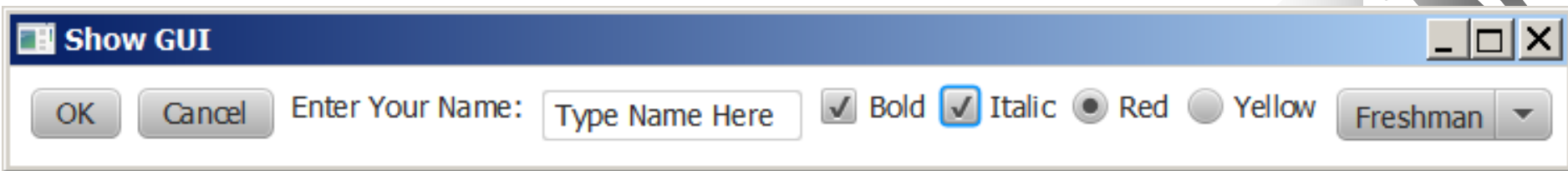


# 09 Objects and Classes



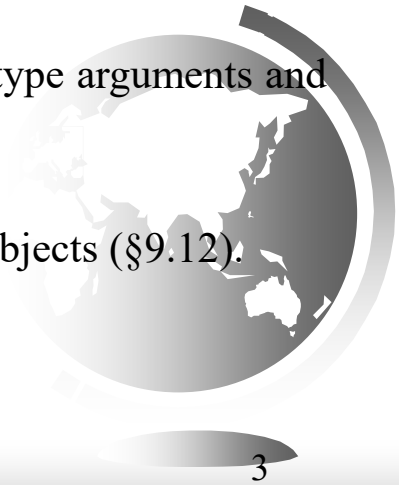
# Motivations

After learning the preceding chapters, you are capable of solving many programming problems **using selections, loops, methods, and arrays**. **However**, these Java features are not sufficient for developing graphical user interfaces and large scale software systems. Suppose you want to develop a graphical user interface as shown below. How do you program it?



# Objectives

- ❑ To describe objects and classes, and use classes to model objects (§9.2).
- ❑ To use UML graphical notation to describe classes and objects (§9.2).
- ❑ To demonstrate how to define classes and create objects (§9.3).
- ❑ To create objects using constructors (§9.4).
- ❑ To access objects via object reference variables (§9.5).
- ❑ To define a reference variable using a reference type (§9.5.1).
- ❑ To access an object's data and methods using the object member access operator (.) (§9.5.2).
- ❑ To define data fields of reference types and assign default values for an object's data fields (§9.5.3).
- ❑ To distinguish between object reference variables and primitive data type variables (§9.5.4).
- ❑ To use the Java library classes **Date**, **Random**, and **Point2D** (§9.6).
- ❑ To distinguish between instance and static variables and methods (§9.7).
- ❑ To define private data fields with appropriate **get** and **set** methods (§9.8).
- ❑ To encapsulate data fields to make classes easy to maintain (§9.9).
- ❑ To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§9.10).
- ❑ To store and process objects in arrays (§9.11).
- ❑ To create immutable objects from immutable classes to protect the contents of objects (§9.12).
- ❑ To determine the scope of variables in the context of a class (§9.13).
- ❑ To use the keyword **this** to refer to the calling object itself (§9.14).



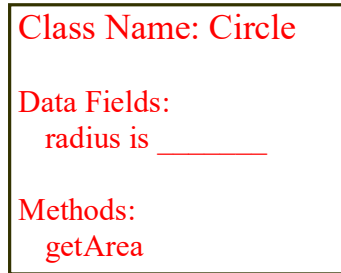
# OO Programming Concepts

Object-oriented programming (OOP) involves programming **using objects**. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.

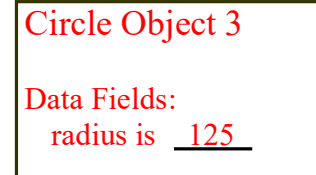
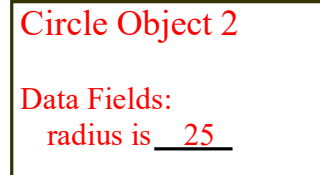
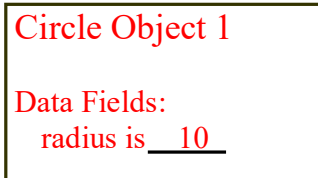
An object has a unique identity, state, and behaviors. **The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values. The *behavior* of an object is defined by a set of methods.**



# Objects



← A class template



← Three objects of the Circle class

An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.



# Classes

*Classes* are constructs that define objects of the **same type**. A Java class uses variables to define data fields and methods to define behaviors.

Additionally, a class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class.



# Classes

```
class Circle {  
    /** The radius of this circle */  
    double radius = 1.0;  
  
    /** Construct a circle object */  
    Circle() {  
    }  
  
    /** Construct a circle object */  
    Circle(double newRadius) {  
        radius = newRadius;  
    }  
  
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * 3.14159;  
    }  
}
```

← Data field

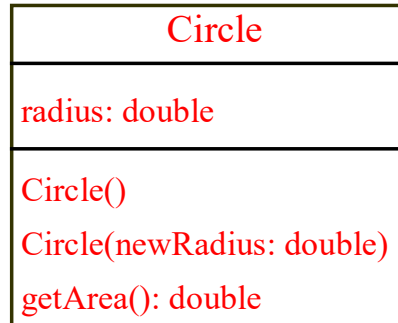
← Constructors

← Method



# UML Class Diagram

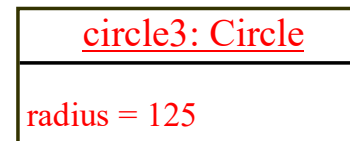
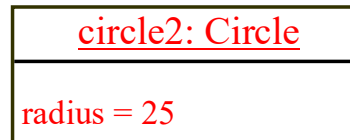
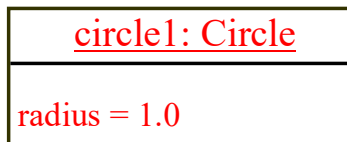
UML Class Diagram



← Class name

← Data fields

← Constructors and methods



← UML notation for objects





# Constructors

Constructors are a special kind of methods that are invoked to construct objects.

```
Circle() {  
}
```

```
Circle(double newRadius) {  
    radius = newRadius;  
}
```



# Constructors, cont.

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.



# Creating Objects Using Constructors

```
new ClassName() ;
```

Example:

```
new Circle() ;
```

```
new Circle(5.0) ;
```



# 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.***



# Default Constructor

```
class Point{  
    int x;  
    int y;  
    Point(int x1, int y1){  
        x = x1;  
        y = y1;  
    }  
    .....  
}
```

```
class Test{  
    public static void main(String[] args){  
        Point p = new Point();  
    }  
    .....  
}
```



# Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Example:

```
Circle myCircle;
```



# Declaring/Creating Objects in a Single Step

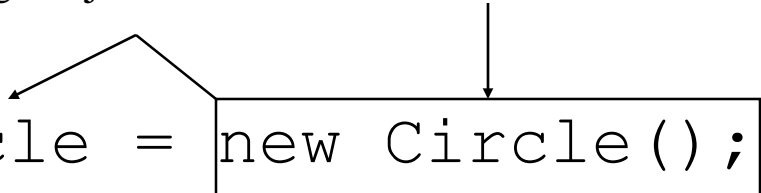
```
ClassName objectRefVar = new ClassName();
```

Assign object reference

Create an object

Example:

```
Circle myCircle = new Circle();
```



# Accessing Object's Members

- ❑ Referencing the object's data:

`objectRefVar.data`

*e.g.*, `myCircle.radius`

- ❑ Invoking the object's method:

`objectRefVar.methodName (arguments)`

*e.g.*, `myCircle.getArea()`





# Trace Code

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Declare myCircle

myCircle

no value



# Trace Code, cont.

**Circle myCircle = new Circle(5.0);**

myCircle no value

**Circle yourCircle = new Circle();**

**yourCircle.radius = 100;**

: Circle  
radius: 5.0

Create a circle



# Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Assign object reference  
to myCircle

myCircle reference value

: Circle

radius: 5.0



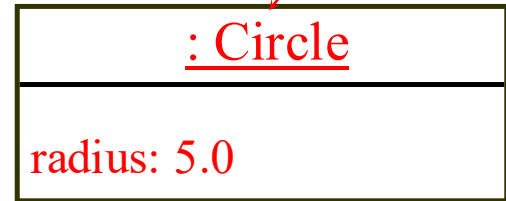
# Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle no value

Declare yourCircle

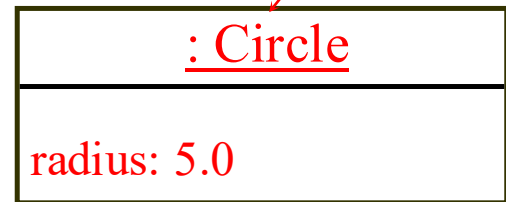
# Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

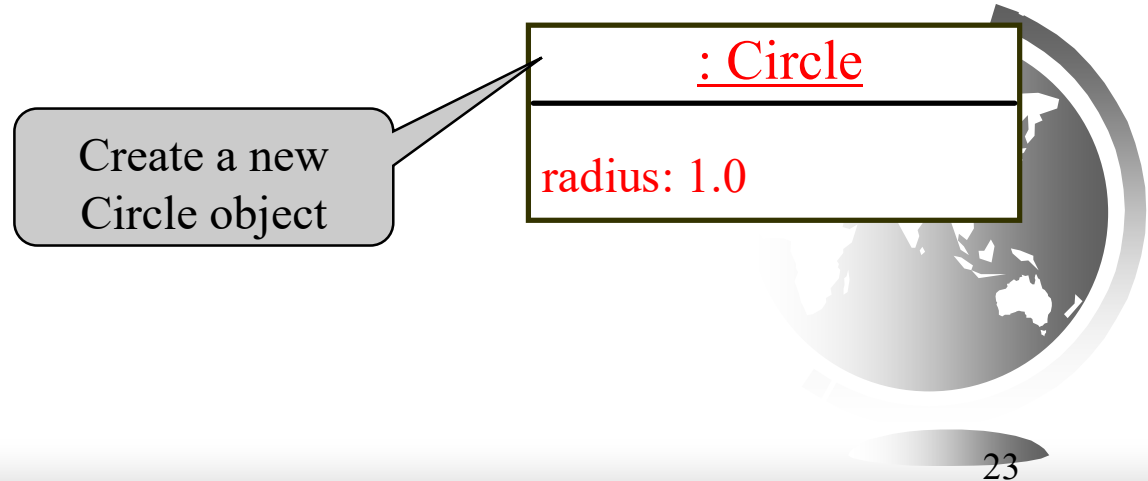
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle no value



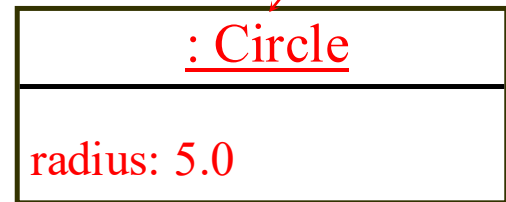
# Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

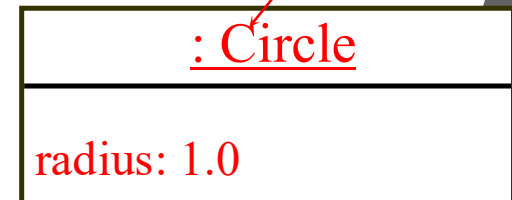
```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **reference value**

Assign object reference  
to yourCircle



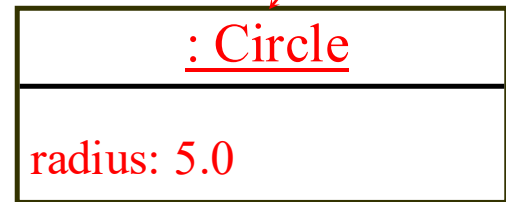
# Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

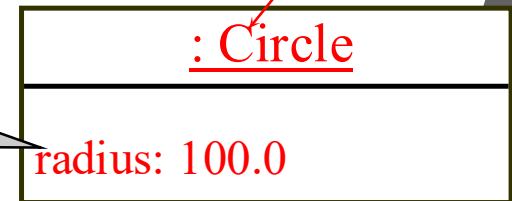
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in  
yourCircle

# Caution

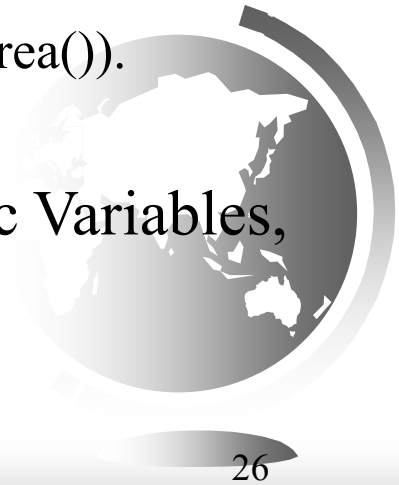
Recall that you use

`Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`)

to invoke a method in the `Math` class. Can you invoke `getArea()` using `SimpleCircle.getArea()`? The answer is no. All the methods used before this chapter are static methods, which are defined using the `static` keyword. However, `getArea()` is non-static. It must be invoked from an object using

`objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`).

More explanations will be given in the section on “Static Variables, Constants, and Methods.”





# Reference Data Fields

The data fields can be of reference types. For example, the following Student class contains a data field name of the String type.

```
public class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default value false  
    char gender; // c has default value '\u0000'  
}
```



# The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.



# Default Value for a Data Field

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```



# Example

Java assigns **no default value to a local variable** inside a method.

```
public class Test {  
    public static void main(String[] args) {  
        int x; // x has no default value  
        String y; // y has no default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```

Compile error: variable not  
initialized



# Modify Default Value for a Data Field

```
public class Test {  
    int x = 1;  
    int y;  
    {  
        y = 2;  
    }  
    .....  
}
```

```
public class Test {  
    static int STATIC_ONE = 1;  
    static int STATIC_TWO;  
    {  
        STATIC_TWO = 2;  
    }  
    .....  
}
```



# Java VS. C++

```
// java数据成员的默认初始化
public class InitialValues{
    boolean t;    //false
    char c;       //[]
    short s;      //0
    byte b;       //0
    int i;        //0
    long l;       //0
    float f;      //0.0
    double d;     //0.0
}
```

- 编译器会为这些数据成员进行**默认初始化**，实际上是把刚分配的对象内存都置零。
- 在对象里定义一个引用，且不将其初始化时，默认初始化为null。这种默认初始化的实现是，在创建（new）一个对象时，在堆上对对象分配足够的空间之后，这块存储空间会被清零，这样就自动把基本类型的数据成员都设置成了默认值。
- 默认初始化动作之后，才执行**指定初始化**。也就是说下面的i经历过被初始化为0后，再赋值为999的过程。

指定初始化  
C++不支持这种操作

```
public class InitialValues{
    int i = 999;
}
```



# Java VS. C++

- java也可以使用构造函数来进行初始化，但构造函数的初始化无法阻止指定初始化和默认初始化的进行，而且总是在它们之后，才会执行构造函数初始化。总结起来说，java中数据成员的初始化过程是：
  - ① 先默认初始化
  - ② 进行定义处的初始化（指定初始化）
  - ③ 构造函数初始化



# Java VS. C++

- C++禁止在定义数据成员时就进行指定初始化，而且C++也没有默认初始化。

```
class Test {  
public:  
    int i;  
    double b;  
    char ch;  
};  
int main()  
{  
    Test *t = new Test();  
    cout << t->b; //输出0  
    cout << t->i; //输出0  
    cout << t->ch; //输出[]  
    return 0;  
}
```

为什么也输出0、[]等？

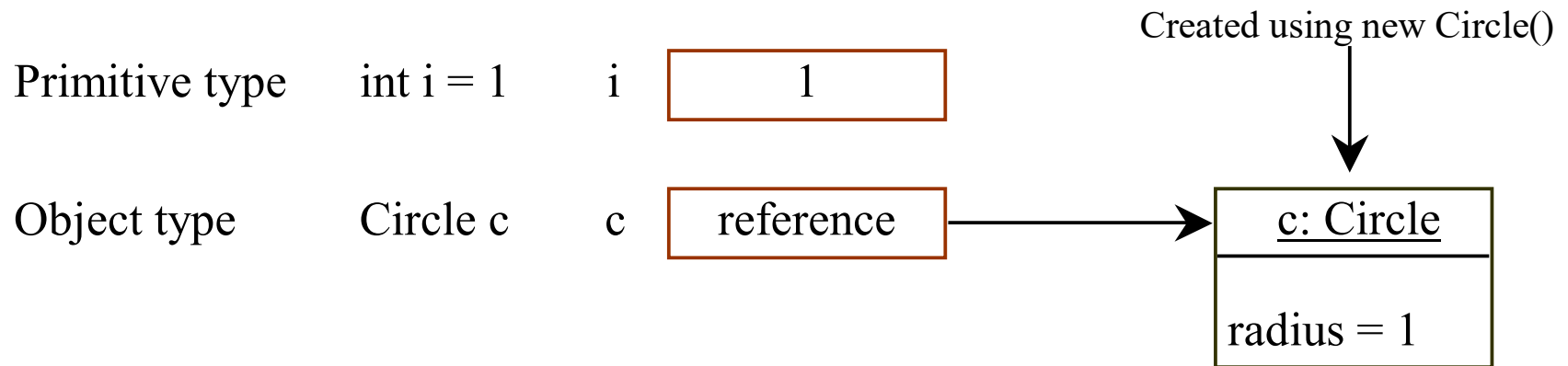
实际上是C++的默认构造函数进行的构造函数初始化。当类没有构造函数时，编译器会为类声明并实现一个默认构造函数，默认构造函数将数据成员初始化为默认值。所以C++数据成员的初始值，只能依赖：

- 成员初始化列表
- 构造函数





# Differences between Variables of Primitive Data Types and Object Types



# Copying Variables of Primitive Data Types and Object Types

Primitive type assignment  $i = j$

Before:

i 

1
---

j 

2
---

After:

i 

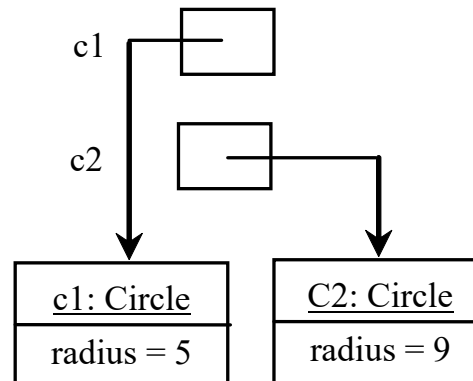
2
---

j 

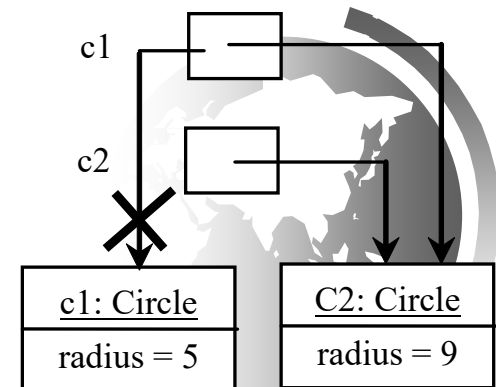
2
---

Object type assignment  $c1 = c2$

Before:



After:



# Garbage Collection

As shown in the previous figure, after the assignment statement  $c1 = c2$ ,  $c1$  points to the same object referenced by  $c2$ . The object previously referenced by  $c1$  is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.



# Garbage Collection, cont

TIP: If you know that an object is no longer needed, **you can explicitly assign null to a reference variable for the object.**

The JVM will automatically collect the space if the object is not referenced by any variable .



// Can you spot the "memory leak"?

```
public class Stack {  
    private Object[] elements;  
    private int size = 0;  
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
```

哪里有内存泄露？

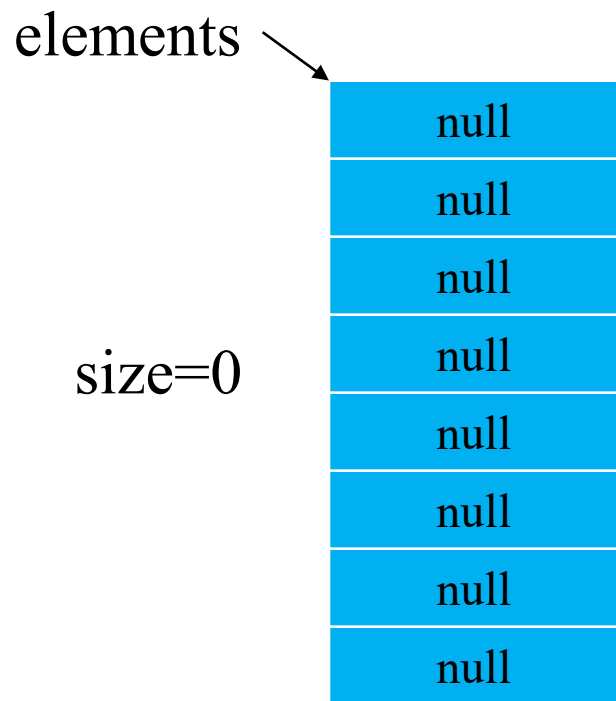
```
    public Stack() {  
        elements = new Object[DEFAULT_INITIAL_CAPACITY];  
    }
```

```
    public void push(Object e) {  
        ensureCapacity();  
        elements[size++] = e;  
    }
```

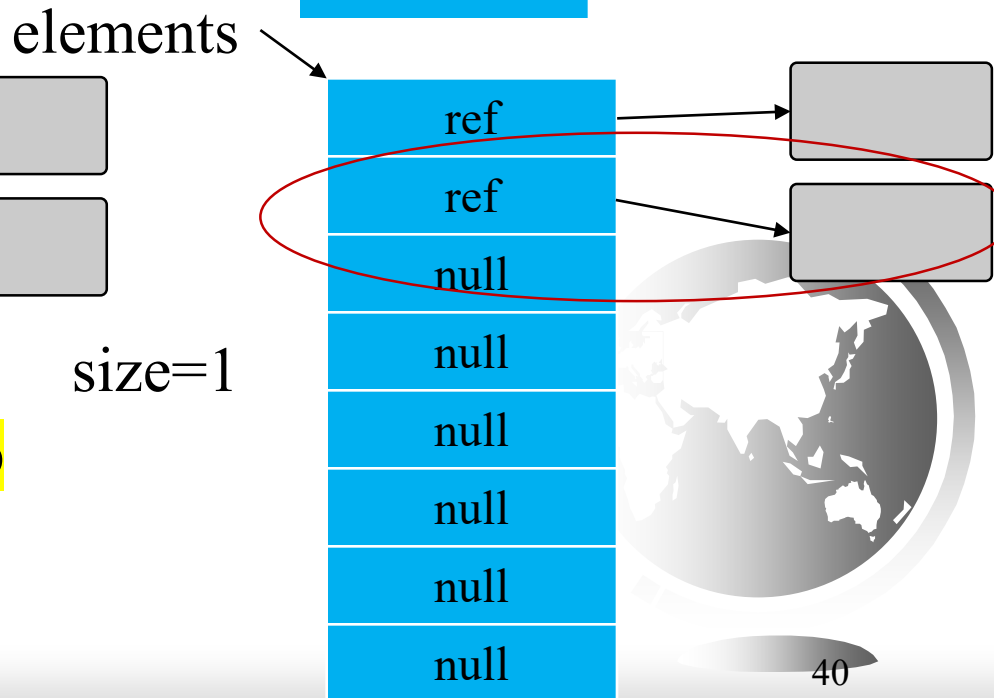
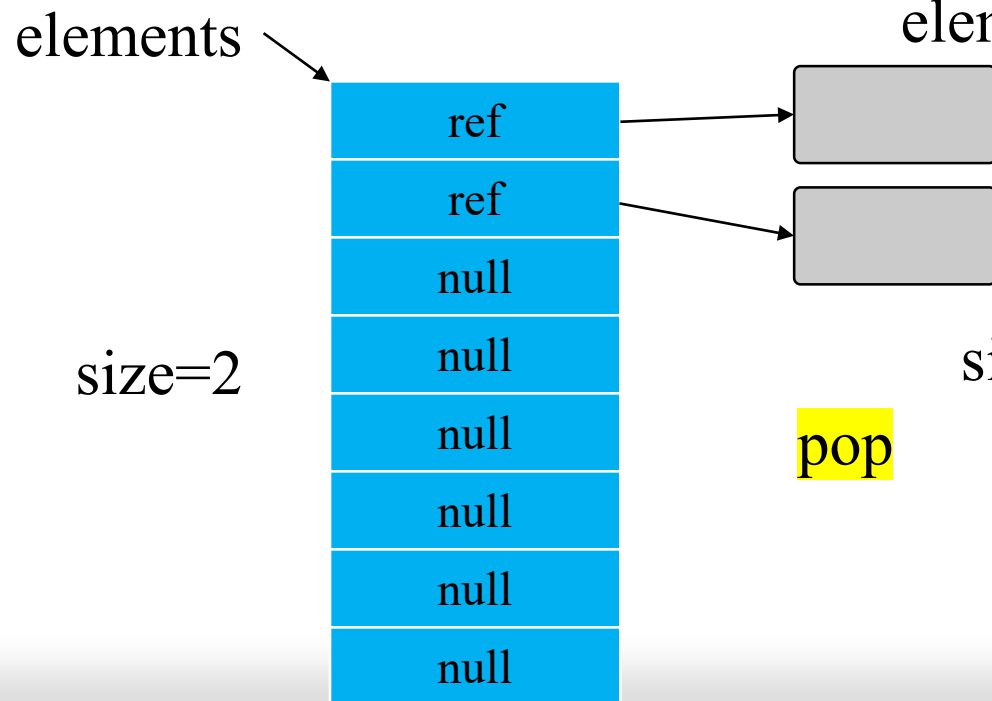
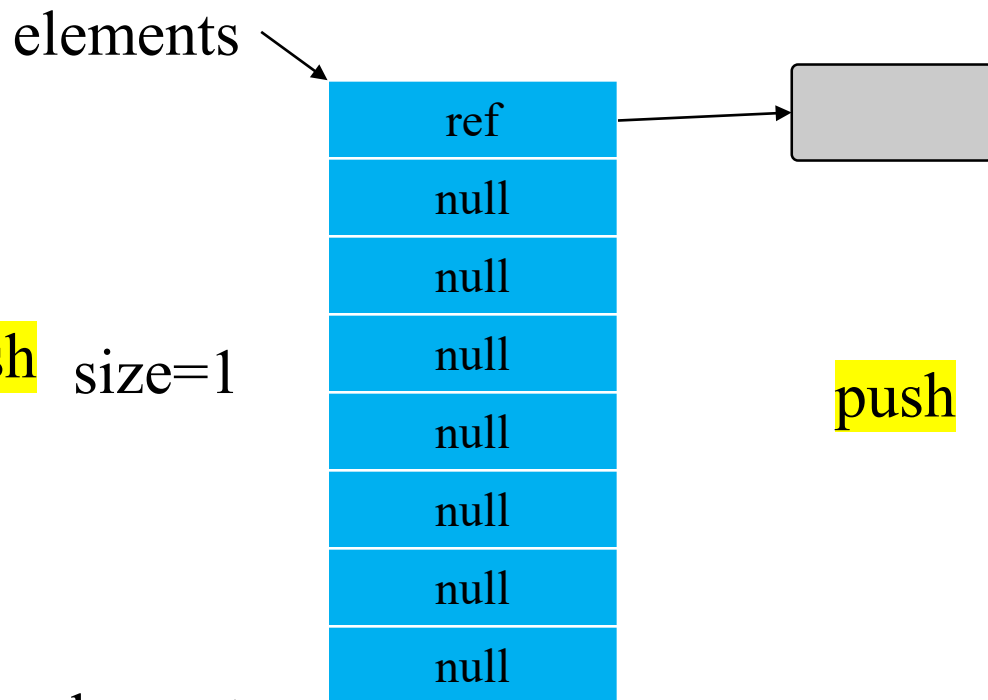
```
    public Object pop() {  
        if (size == 0)  
            throw new EmptyStackException();  
        return elements[--size];  
    }
```

```
    /**  
     * Ensure space for at least one more element, roughly  
     * doubling the capacity each time the array needs to grow.  
     */  
    private void ensureCapacity() {  
        if (elements.length == size)  
            elements = Arrays.copyOf(elements, 2 * size + 1);  
    }  
}
```





push



```
public  
class Stack<E> extends Vector<E> {
```

```
    public synchronized E pop() {  
        E      obj;  
        int    len = size();  
  
        obj = peek();  
        removeElementAt(len - 1);  
  
        return obj;  
    }
```

```
    public synchronized E peek() {  
        int      len = size();  
  
        if (len == 0)  
            throw new EmptyStackException();  
        return elementAt(len - 1);  
    }
```



```
public synchronized void removeElementAt(int index) {  
    modCount++;  
    if (index >= elementCount) {  
        throw new ArrayIndexOutOfBoundsException(index + " >= " +  
                                                    elementCount);  
    }  
    else if (index < 0) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
    int j = elementCount - index - 1;  
    if (j > 0) {  
        System.arraycopy(elementData, index + 1, elementData, index, j);  
    }  
    elementCount--;  
    elementData[elementCount] = null; /* to let gc do its work */  
}
```





# Garbage Collection (cont.)

- 一般而言，只要类自己管理内存，程序员就应该警惕内存泄漏问题。
- 如前面Stack类自己管理内存: elements数组

```
private Object[] elements;  
private int size = 0;  
private static final int DEFAULT_INITIAL_CAPACITY = 16;
```

这里elements是存储池，size大小的内存则是活动区域，而剩余部分则是free的。但，垃圾回收器并不知道这个。所以，对于垃圾回收器来说，elements中的所有对象引用都是同等有效的。而只有程序员才知道非活动区域是不重要的。



# The Date Class

Java provides a **system-independent** encapsulation of date and time in the java.util.Date class. You can use the Date class to create an instance for the current date and time and use its toString method to return the date and time as a string.

The + sign indicates  
public modifier



java.util.Date	
+Date()	
+Date(elapseTime: long)	
+toString(): String	
+getTime(): long	
+setTime(elapseTime: long): void	

Constructs a Date object for the current time.

Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.

Returns a string representing the date and time.

Returns the number of milliseconds since January 1, 1970, GMT.

Sets a new elapse time in the object.



# The Date Class Example

For example, the following code

```
java.util.Date date = new java.util.Date();  
System.out.println(date.toString());
```

displays a string like Sun Mar 09 13:50:19  
EST 2003.



# The Random Class

You have used Math.random() to obtain a random double value between 0.0 and 1.0 (excluding 1.0). **A more useful random number generator is provided in the java.util.Random class.**

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

# The Random Class Example

If two Random objects have the same seed, they will generate **identical sequences of numbers**. For example, the following code creates two Random objects with the same seed 3.

```
Random random1 = new Random(3);  
System.out.print("From random1: ");  
for (int i = 0; i < 10; i++)  
    System.out.print(random1.nextInt(1000) + " ");  
Random random2 = new Random(3);  
System.out.print("\nFrom random2: ");  
for (int i = 0; i < 10; i++)  
    System.out.print(random2.nextInt(1000) + " ");
```

From random1: 734 660 210 581 128 202 549 564 459 961

From random2: 734 660 210 581 128 202 549 564 459 961



# Instance Variables, and Methods

Instance variables belong to a specific instance.

Instance methods are invoked by an instance of the class.



# Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object.

Static constants are final variables shared by all the instances of the class.



# Static Variables, Constants, and Methods, cont.

To declare static variables, constants, and methods,  
use the **static** modifier.





# Static Variables, Constants, and Methods, cont.

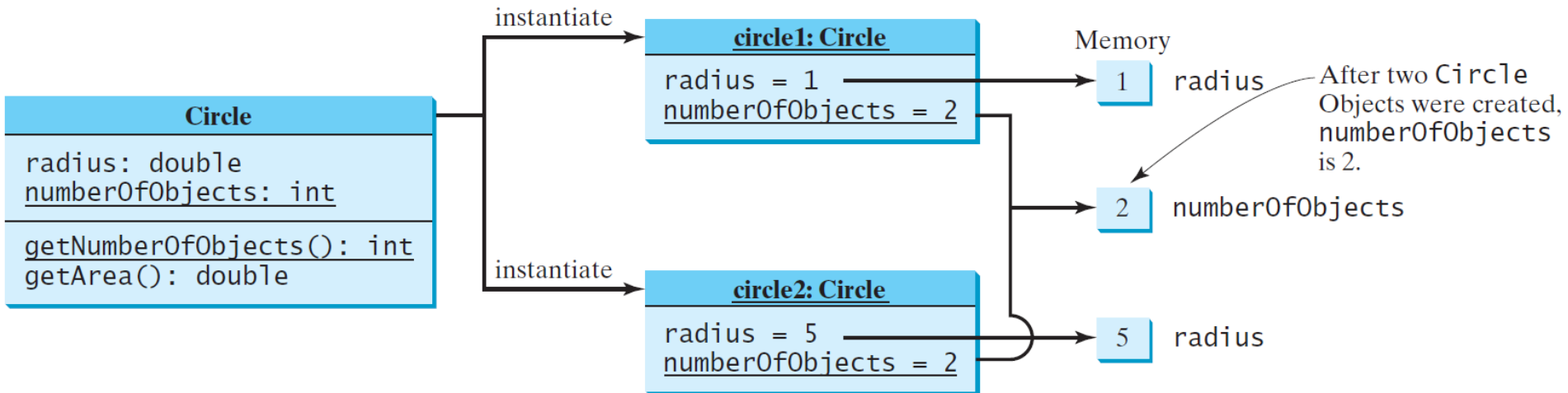
- Static Methods只能访问Static Variables，不能访问Instance variables; 可以调用其他的Static methods，但不能调用instance methods.
- Instance methods既能访问instance variables，也能访问static variables; 既能调用instance methods，也能调用static methods.



# Static Variables, Constants, and Methods, cont.

UML Notation:

underline: static variables or methods



# Example of Using Instance and Class Variables and Method

Objective: Demonstrate the roles of instance and class variables and their uses. This example adds a class variable `numberOfObjects` to track the number of `Circle` objects created.



[CircleWithStaticMembers](#)



[TestCircleWithStaticMembers](#)

Run



```
public class CircleWithStaticMembers {
    /** The radius of the circle */
    double radius;

    /** The number of the objects created */
    static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    CircleWithStaticMembers() {
        radius = 1.0;
        numberOfObjects++;
    }

    /** Construct a circle with a specified radius */
    CircleWithStaticMembers(double newRadius) {
        radius = newRadius;
        numberOfObjects++;
    }

    /** Return numberOfObjects */
    static int getNumberOfObjects() {
        return numberOfObjects;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }
}
```

```
public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
        CircleWithStaticMembers.numberOfObjects);

    // Create c1
    CircleWithStaticMembers c1 = new CircleWithStaticMembers();

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
        ") and number of Circle objects (" +
        c1.numberOfObjects + ")");

    // Create c2
    CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
        ") and number of Circle objects (" +
        c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
        ") and number of Circle objects (" +
        c2.numberOfObjects + ")");
}
```



Problems Javadoc Declaration Console Call Hierarchy

<terminated> TestCircleWithStaticMembers [Java Application] C:\Prog

Before creating objects

The number of Circle objects is 0

After creating c1

c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1

c1: radius (9.0) and number of Circle objects (2)

c2: radius (5.0) and number of Circle objects (2)



# Visibility Modifiers and Accessor/Mutator Methods

**By default**, the class, variable, or method can be accessed by **any class in the same package**.

- ❑ `public`

The class, data, or method is visible to **any class in any package**.

- ❑ `private`

The data or methods can be accessed **only by the declaring class**.

The **get and set methods** are used to read and modify private properties.



```

package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}

```

```

package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}

```

```

package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}

```

```

package p1;

class C1 {
    ...
}

```

```

package p1;

public class C2 {
    can access C1
}

```

```

package p2;

public class C3 {
    cannot access C1;
    can access C2;
}

```

The **private** modifier restricts access to within a class, the **default** modifier restricts access to within a package, and the **public** modifier enables unrestricted access.



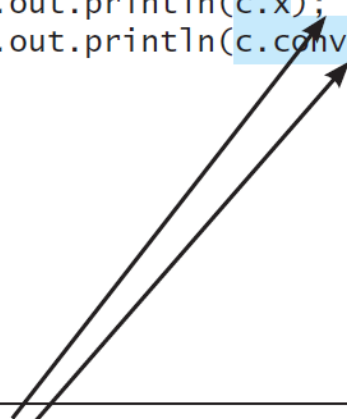
# NOTE

An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class C {  
    private boolean x;  
  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
  
    private int convert() {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is okay because object **c** is used inside the class **C**.

```
public class Test {  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
}
```



(b) This is wrong because **x** and **convert** are private in class **C**.

# Why Data Fields Should Be private?

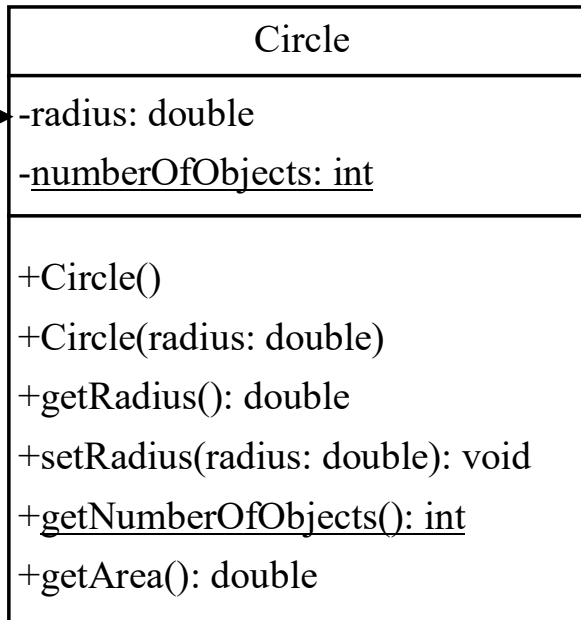
To protect data.

To make code easy to maintain.



# Example of Data Field Encapsulation

The - sign indicates  
private modifier



The radius of this circle (default: 1.0).

The number of circle objects created.

Constructs a default circle object.

Constructs a circle object with the specified radius.

Returns the radius of this circle.

Sets a new radius for this circle.

Returns the number of circle objects created.

Returns the area of this circle.

CircleWithPrivateDataFields

TestCircleWithPrivateDataFields

Run

```

public class CircleWithPrivateDataFields {
    /** The radius of the circle */
    private double radius = 1;
    /** The number of the objects created */
    private static int numberOfObjects = 0;
    /** Construct a circle with radius 1 */
    public CircleWithPrivateDataFields() {
        numberOfObjects++;
    }
    /** Construct a circle with a specified radius
    */
    public CircleWithPrivateDataFields(double
newRadius) {
        radius = newRadius;
        numberOfObjects++;
    }
    .....
}

```

```

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

/** Set a new radius */
public void setRadius(double
newRadius) {
    radius = (newRadius >= 0) ?
newRadius : 0;
}

/** Return numberOfObjects */
public static int getNumberOfObjects()
{
    return numberOfObjects;
}

/** Return the area of this circle */
public double getArea() {
    return radius * radius * Math.PI;
}

```

□ Constructor是否可以private?



# private Constructor

- 1. 不能创建类的实例，类只能被静态访问

```
105 public final class Math {  
106  
107     /**  
108      * Don't let anyone instantiate this class.  
109      */  
110     private Math() {}  
111
```

```
71 public class Arrays {  
72  
73     /**  
74      * The minimum array length below which a parallel sorting  
75      * algorithm will not further partition the sorting task. Using  
76      * smaller sizes typically results in memory contention across  
77      * tasks that makes parallel speedups unlikely.  
78      */  
79     private static final int MIN_ARRAY_SORT_GRAN = 1 << 13;  
80  
81     // Suppresses default constructor, ensuring non-instantiability.  
82     private Arrays() {}  
83
```



# private Constructor

- 2. 能创建类的实例，但只能被类的静态方法调用。常见场景：单例模式

```
46 public class Runtime {
47     private static Runtime currentRuntime = new Runtime();
48
49     /**
50      * Returns the runtime object associated with the current Java application.
51      * Most of the methods of class <code>Runtime</code> are instance
52      * methods and must be invoked with respect to the current runtime object.
53      *
54      * @return the <code>Runtime</code> object associated with the current
55      *         Java application.
56      */
57     public static Runtime getRuntime() {
58         return currentRuntime;
59     }
60
61     /** Don't let anyone else instantiate this class */
62     private Runtime() {}
63 }
```

# private Constructor

```
76  * @since 1.6
77  * @author Armin Chen
78  * @author George Zhang
79  */
80  public class Desktop {
81      ~~~
119      private DesktopPeer peer;
120
121      /**
122       * Suppresses default constructor for noninstantiability.
123       */
124      private Desktop() {
125          peer = Toolkit.getDefaultToolkit().createDesktopPeer(this);
126      }
127
141      public static synchronized Desktop getDesktop() {
142          if (GraphicsEnvironment.isHeadless()) throw new HeadlessException();
143          if (!Desktop.isDesktopSupported()) {
144              throw new UnsupportedOperationException("Desktop API is not " +
145                  "supported on the current platform");
146          }
147
148          sun.awt.AppContext context = sun.awt.AppContext.getAppContext();
149          Desktop desktop = (Desktop) context.get(Desktop.class);
150
151          if (desktop == null) {
152              desktop = new Desktop();
153              context.put(Desktop.class, desktop);
154          }
155
156          return desktop;
157      }
158  }
```





# private Constructor

- 3. 只能被其他构造函数调用，用于减少重复代码

```
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {


    private static final long serialVersionUID = -2851667679971038690L;

    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }

    public LinkedHashSet(int initialCapacity) {
        super(initialCapacity, .75f, true);
    }

    public LinkedHashSet() {
        super(16, .75f, true);
    }

    public LinkedHashSet(Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11), .75f, true);
        addAll(c);
    }
}
```



```
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
    map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
```



# 类和对象的生命周期

- 当程序运行的时候，第一次new创建一个类的对象，或通过类名访问静态变量或静态方法时，java会将类加载进内存，为这个类分配一块空间（其实是方法区），这块空间会包括类的定义、它的变量和方法信息，还有类的静态变量，并对静态变量赋值。
- 类加载进内存后，一般不会释放，直到程序结束。一般情况，类只会加载一次，所以静态变量在内存中只存在一份。
- 通过new创建一个对象时，对象产生，在内存中会存储这个对象的实例变量值。每new一次，都会产生一个变对象，会有一份独立的实例变量。



# Passing Objects to Methods

- ❑ **Passing by value** for primitive type value  
(the value is passed to the parameter)
- ❑ **Passing by value** for reference type value  
(the value is the reference to the object)



TestPassObject

Run

```

public static void main(String[] args) {
    // Create a Circle object with radius 1
    CircleWithPrivateDataFields myCircle =
        new CircleWithPrivateDataFields(1);

    // Print areas for radius 1, 2, 3, 4, and 5.
    int n = 5;
    printAreas(myCircle, n);

    // See myCircle.radius and times
    System.out.println("\n" + "Radius is " + myCircle.getRadius());
    System.out.println("n is " + n);
}

```

```

/** Print a table of areas for radius */

```

```

public static void printAreas(
    CircleWithPrivateDataFields c, int times) {
    System.out.println("Radius \t\tArea");
    while (times >= 1) {
        System.out.println(c.getRadius() + "\t\t" + c.getArea());
        c.setRadius(c.getRadius() + 1);
        times--;
    }
}

```

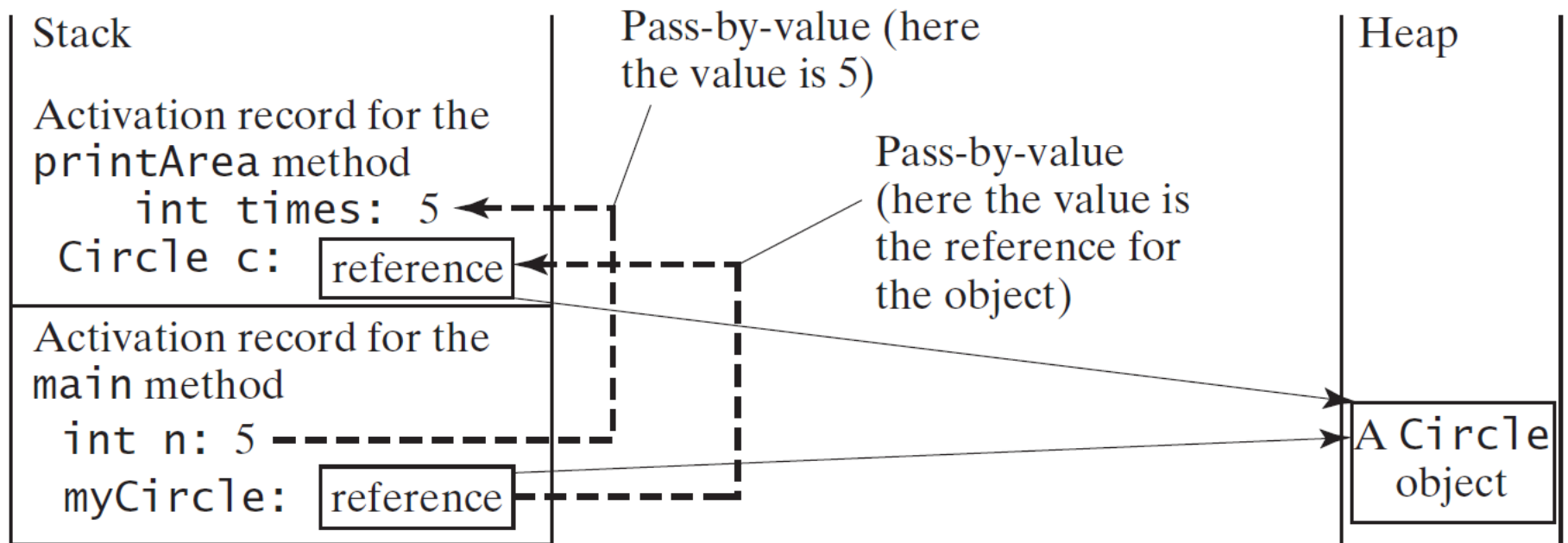
Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483

```

Radius is 6.0
n is 5

```

# Passing Objects to Methods, cont.



# NOTE

- 很多程序设计语言（特别是C++, Pascal) 提供了两种参数传递的方式：值传递和引用传递。
- 有些程序员会认为JAVA采用的是引用传递，实际上，这是不对的。
- 其实本质上，JAVA对象引用还是按值传递的。



```
public static void swap(Employee x, Employee y)
{
    Employee temp = x;
    x = y;
    y = temp;
}

Employee a = new Employee("alice",...);
Employee b = new Employee("bob",...);
swap(a,b);
```



```
22 class Employee {
23     private String name;
24     private double salary;
25
26     public Employee(String n, double s) {
27         name = n;
28         salary = s;
29     }
30
31     public String getName() {
32         return name;
33     }
34
35     public double getSalary() {
36         return salary;
37     }
38
39     public void raiseSalary(double bypercent) {
40         double raise = salary * bypercent / 100;
41         salary += raise;
42     }
43 }
```



```

3 public class ParameterTest {
4     public static void main(String[] args) {
5         Employee a = new Employee("Alice", 10000);
6         Employee b = new Employee("Bob", 30000);
7         System.out.println("Before: a=" + a.getName());
8         System.out.println("Before: b=" + b.getName());
9         swap(a, b);
10        System.out.println("After: a=" + a.getName());
11        System.out.println("After: b=" + b.getName());
12    }
13
14    public static void swap(Employee x, Employee y) {
15        Employee temp = x;
16        x = y;
17        y = temp;
18        System.out.println("End of the method :x=" + x.getName());
19        System.out.println("End of the method :y=" + y.getName());
20    }
21 }

```

Problems Javadoc Declaration Console

<terminated> ParameterTest [Java Application] C:\Program Files\Java\jdk1.8.0\_141\jre\bin\javaw

```

Before: a=Alice
Before: b=Bob
End of the method :x=Bob
End of the method :y=Alice
After: a=Alice
After: b=Bob

```

拷贝的引用

alice =

bob =

x =

y =

Employee

Employee

交换的引用

# Array of Objects

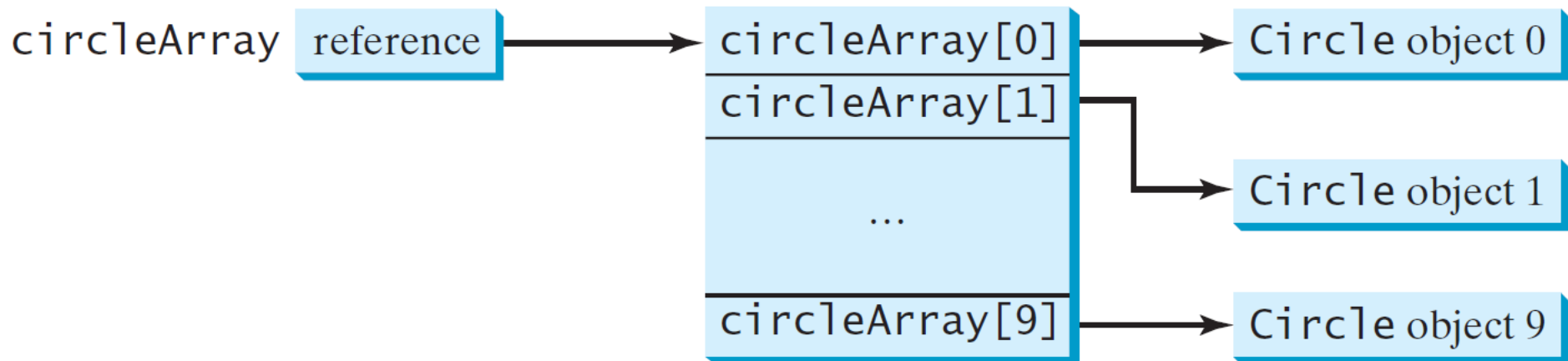
```
Circle[] circleArray = new Circle[10];
```

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.



# Array of Objects, cont.

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



# Array of Objects, cont.

## Summarizing the areas of the circles



TotalArea

Run



# Immutable Objects and Classes

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*. *If you delete the set method in the Circle class defined ahead, the class would be immutable because radius is private and cannot be changed without a set method.*

*A class with all private data fields and without mutators is not necessarily immutable.* For example, the following class Student has all private data fields and no mutators, but it is mutable.



# Example

```
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn,
        int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

    public BirthDate getBirthDate() {
        return birthDate;
    }
}
```

```
public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear,
        int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }

    public void setYear(int newYear) {
        year = newYear;
    }
    .....
}
```

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
```



# 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.**






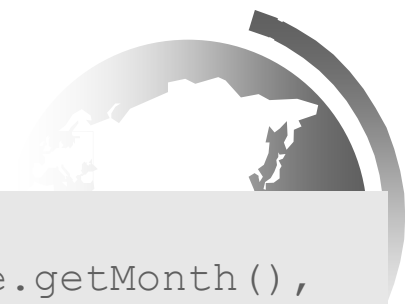
# 不可变类

- Java平台类库包含许多不可变的类，如String, 基本类型的包装类(如Integer, Long等)， BigInteger和BigDecimal。
- Java 8中提供了LocalDate, LocalTime, LocalDateTime也是不可变类。
- 不可变类的优点：更易于设计、实现和使用，更安全。
- 使类变为不可变类，要遵循下面五条规则：
  - 1. 不要提供任何会修改对象状态的方法（mutator）
  - 2. 保证类不会被扩展。防止粗心或恶意的子类假装对象的状态已改变。一般做法是使这个类为final。
  - 3. 使所有域都是final的
  - 4. 使所有域都是private的
  - 5. 确保对于任何可变组件的互斥访问。如果类有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用



## □ NOTE: 必要时进行保护性拷贝

```
public class Student {  
    private int id;  
    private BirthDate birthDate;  
  
    public Student(int ssn,  
        int year, int month, int day) {  
        id = ssn;  
        birthDate = new BirthDate(year, month, day);  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public BirthDate getBirthDate() {  
        return birthDate;  
    }  
}
```



```
public BirthDate getBirthDate() {  
    return new BirthDate(birthDate.getYear(), birthDate.getMonth(),  
        birthDate.getDay());  
}
```

另一个例子：类具有公有的静态`final`数组域，或者返回这种域的访问方法，这是安全漏洞的一个常见根源。

//Potential security hole!

```
public static final Thing[] VALUES = {...};
```

为什么？

修正问题的两种方法：

```
private static final Thing[] PRIVATE_VALUES = {...};
```

```
public static final List<Thing> VALUES =
```

```
    Collections.unmodifiableList(Arrays.asList(PRIVATE_V  
VALUES));
```

```
private static final Thing[] PRIVATE_VALUES = {...};
```

```
public static final Thing[] values(){  
    return PRIVATE_VALUES.clone();
```

```
}
```



# 不可变类

- 不可变对象比较简单，可以只有一种状态，即被创建时的状态。
- 不可变对象本质上是线程安全的。
- 不可变的类可以提供静态工厂，把频繁被请求的实例缓存起来。（基本类型的包装类和 BigInteger 都有这样的静态工厂）。
- 不需要进行保护性拷贝，因为拷贝始终都是原始的对象。



# 不可变类

```
public final class Boolean implements java.io.Serializable,
                                   Comparable<Boolean>
{
    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code true}.
     */
    public static final Boolean TRUE = new Boolean(true);

    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code false}.
     */
    public static final Boolean FALSE = new Boolean(false);
```

```
private final boolean value;
```

```
public Boolean(boolean value) {
    this.value = value;
}
```

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```



# 不可变类

- `public static final Complex ZERO = new Complex(0,0);`
- `public static final Complex ONE = new Complex(1,0);`
- `public static final Complex I = new Complex(0,1);`

```
private static final Object NULL = new Object() {  
    public int hashCode() {  
        return 0;  
    }  
  
    public String toString() {  
        return "java.util.EnumMap.NULL";  
    }  
};
```

# 不可变类

- 缺点：对于每一个不同的值都需要一个单独的对象。

```
BigInteger moby = .....;  
moby = moby.flipBit(0);
```

虽然只是修改了一个bit，但是flipBit需要创建一个新的BigInteger，消耗时间和空间。

如果执行一个多步骤的操作，则每个步骤都会产生一个新的对象，除了最后的结果之外其他对象都会被丢弃。

与BigInteger类似，BitSet代表一个任意长度的位序列，但它是可变的。

# 不可变类

□ 其他的例子，如 String类，也是不可变类

```
String s = “abc”;
```

```
for(int i= 1; i<10;i++)
```

```
    s += i;
```

String的可变配套类为StringBuilder





# 不可变类

```
public static void main(String[] args) {  
    String s = "a";  
    Integer t = 1;  
    Long w1 = 1L;  
    long l = 1;  
    for(int i = 0; i < 100; i++){  
        s = s + i;  
        t = t + i;  
        w1 = w1 + i;  
        l = l + i;  
    }  
}
```

# 不可变类

class文件反编译结果

```
public static void main(String args[])
{
    String s = "a";
    Integer t = Integer.valueOf(1);
    Long w1 = Long.valueOf(1L);
    long l = 1L;
    for(int i = 0; i < 100; i++)
    {
        s = (new StringBuilder(String.valueOf(s))).append(i).toString();
        t = Integer.valueOf(t.intValue() + i);
        w1 = Long.valueOf(w1.longValue() + (long)i);
        l += i;
    }
}
```

# 不可变类

## ▣ Java反编译工具

### – JAD Java Decompiler

jad xxxx.class

▣ <https://varaneckas.com/jad/>

- [Jad 1.5.8g for Windows 9x/NT/2000 on Intel platform](#) (238600 bytes).
- [Jad 1.5.8g for Mac OS X 10.4.6 on Intel platform](#) (170707 bytes, compiled by GCC 4.0).
- [Jad 1.5.8e for HP-UX 11.x](#) (293214 bytes).
- [Jad 1.5.8e for Linux on Intel platform](#) (214917 bytes).
- [Jad 1.5.8e for Linux \(statically linked\)](#) (389972 bytes) - take this version if the one above crashes or displays the
- [Jad 1.5.8d for OS/2](#) (288717 bytes) - OS/2 version was cross-compiled on Windows using EMX and RSXNT pack enough).
- [Jad 1.5.8c for FreeBSD 4.0](#) (322311 bytes) - statically linked.
- [Jad 1.5.8c for OpenBSD 2.7](#) (334499 bytes) - statically linked.
- [Jad 1.5.8c for NetBSD 1.5](#) (324440 bytes) - statically linked.
- [Jad 1.5.8c for Solaris 8 on Intel platform](#) (264751 bytes).
- [Jad 1.5.8c for Mac OS X \(Darwin 1.3\) on PowerPC platform](#) (266264 bytes).

## Installation

Unzip jad.zip file into any appropriate directory on your hard drive.

This will create two files:

- an executable file named 'jad.exe' (Windows 9x/NT/2000) or 'jad' (UNIX)
- README file 'Readme.txt', which contains the short user's manual

For UNIX users: make 'jad' executable: `chmod a+x jad`

No further setup is required.



# 不可变类

□ Java反编译工具: Java反编译工具 For Eclipse [支持Eclipse3.x]

jad是java的反编译工具，是命令行执行，反编译出来的源文件可读性较高。可惜用起来不太方便。可利用eclipse下的插件：jadclipse，安装好之后,可在Eclipse中双击.class文件，就能直接看源文件，或jar包中的class文件，也可以直接反编译[可直接查看]。

1.在<http://www.varanekas.com/jad>下载对应版本的jad后解压。

解压缩后将jad.exe拷贝到虚拟机目录下[JDK OR JRE；其实可以任何目录，因为可以在安装好jadclipse后，在eclipse中设置的]，如: D:\JavaSoft\jdk1.6.0\_19\bin

2.下载 jadclipse\_3.3.jar

[http://jaist.dl.sourceforge.net/sourceforge/jadclipse/net.sf.jadclipse\\_3.3.0.jar](http://jaist.dl.sourceforge.net/sourceforge/jadclipse/net.sf.jadclipse_3.3.0.jar)

将jadclipse\_3.3.0.jar复制到eclipse\plugins目录下。

3.启动Eclipse后，在Windows——>Perference——>Java下面应该会多出一个JadClipse目录，

相关的设置可以在此修改配置jadclipse:

path to decompiler=====>>D:\JavaSoft\jdk1.6.0\_19\bin\jad.exe[jad.exe的实际路径]

Directory for temporary files=====>>D:\Class2JavaTemp[临时目录]

4、在Eclipse的Windows——>Perference——>General->Editors->File Associations中修改 “\*.class”默认关联的编辑器为 “JadClipse Class File Viewer”

5、安装完成，双击class文件，Eclipse将自动反编译了。

# 不可变类

- 为确保不可变性，类不允许自身被子类化。除了“使类成为 `final`”之外，还有一种更加灵活的方式：让类的所有构造器都变为私有的或包级私有的，并添加公共的静态工厂来代替公有的构造器。

```
public class Complex{  
    private final double re;  
    private final double im;  
    private Complex(double re, double im){  
        this.re = re;  
        this.im = im;  
    }  
    public static Complex valueOf(double re, double im){  
        return new Complex(re,im);  
    }  
}
```

# 不可变类

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

Constant Pool (Java常量池技术)



```

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

```

```

public class Test
{

    public Test()
    {
    }

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

```

## jad反编译结果

```

public Integer(int value) {
    this.value = value;
}

```

```

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

```

# Scope of Variables

- ❑ **The scope of instance and static variables** is the entire class. They can be declared anywhere inside a class.
- ❑ **The scope of a local variable** starts from its declaration and continues to the end of the block that contains the variable. **A local variable must be initialized explicitly before it can be used.**





# 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.



# Reference the Hidden Data Fields

```
public class F {  
    private int i = 5;  
    private static double k = 0;  
  
    void setI(int i) {  
        this.i = i;  
    }  
  
    static void setK(double k) {  
        F.k = k;  
    }  
}
```

Suppose that f1 and f2 are two objects of F.  
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute  
**this.i = 10**, where **this** refers f1

Invoking f2.setI(45) is to execute  
**this.i = 45**, where **this** refers f2



# 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

# 对象构建TIPs

## □ 1. 遇到多个构造器参数时可以考虑用构建器(Builder)

假设一个类有多个参数

```
public class NutritionFacts {  
    private final int servingSize;      // (mL)           required  
    private final int servings;         // (per container) required  
    private final int calories;         //                optional  
    private final int fat;              // (g)            optional  
    private final int sodium;           // (mg)            optional  
    private final int carbohydrate;     // (g)            optional  
}
```

大多数程序员会采用**重叠构造器**的方式——提供第一个只有必要参数的构造器，第二个构造器有一个可选参数，第三个有两个可选参数，以此类推，最后一个构造器包含所有可选参数。

## 重叠构造器的方式

```
public NutritionFacts(int servingSize, int servings) {
    this(servingSize, servings, 0);
}

public NutritionFacts(int servingSize, int servings, int calories) {
    this(servingSize, servings, calories, 0);
}

public NutritionFacts(int servingSize, int servings, int calories, int fat) {
    this(servingSize, servings, calories, fat, 0);
}

public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
}

public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int
carbohydrate) {
    super();
    this.servingSize = servingSize;
    this.servings = servings;
    this.calories = calories;
    this.fat = fat;
    this.sodium = sodium;
    this.carbohydrate = carbohydrate;
}
```

使用: `NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);`

## 缺点：

通常需要许多你本不想设置的参数，但还是不得不为它们传递值，这个例子中fat传递了一个为0的初始值。如果参数仅此6个，情况还不算太糟糕，但是随着成员域的增多，这种情况很快就会失去控制。首先更多的参数构造器会让代码非常**难以阅读**，使用这个类需要非常仔细的探究每个参数具体是什么意思；其次一长串类型相同的参数会**导致一些人为的微妙错误**，比如客户端不小心颠倒了其中连个参数的位置，此时编译器不会报错，但是程序运行会有错误的显示。



另一种方案：用一个无参的构造器来创建对象，然后调用setter方法来设置每个必要的参数，以及每个相关的可选参数。

```
public class NutritionFacts {  
    /** 每罐的容量 ml */  
    private int servingSize    = -1;  
    /** 每箱的数量 */  
    private int servings       = -1;  
    /** 卡路里 */  
    private int calories       = 0;  
    /** 脂肪含量 */  
    private int fat            = 0;  
    /** 钠含量 */  
    private int sodium         = 0;  
    /** 糖含量 */  
    private int carbohydrate   = 0;  
  
    public NutritionFacts() { }  
  
    public void setServingSize(int servingSize)    { this.servingSize = servingSize; }  
    public void setServings(int servings)          { this.servings = servings; }  
    public void setCalories(int calories)          { this.calories = calories; }  
    public void setFat(int fat)                    { this.fat = fat; }  
    public void setSodium(int sodium)              { this.sodium = sodium; }  
    public void setCarbohydrate(int carbohydrate)  { this.carbohydrate = carbohydrate; }  
}
```

```
NutritionFacts cocaCola = new NutritionFacts();  
cocaCola.setServingSize(240);  
cocaCola.setServings(8);  
cocaCola.setCalories(100);  
cocaCola.setSodium(35);  
cocaCola.setCarbohydrate(27);
```

- 这种方式将构造过程分到了几个调用中，在构造过程中对象可能处于不一致的状态。类无法仅仅通过校验构造器参数的有效性来保证一致性。
- 阻止了把类做成不可变类的可能，这时需要程序员付出额外的努力来确保它线程安全。





# 基于Builder的构造

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;
        private int carbohydrate = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

        public Builder calories(int val) {
            calories = val;
            return this;
        }
    }
}
```

```
public Builder fat(int val) {
    fat = val;
    return this;
}

public Builder sodium(int val) {
    sodium = val;
    return this;
}

public Builder carbohydrate(int val) {
    carbohydrate = val;
    return this;
}

public NutritionFacts build() {
    return new NutritionFacts(this);
}

public NutritionFacts(Builder builder) {
    super();
    this.servingSize = builder.servingSize;
    this.servings = builder.servings;
    this.calories = builder.calories;
    this.fat = builder.fat;
    this.sodium = builder.sodium;
    this.carbohydrate = builder.carbohydrate;
}
```

这里 NutritionFacts 是不可变的，所有默认参数值都单独放在一个地方。builder 的具名的 setter 方法返回 builder 本身，以便可以进行类似链式操作的方式进行可选参数的赋值，客户端代码可以像这样：

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
    .calories(100)
    .sodium(35)
    .carbohydrate(27)
    .build();
```



# 对象构建TIPs

- 2. 有时可以考虑用静态工厂方法代替构造器

```
public static Boolean valueOf(boolean b) {  
    return (b ? TRUE : FALSE);  
}
```

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



- 静态工厂方法的优点之一：它们有名字，更易于阅读。

```
public static BigInteger valueOf(long val) {  
    // If -MAX_CONSTANT < val < MAX_CONSTANT, return stashed constant  
    if (val == 0)  
        return ZERO;  
    if (val > 0 && val <= MAX_CONSTANT)  
        return posConst[(int) val];  
    else if (val < 0 && val >= -MAX_CONSTANT)  
        return negConst[(int) -val];  
  
    return new BigInteger(val);  
}
```

返回有可能是素数的、具有指定长度的正 BigInteger:

```
public static BigInteger probablePrime(int bitLength, Random rnd) {  
    if (bitLength < 2)  
        throw new ArithmeticException("bitLength < 2");  
  
    return (bitLength < SMALL_PRIME_THRESHOLD ?  
        smallPrime(bitLength, DEFAULT_PRIME_CERTAINTY, rnd) :  
        largePrime(bitLength, DEFAULT_PRIME_CERTAINTY, rnd));  
}
```

- 静态工厂方法的优点之二：不必在每次调用它们的时候都创建一个新的对象。
- 这使得不可变类可以使用预先构建好的实例，或者将构建好的实例缓存起来，进行重复利用。

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



```
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() {}
}
```

- 静态工厂方法的优点之三：可以返回原返回类型的任何子类型的对象。我们在选择返回对象的类有更大的灵活性。

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c) {  
    return new UnmodifiableCollection<>(c);  
}
```

```
static class UnmodifiableCollection<E> implements Collection<E>, Serializable {  
    private static final long serialVersionUID = 1820017752578914078L;  
  
    final Collection<? extends E> c;  
  
    UnmodifiableCollection(Collection<? extends E> c) {  
        if (c==null)  
            throw new NullPointerException();  
        this.c = c;  
    }  
}
```

```

/**
 * @serial include
 */
static class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    private static final long serialVersionUID = 1820017752578914078L;

    final Collection<? extends E> c;

    UnmodifiableCollection(Collection<? extends E> c) {
        if (c==null)
            throw new NullPointerException();
        this.c = c;
    }

    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}

    public Iterator<E> iterator() {...}

    public boolean add(E e) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Object o) {
        throw new UnsupportedOperationException();
    }

    public boolean containsAll(Collection<?> coll) {
        return c.containsAll(coll);
    }
    public boolean addAll(Collection<? extends E> coll) {
        throw new UnsupportedOperationException();
    }
}

```





```
public static <T> Collection<T> synchronizedCollection(Collection<T> c) {  
    return new SynchronizedCollection<>(c);  
}
```

```
static class SynchronizedCollection<E> implements Collection<E>, Serializable {  
    private static final long serialVersionUID = 3053995032091335093L;  
  
    final Collection<E> c; // Backing Collection  
    final Object mutex;    // Object on which to synchronize  
  
    SynchronizedCollection(Collection<E> c) {  
        this.c = Objects.requireNonNull(c);  
        mutex = this;  
    }  
  
    SynchronizedCollection(Collection<E> c, Object mutex) {  
        this.c = Objects.requireNonNull(c);  
        this.mutex = Objects.requireNonNull(mutex);  
    }  
  
    public int size() {  
        synchronized (mutex) {return c.size();}  
    }  
    public boolean isEmpty() {  
        synchronized (mutex) {return c.isEmpty();}  
    }  
    public boolean contains(Object o) {  
        synchronized (mutex) {return c.contains(o);}  
    }  
    public Object[] toArray() {  
        synchronized (mutex) {return c.toArray();}  
    }  
}
```



# 对象构建TIPs

3. 单例对象构建：通常用于那些本质上唯一的对象，如文件系统、窗口管理等。

Java 1.5之前，实现Singleton有两种方法：

```
public class Elvis{  
    public static final Elvis INSTANCE = new Elvis();  
    private Elvis(){...}  
}
```

```
public class Elvis{  
    private static final Elvis INSTANCE = new Elvis();  
    private Elvis(){...}  
    public static Elvis getInstance() {return INSTANCE;}  
}
```

注意：有特权的客户端可以借助`AccessibleObject.setAccessible`方法，通过**反射机制**，调用私有构造器。

要抵御这种攻击，可以修改构造器，要求创建第二个实例的时候抛出异常。



□ 从Java1.5起，实现Singleton有第三种方法，只需要编写一个包含单个元素的枚举即可：

```
public enum Elvis{  
    INSTANCE;  
}
```

Enum提供了序列化方式，而且绝对防止多次实例化。



# 包(Package)

- package是一个为了方便管理组织java文件的目录结构，并防止不同java文件之间发生命名冲突。
- package语句作为Java源文件的第一条语句。（若缺省该语句，则指定为无名包。）约定俗成的给包起名为把公司域名倒过来写，如com.sun  
Java编译器把包对应于文件系统的目录管理，package语句中，用‘.’来指明包（目录）的层次，例如package com.sun;则该文件中所有的类位于.\com\sun目录下。



# package

## ➤ 类的导入

- 可以在每个类前添加完整的包名

```
java.time.LocalDate today = java.time.LocalDate.now();
```

- 使用import语句

```
import java.time.*;
```

```
LocalDate today = LocalDate.now();
```

也可以只导入包中的特定类

```
import java.time.LocalDate;
```

import java.time.\*的语法比较简单，对代码的大小也没有任何负面的影响。但如果能明确指出导入的类，可以让读者明确知道加载了哪个类。



# package

➤ 当发生命名冲突时，需要注意包的名字  
java.util和java.sql都有Date类，如果：

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

```
import java.sql.*;
```

```
Date today; //编译错误
```

需要明确import java.util.Date

或 java.util.Date today = new java.util.Date();



# package

- `import`不会递归，只会引入当前package下的直接类

`import java.util.*`，只会引入`java.util`下的直接类，而不会引入`java.util`下嵌套包的类，如不会引入`java.util.zip`下的类。

试图嵌套引入的形式也是无效的，如`import java.util.*.*`



# package

- C++程序员会将import与#include弄混。其实，两者并没有共同之处。
- C++中需要用#include将外部声明加载进来，这是因为C++编译器只能查看正在编译的文件和include的文件；而java编译器可以查看其他文件，只要告诉它到哪里去查看就可以了。
- 在Java中，显式给出包名时（如java.util.Date），则不需要import；而C++中无法避免用#include。
- Java中，package与import类似于C++中的namespace和using.





# package

- 默认import java.lang.\*;
- 我们在程序中经常使用System.out这个类，为什么没有import System.out呢，因为java.lang 这个套件实在是太常用到了，几乎没有程序不用它的，所以不管你有没有写import java.lang;，编译器都会自动帮你补上，也就是说编译器只要看到没有package的类别，它就会自动去java.lang 里面找找看，看这个类别是不是属于这个套件的。所以我们就不用特别去import java.lang 了。



# package

## ➤ 静态导入

import不仅可以导入类，还可以导入静态方法和静态域。

```
import static java.lang.System.*;
```

就可以使用System类的静态方法和静态域了。

```
out.println("Hello World!"); //System.out  
exit(0); //System.exit
```



# package

## ➤ 静态导入

也可以导入特定的方法或域

```
import static java.lang.System.out;
```



# package

## ➤ 将类放入包中

- 要将类放到包中，必须将包的名字放在源文件的开始，例如：

```
package com.horstmann.corejava  
public class Employee  
{  
...  
}
```

- 如果没有放置package语句，则这个类被放置在默认包（default package)中。



# package

- 需要将类文件切实安置到其所归属之Package所对应的相对路径下。

```
package com.horstmann.corejava  
public class Employee  
{  
    ...  
}
```

- 需要将Employee.java放到com/horstmann/corejava目录下
- 编译器也会将class文件放到相同的目录结构中。



# package

## □ 编译器在编译源文件时不检查目录结构

例如：以下面程序为例：假设此Hello.java文件在D:\Java\下

```
package p;  
public class Hello{  
    public static void main(String args[]){  
        System.out.println("Hello World!");  
    }  
}
```

D:\Java>javac Hello.java 此程序可以编译通过.接着执行。

D:\Java>java Hello 但是执行时，却提示以下错误！

Exception in thread “main” java.lang.NoClassDefFoundError: Hello (wrong name: p/Hello)

```
at java.lang.ClassLoader.defineClass0(Native Method)  
at java.lang.ClassLoader.defineClass(ClassLoader.java:537)  
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)  
at java.net.URLClassLoader.defineClass(URLClassLoader.java:251)  
at java.net.URLClassLoader.access$100(URLClassLoader.java:55)  
.....  
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:274)  
at java.lang.ClassLoader.loadClass(ClassLoader.java:235)  
at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:302)
```

原因是我们把生成的Hello.class规定打包在D:\Java\p文件中，必须在p文件中才能去运行。所以应该在D:\Java目录下建立一个p目录,然后把Hello.class放在它下面，执行时，可正常通过！

D:\Java\>java p.Hello 就会输出：Hello World!

# package

当JVM加载某个class时，它首先找到环境变量CLASSPATH，将其中的目录作为查找.class文件的根目录

如：

```
package com.example;
```

```
public class List{
```

```
.....
```

```
}
```

```
CLASSPATH=.;D:\JAVA\LIB; C:\flavors\grape.jar
```

从CLASSPATH的目录中找子目录com\example，再找相应的class文件(List.class)



# JAR

- **JAR: Java ARchive.** A group of Java classes and supporting files combined into a single file compressed with ZIP format, and given .JAR extension.
  
- Advantages of JAR files:
  - compressed; quicker download
  - just one file; less mess
  - can be executable
  
- The closest you can get to having a .exe file for your Java application.





# Creating a JAR archive

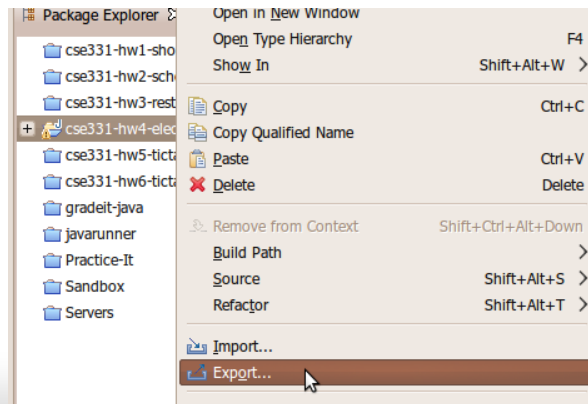
- from the command line:

```
jar -cvf filename.jar files
```

- Example:

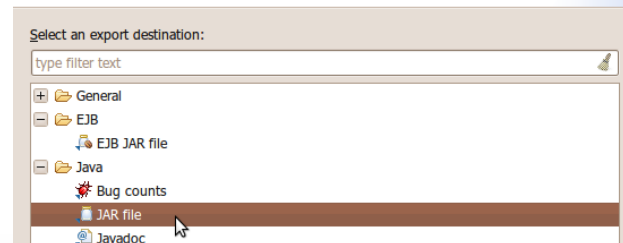
```
jar -cvf MyProgram.jar *.class *.gif *.jpg
```

- some IDEs (e.g. Eclipse) can create JARs automatically
  - File → Export... → JAR file



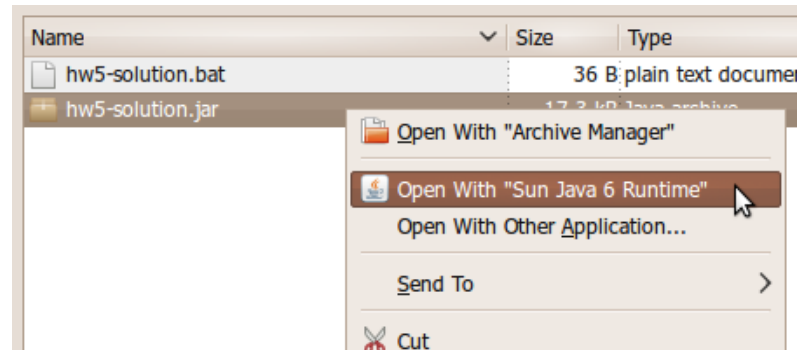
## Select

Export resources into a JAR file on the local file system.



# Running a JAR

- Running a JAR from the command line:
  - `java -jar filename.jar`
- Most OSes can run JARs directly by double-clicking them:



# Making a runnable JAR

- **manifest file:** Used to create a JAR runnable as a program.

```
jar -cvmf manifestFile MyAppletJar.jar  
      mypackage/*.class *.gif
```

*Contents of MANIFEST file:*

Main-Class: <b>MainClassName</b>
----------------------------------

- Eclipse will automatically generate and insert a proper manifest file into your JAR if you specify the main-class to use.



# Resources inside a JAR

- You can embed external resources inside your JAR:
  - images (GIF, JPG, PNG, etc.)
  - audio files (WAV, MP3)
  - input data files (TXT, DAT, etc.)
  - ...
- But code for opening files will look outside your JAR, not inside it.

```
- Scanner in = new Scanner(new File("data.txt")); // fail
- ImageIcon icon = new ImageIcon("pony.png"); // fail
- Toolkit.getDefaultToolkit().getImage("cat.jpg"); // fail
```



# Accessing JAR resources

- Every class has an associated `.class` object with these methods:
  - `public URL getResource(String filename)`
  - `public InputStream getResourceAsStream(String name)`
- If a class named `Example` wants to load resources from within a JAR, its code to do so should be the following:
  - `Scanner in = new Scanner(  
    Example.class.getResourceAsStream("/data.txt"));`
  - `ImageIcon icon = new ImageIcon(  
    Example.class.getResource("/pony.png"));`
  - `Toolkit.getDefaultToolkit().getImage(  
    Example.class.getResource("/images/cat.jpg"));`
  - (Some classes like `Scanner` read from streams; some like `Toolkit` read from URLs.)
  - NOTE the very important leading `/` character; without it, you will get a null result

