

可以借助于*策略枚举*，将处理委托给另一个枚举类型。

```java
public enum Day {
    MONDAY(DayType.WEEKDAY), TUESDAY(DayType.WEEKDAY), WEDNESDAY(
            DayType.WEEKDAY), THURSDAY(DayType.WEEKDAY), FRIDAY(DayType.WEEKDAY), SATURDAY(
            DayType.WEEKDAY), SUNDAY(DayType.WEEKDAY);
    private final DayType dayType;

    Day(DayType daytype) {
        this.dayType = daytype;
    }

    void apply() {
        dayType.apply();
    }

    private enum DayType {
        WEEKDAY {
            @Override
            void apply() {
                System.out.println("hi, weekday");
            }
        },
        WEEKEND {
            @Override
            void apply() {
                System.out.println("hi, weekend");
            }
        };
        abstract void apply();
    }
}
```