CEN 419
Introduction to Java Programming

Instructor: H. Esin ÜNAL

AUTUMN 2018

Slides are modified from original slides of Y. Daniel Liang

WEEK 12

Inheritance and Polymorphism

Motivations

Suppose you will **define classes to model:**

- ***circles,***
- ***rectangles***
- ***triangles***

These classes have many common features.

What is the best way to design these classes so to avoid redundancy?

The answer is to use **inheritance**.

Inheritance

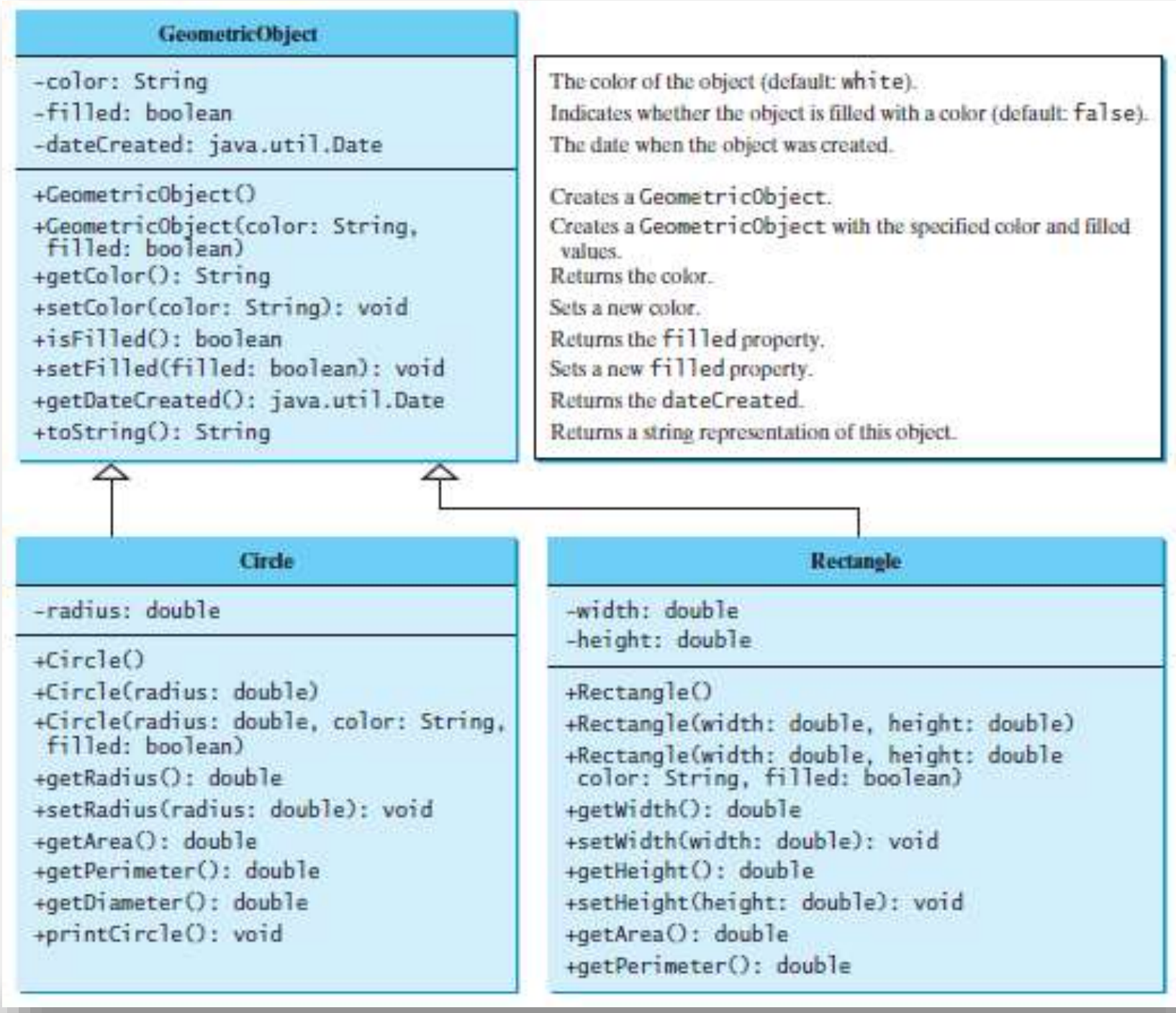
- Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.
- You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.
- Such an *inherited class* is called a **subclass** of its *parent class* or **superclass**.
- It is a mechanism for code reuse.

Superclasses and Subclasses

- The keyword **extends** tells the compiler that the **Circle** class extends the **GeometricObject** class, thus inheriting the methods it has.

NOTE:

Even if you don't inherit a class from another class, the compiler automatically inherit the class from **Object** class. Every class you declare is inherited directly or indirectly from the **Object** class.


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Superclasses and Subclasses

http://www.cs.armstrong.edu/liang/intro10e/html/SimpleGeometricObject.html

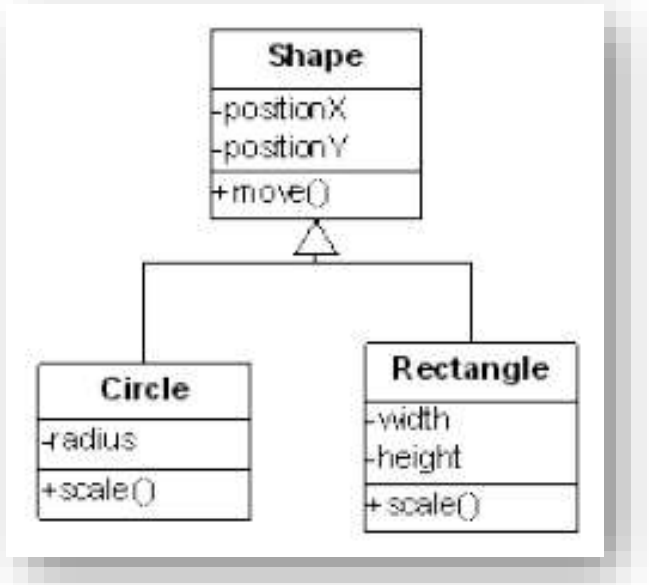
http://www.cs.armstrong.edu/liang/intro10e/html/CircleFromSimpleGeometricObject.html

http://www.cs.armstrong.edu/liang/intro10e/html/RectangleFromSimpleGeometricObject.html

http://www.cs.armstrong.edu/liang/intro10e/html/TestCircleRectangle.html

A Simpler Example

```
class Shape{
  int positionX;
  int positionY;
  void move(int newX, int newY){
    positionX = newX;
    positionY = newY;
  }
}
class Circle extends Shape{
  int radius;
  void scale(int scaleFactor){
    radius *= scaleFactor;
  }
}
class Rectangle extends Shape{
  int radius;
  void scale(int scaleFactor){
    width *= scaleFactor;
    height *= scaleFactor;
  }
}
```



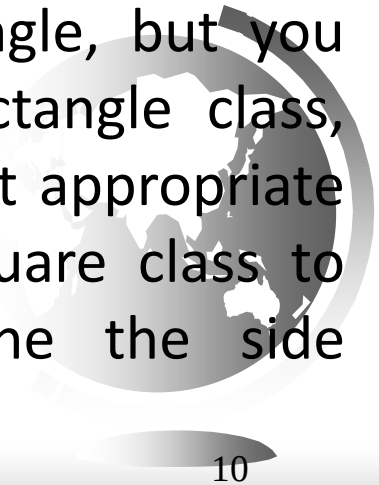

```
Circle c = new Circle();
c.positionX = 10;
c.positionY = 20;
c.radius = 3;
c.move(11,11);
c.scale(5);
```

Important Points of Inheritance

1. Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.
2. Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public getter and setter methods if defined in the superclass.

Important Points of Inheritance

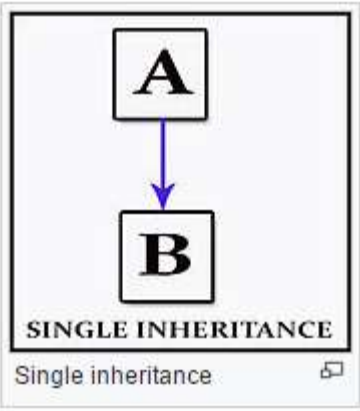
3. Inheritance is used to model the is-a relationship.
   - **Do not blindly extend a class just for the sake of reusing methods.** For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.
   - **Not all is-a relationships should be modeled using inheritance.** For example, a square is a rectangle, but you should not extend a Square class from a Rectangle class, because the width and height properties are not appropriate for a square. Instead, you should define a Square class to extend the GeometricObject class and define the side property for the side of a square.

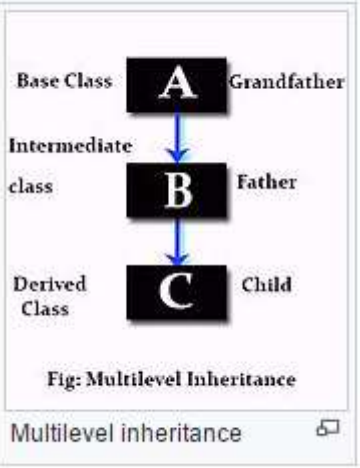


Important Points of Inheritance

4. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as *single inheritance*..

5. *Multilevel inheritance* is where a subclass is inherited from another subclass.





A derived class with multilevel inheritance is declared as follows:

```
Class A(...); //Base class
Class B : public A(...); //B derived from A
Class C : public B(...); //C derived from B
```

Using the **super** Keyword

- A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors?
- **No. They are not inherited. They are invoked explicitly or implicitly.**
- The keyword **super** refers to the superclass and can be used:
  - To call a superclass constructor
  - To call a superclass method

Calling Superclass Constructors

- **A constructor is used to construct an instance of a class**. Unlike properties and methods, a superclass's **constructors are not inherited in the subclass**.
- They are invoked explicitly or implicitly.
- In order to invoke explicitly use the **super** keyword.

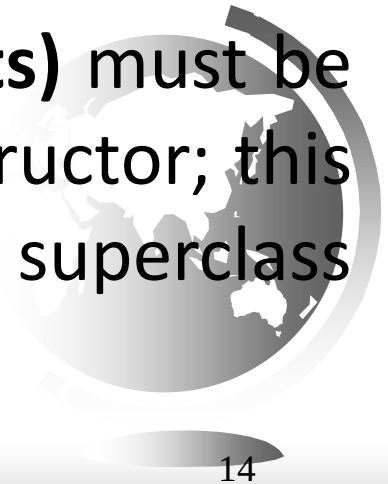
Calling Superclass Constructors

- They **can only be called from the subclasses' constructors**, using the keyword **super**. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

- The syntax to call a superclass's constructor is:

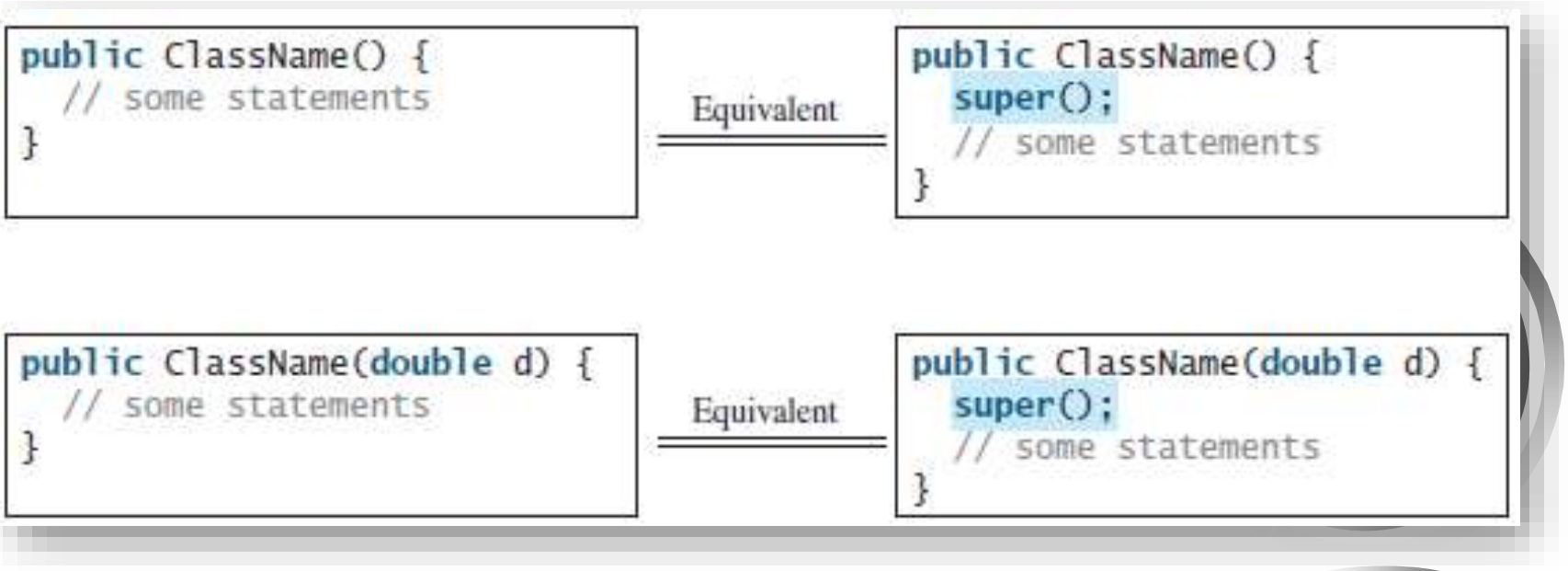
  super(), or **super**(parameters);

- The statement **super()** or **super(arguments)** must be the **first** statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor.



Superclass's Constructor Is Always Invoked

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



CAUTION

- You must use the keyword **super** **to call the superclass constructor.**
- Invoking a **superclass constructor's name** in a subclass causes a syntax error.

Constructor Chaining

- Constructing an instance of a class invokes all the **superclasses' constructors** along the inheritance chain.
- The subclass constructor first invokes its superclass constructor before performing its own tasks.
- This is known as ***constructor chaining***.

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

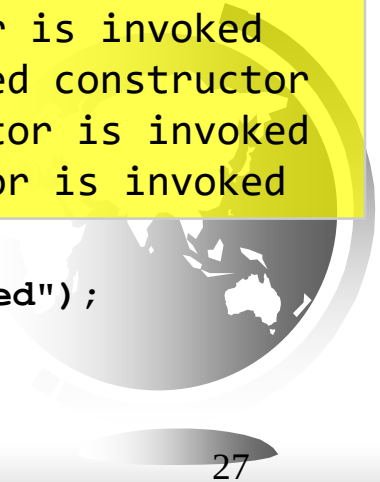
9. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

So, the output is:

(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked



CAUTION

Consider the following code:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Since no constructor is explicitly defined in Apple, Apple's default no-arg constructor is defined implicitly. Since **Apple** is a subclass of **Fruit**, **Apple**'s default constructor automatically invokes **Fruit**'s no-arg constructor. However, **Fruit** does not have a no-arg constructor, because **Fruit** has an explicit constructor defined. Therefore, the program cannot be compiled.

Calling Superclass Methods

- The keyword **super** can also be used to reference a method other than the constructor in the superclass.
- The syntax is:

super.method(parameters);

Defining a Subclass

A **subclass inherits from a superclass**. You can also:

- **Add new properties**
- **Add new methods**
- Override the methods of the superclass

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as ***method overriding***.

- To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.

```
public class Circle extends GeometricObject {
  // Other methods are omitted
  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

NOTE

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden. *If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.*
- Like an instance method, a static method can be inherited. However, **a static method cannot be overridden**. *If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax* ***SuperClassName.staticMethodName****.*

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

The example above show the differences between overriding and overloading.
In (a), the method p(double i) in class A overrides the same method in class B.
In (b), the class A has two overloaded methods: p(double i) and p(int i).
The method p(double i) is inherited from B.

Overriding vs. Overloading

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.

- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

NOTE

- To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the method in the subclass.
- For example:

```
public class CircleFromSimpleGeometricObject
        extends SimpleGeometricObject {
    // Other methods are omitted

   @Override
   public String toString() {
      return super.toString() + "\nradius is " + radius;
   }
}
```

Polymorphism

- The **occurrence of different forms** among the members of a population or colony, or in the life cycle of an individual organism.
- In programming languages and type theory, **polymorphism** (from Greek, "many, much" "form, shape") is the provision of a single interface to entities of different types.
- A **polymorphic type** is one whose operations can also be applied to values of some other type, or types.

Subtype-Supertype

- A class defines a type.
- A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.
- Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

Polymorphism

- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.
- **Polymorphism** means that a variable of a supertype can refer to a subtype object.

Polymorphism Demo

```
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new Circle4(1, "red", false));
    displayObject(new Rectangle1(1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(GeometricObject object) {
    System.out.println("Created on " + object.getDateCreated()
        + ". Color is " + object.getColor());
  }
}
```

Dynamic Binding

A method can be defined in a superclass and overridden in its subclass. For example, the **toString()** method is defined in the **Object** class and overridden in **GeometricObject**.

Consider the following code:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

Which **toString()** method is invoked by **o**? To answer this question, we first introduce two terms: declared type and actual type.

A variable must be declared a type. The type that declares a variable is called the variable's ***declared type***. Here **o**'s declared type is **Object**. A variable of a reference type can hold a **null** value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype.

The ***actual type*** of the variable is the actual class for the object referenced by the variable. Here **o**'s actual type is **GeometricObject**, because **o** references an object created using **new GeometricObject()**.

Which toString() method is invoked by o is determined by o's actual type. This is known as *dynamic binding*.

Dynamic Binding

- *A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.*
- Dynamic binding works as follows: Suppose an object o is an instance of classes C1, C2, ..., Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, ..., and Cn-1 is a subclass of Cn. That is, **Cn is the most general class**, and C1 is the most specific class. In Java, Cn is the Object class. ***If o invokes a method p, the JVM searches the implementation for the method p in C1, C2, ..., Cn-1 and Cn, in this order, until it is found.*** Once an implementation is found, the search stops and the first-found implementation is invoked.

```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

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

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. ***You can invoke it with any object.***

An object of a subtype can be used wherever its supertype value is required. This feature is known as ***polymorphism***.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as ***dynamic binding***.

Casting Objects

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

```
m(new Student());
```

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

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

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

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compile error would occur.

*Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?*

This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

instanceof Operator

- For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime ***ClassCastException*** occurs.
- This can be ensured by using the **instanceof** operator:

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

Example:
Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

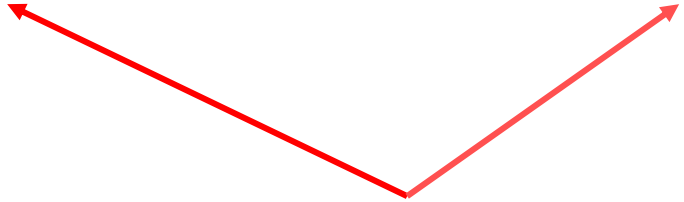
Casting Demo

http://www.cs.armstrong.edu/liang/intro10e/html/CastingDemo.html

CAUTION

- The object member access operator (.) precedes the casting operator. Use parentheses to ensure that casting is done before the . operator, as in

```
((Circle)object).getArea();
```



Use Parantheses!!!

NOTE

- Casting a primitive type value is different from casting an object reference. Casting a primitive type value returns a new value. For example:

```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

- However, casting an object reference does not create a new object. For example:

```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

Now reference variables **o** and **c** point to the same object.

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

The ArrayList Class

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

java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element o at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element o at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element o.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element o from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : E`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

Generic Type

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

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

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

Test ArrayList

http://www.cs.armstrong.edu/liang/intro10e/html/TestArrayList.html

Differences and Similarities between Arrays and ArrayList

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

- ✓ You can sort an array using the **java.util.Arrays.sort(array)** method.
- ✓ To sort an array list, use the **java.util.Collections.sort(arraylist)** method.

Example: ArrayList

- Suppose you want to create an **ArrayList** for storing integers. Can you use the following code to create a list?

```
ArrayList<int> list = new ArrayList<>();
```

- ***This will not work*** because the elements stored in an ArrayList must be of an object type (put `Integer` instead of `int`).

```
ArrayList<int> list = new ArrayList<>();
```

- So, lets write a program that prompts the user to enter a sequence of numbers and displays the distinct numbers in the sequence.
- Assume that the input ends with **0** and **0** is not counted as a number in the sequence.

Distinct Numbers

http://www.cs.armstrong.edu/liang/intro10e/html/DistinctNumbers.html

Array Lists from/to Arrays

- Creating an ArrayList from an array of objects:

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

- Creating an array of objects from an ArrayList:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

max and min in an Array List

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.max(new
ArrayList<String>(Arrays.asList(array)));
```

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.min(new
ArrayList<String>(Arrays.asList(array)));
```

Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));

java.util.Collections.shuffle(list);

System.out.println(list);
```

Case Study: A Custom Stack Class

Objective: A stack class to hold objects.

MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

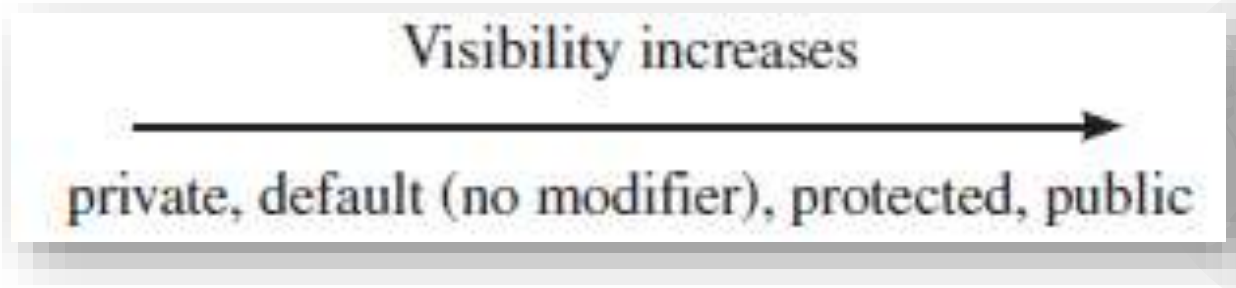
MyStack

http://www.cs.armstrong.edu/liang/intro10e/html/MyStack.html

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.

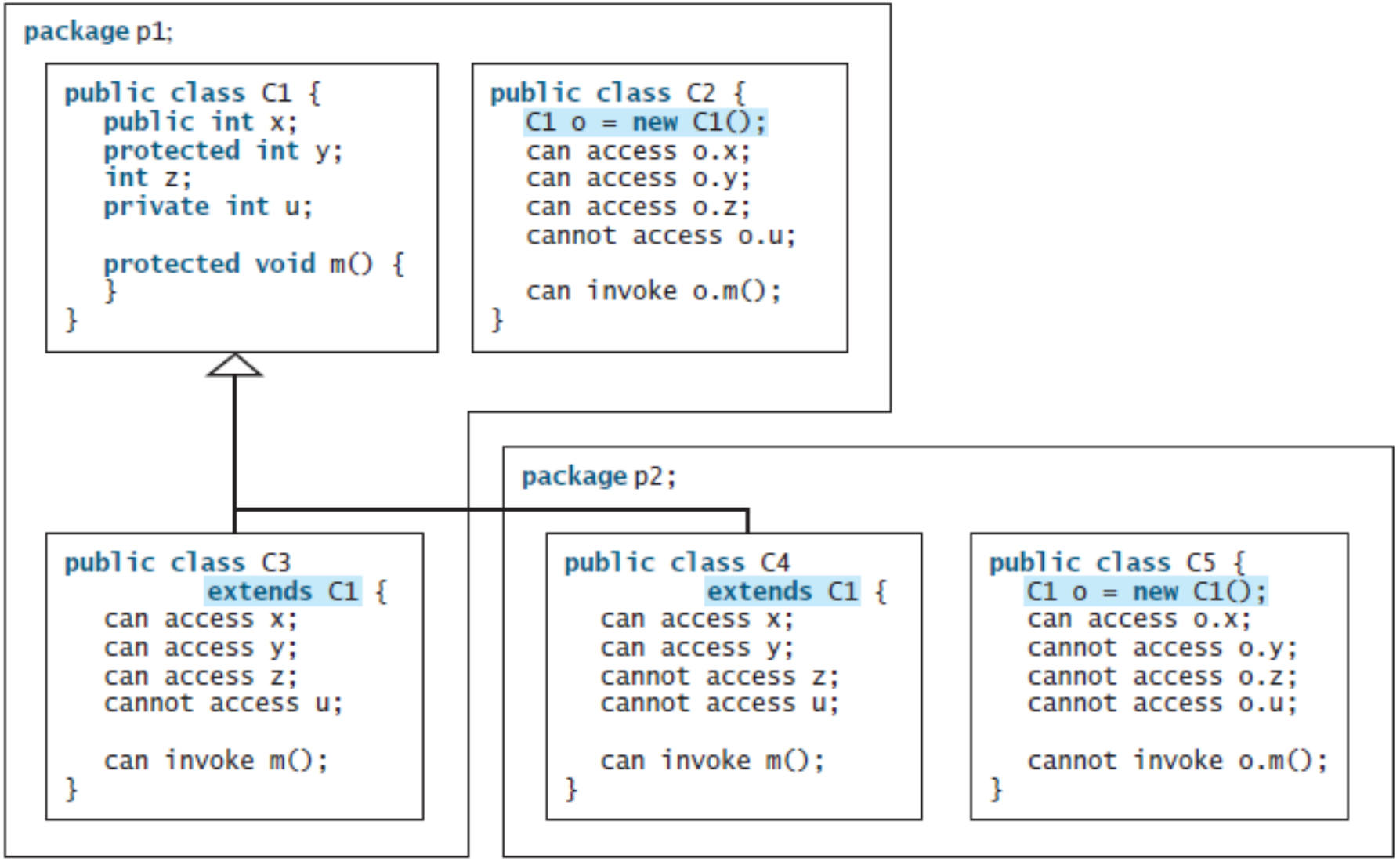
- private, default, protected, public



Accessibility Summary

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The `final` Modifier

- You may occasionally want to prevent classes from being extended. In such cases, use the **final** modifier to indicate that a class is final and **cannot be a parent class**.
- The **Math, String**, **StringBuilder**, and **StringBuffer** classes are final classes.

The `final` Modifier

- The `final` class cannot be extended:

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

- The `final` variable is a constant:

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

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