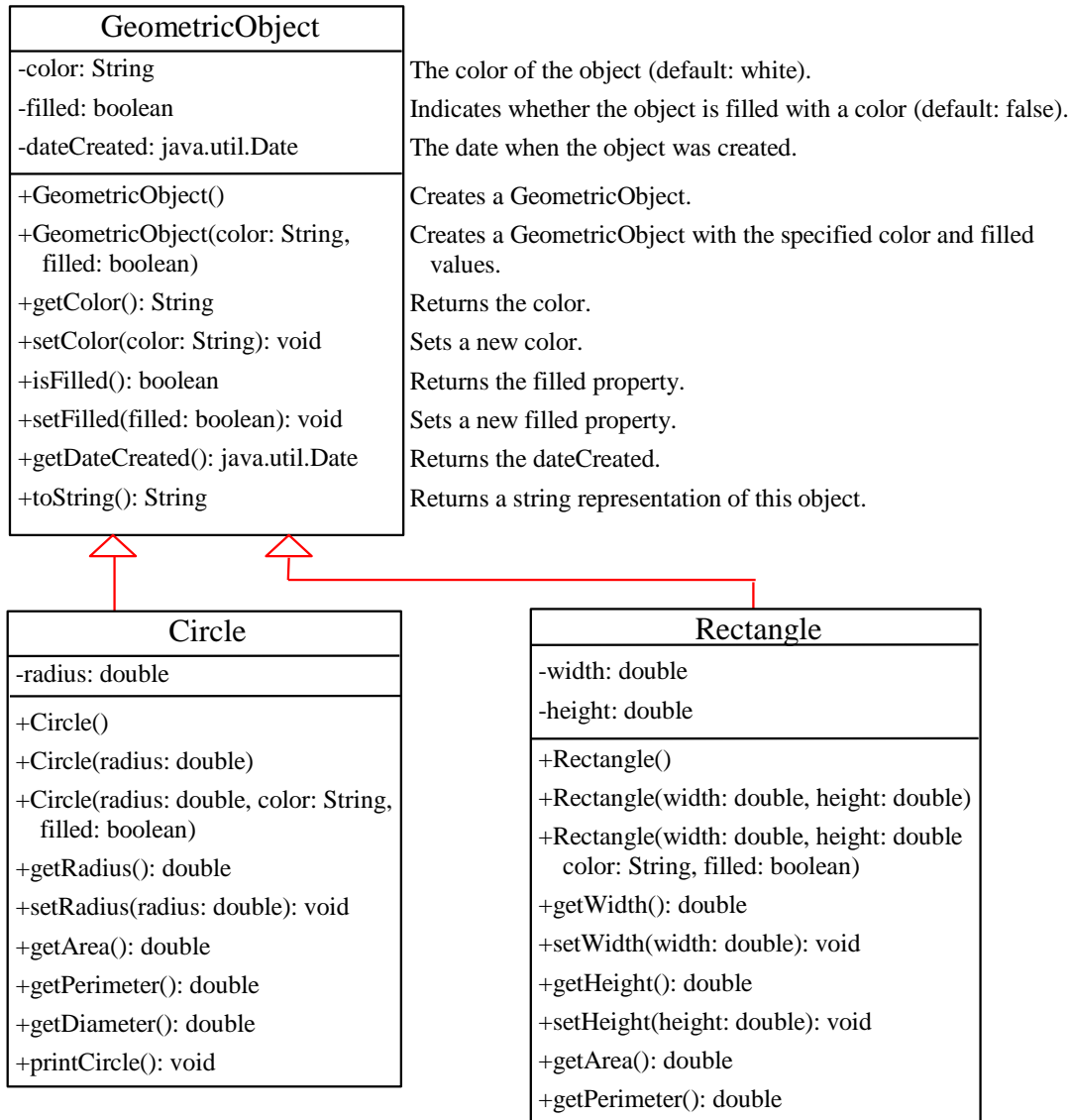


# Inheritance and Polymorphism

Suppose you will define classes to model circles, rectangles, and 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.

# Superclasses and Subclasses



GeometricObject1

Circle4

Rectangle1

TestCircleRectangle

Run

# Are superclass's Constructor Inherited?

No. They are not inherited.

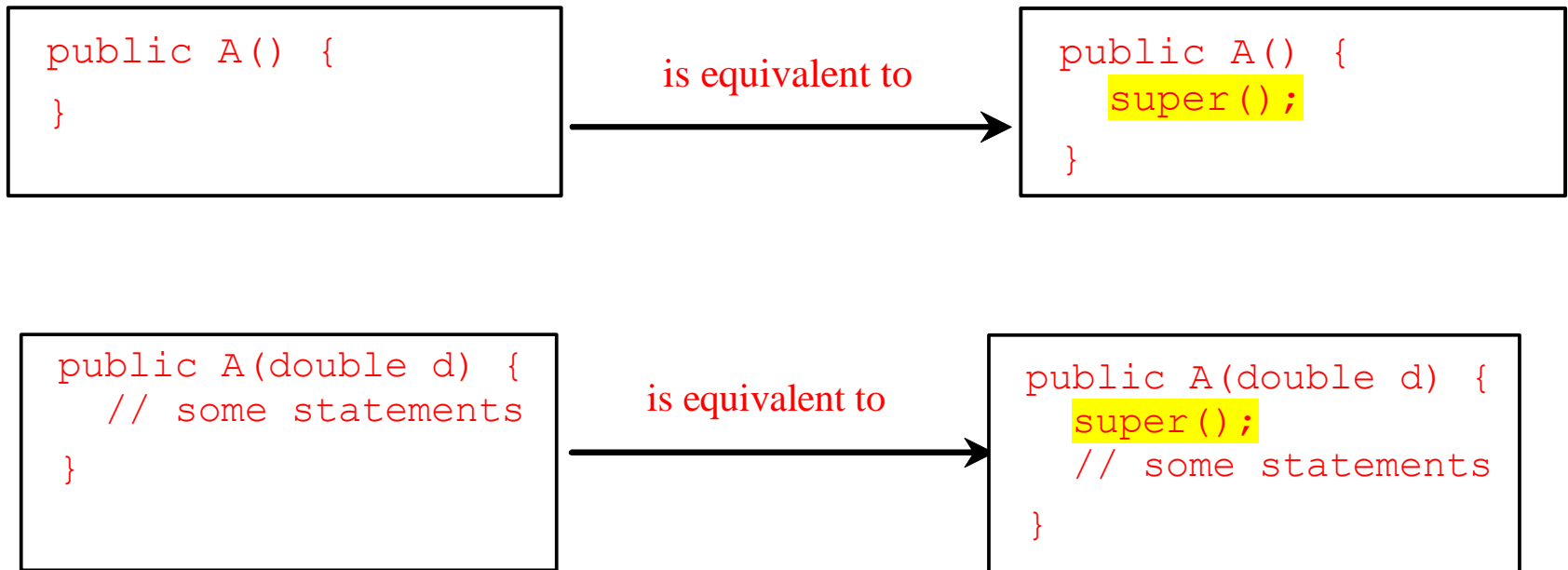
They are invoked explicitly or implicitly.

Explicitly using the `super` keyword.

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 can only be invoked 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.*

# 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,



# Using the Keyword **super**

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

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

## 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. Java requires that the statement that uses the keyword super appear first in the constructor.

# Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        Faculty f1 = 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");
    }
}
```

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        Faculty f1 = 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) {  
        Faculty f1 = 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) {  
        Faculty f1 = 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) {  
        Faculty f1 = new Faculty();  
    }  
}
```

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

**4. Invoke Employee(String)  
constructor**

```
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");  
    }  
}
```

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        Faculty f1 = 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

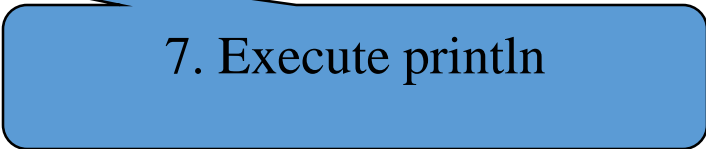
# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        Faculty f1 = 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) {  
        Faculty f1 = 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");  
    }  
}
```



# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        Faculty f1 = 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) {
        Faculty f1 = 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



# Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

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

Reference variable of a Superclass can refer to an object of that class and its subclasses.

```
Fruit f1 = new Fruit();  
Fruit f2 = new Orange();  
Fruit f3 = new Apple();
```

```
1 public class Fruit {  
2     public String toString() {  
3         return "Fruit";  
4     }  
5 }  
6  
7 class Orange extends Fruit {  
8     public String toString() {  
9         return "Orange";  
10    }  
11  
12 }  
13  
14 class Apple extends Fruit {  
15     public String toString() {  
16         return "Apple";  
17    }  
18  
19 }  
20
```

# Declaring a Subclass

A subclass extends properties and methods from the superclass. You can also:

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

# Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

# Overriding Methods in the Subclass

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*.

```
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: Instance Method Inheritance

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

# NOTE: Class/Static Method Inheritance

- A static method of superclass is not a part of a subclass (although it is accessible), so there is no question of overriding it.
- Even if you add another static method in a subclass, identical to the one in its superclass, this subclass static method is unique and distinct from the static method in its superclass.
- In that case, the method defined in the superclass is hidden.

# Method Overriding and Static Methods

- For static methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time.
- For instance methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.



# Static & Instance Methods

```
class A {  
    public int n;  
    public void m1() {  
    }  
    public static void m2() {  
    }  
}  
class B extends A {  
    public void m1() {}  
    public static void m2() {}  
}
```

```
A x = new B();  
B y = new B();  
  
x.m1();  
y.m1();  
  
x.m2();  
y.m2();
```

# Static & Instance Methods

```
public class Animal {  
    public static void testStaticMethod() {  
        System.out.println("The static method in Animal");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Animal");  
    }  
}  
  
public class Cat extends Animal {  
    public static void testStaticMethod() {  
        System.out.println("The static method in Cat");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Cat");  
    }  
}
```

```
public class TestDemo {  
    public static void main(String[] args) {  
        Cat myCat = new Cat();  
        Animal myAnimal = myCat;  
  
        Animal.testStaticMethod();  
        Cat.testStaticMethod();  
  
        myAnimal.testStaticMethod();  
        myCat.testStaticMethod();  
  
        myAnimal.testInstanceMethod();  
        myCat.testInstanceMethod();  
    }  
}
```

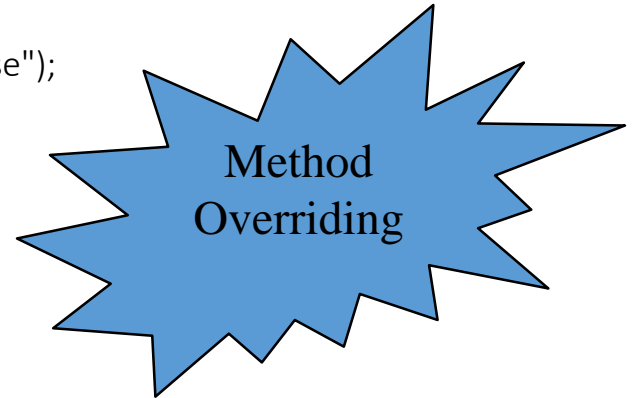
- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

The static method in Animal  
The static method in Cat  
The static method in Animal  
The static method in Cat  
The instance method in Cat  
The instance method in Cat

**NOTE:** An instance method cannot override a static method, and a static method cannot hide an instance method.

```
class Base {           // Superclass
    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }
    // Non-static method which will be overridden in derived class
    public void print() {
        System.out.println("Non-static or Instance method from Base");
    }
}

class Derived extends Base { // Subclass
    // Static is removed here (Causes Compiler Error)
    public void display() {
        System.out.println("Non-static method from Derived");
    }
    // Static is added here (Causes Compiler Error)
    public static void print() {
        System.out.println("Static method from Derived");
    }
}
```



# Overriding vs. Overloading

In a subclass, we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

```
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 Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

# The toString() method in Object

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

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

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

# Polymorphism, Dynamic Binding and Generic Programming

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

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

class GraduateStudent extends Student {
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        Object obj = new GraduateStudent();
        test(obj);
        obj = new Student();
        test(obj);
        obj = new Person();
        test(obj);
        obj = new Object();
        test(obj);
    }

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

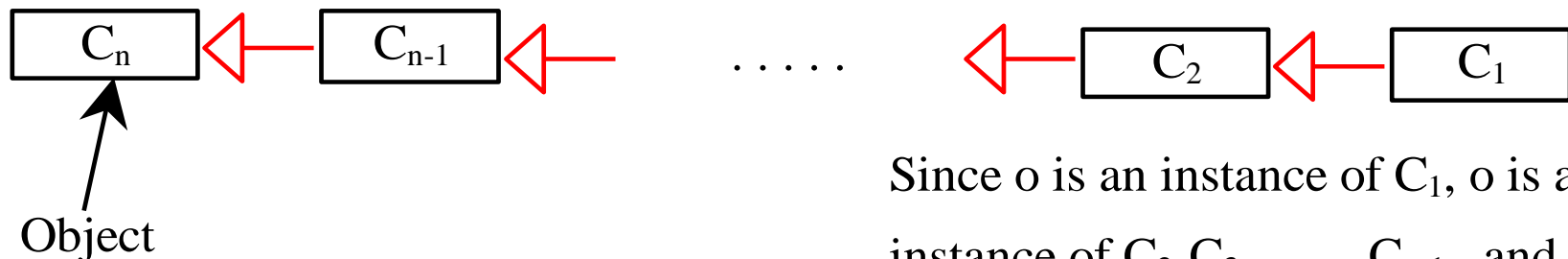
Method *test* takes a parameter of the `Object` type. You can invoke it with any object.

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

When the method `test(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*.

# Dynamic Binding

Dynamic binding works as follows: Suppose an object  $o$  is an instance of classes  $C_1, C_2, \dots, C_{n-1}$ , and  $C_n$ , where  $C_1$  is a subclass of  $C_2$ ,  $C_2$  is a subclass of  $C_3$ , ..., and  $C_{n-1}$  is a subclass of  $C_n$ . That is,  $C_n$  is the most general class, and  $C_1$  is the most specific class. In Java,  $C_n$  is the Object class. If  $o$  invokes a method  $p$ , the JVM searches the implementation for the method  $p$  in  $C_1, C_2, \dots, C_{n-1}$  and  $C_n$ , in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since  $o$  is an instance of  $C_1$ ,  $o$  is also an instance of  $C_2, C_3, \dots, C_{n-1}$ , and  $C_n$



# Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime. See Review Questions 10.7 and 10.9.

# Generic Programming

```
public class PolymorphismDemo {
    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";
    }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter's subclasses (e.g., `Student` or `String`). When an object (e.g., a `Student` object or a `String` object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., `toString`) is determined dynamically.

# 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 section, 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 compilation 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. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

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

# Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Fruit fruit = new Apple();
```

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

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

# Calling a Subclass's Method Using A Superclass's Reference Variable

```
GeometricObj g1;  
g1 = new GeometricObj("Green", true);
```

```
String s1 = g1.getColor(); Green
```

```
g1 = new Circle("Red", false, 2.5);
```

```
String s2 = g1.getColor(); Red
```

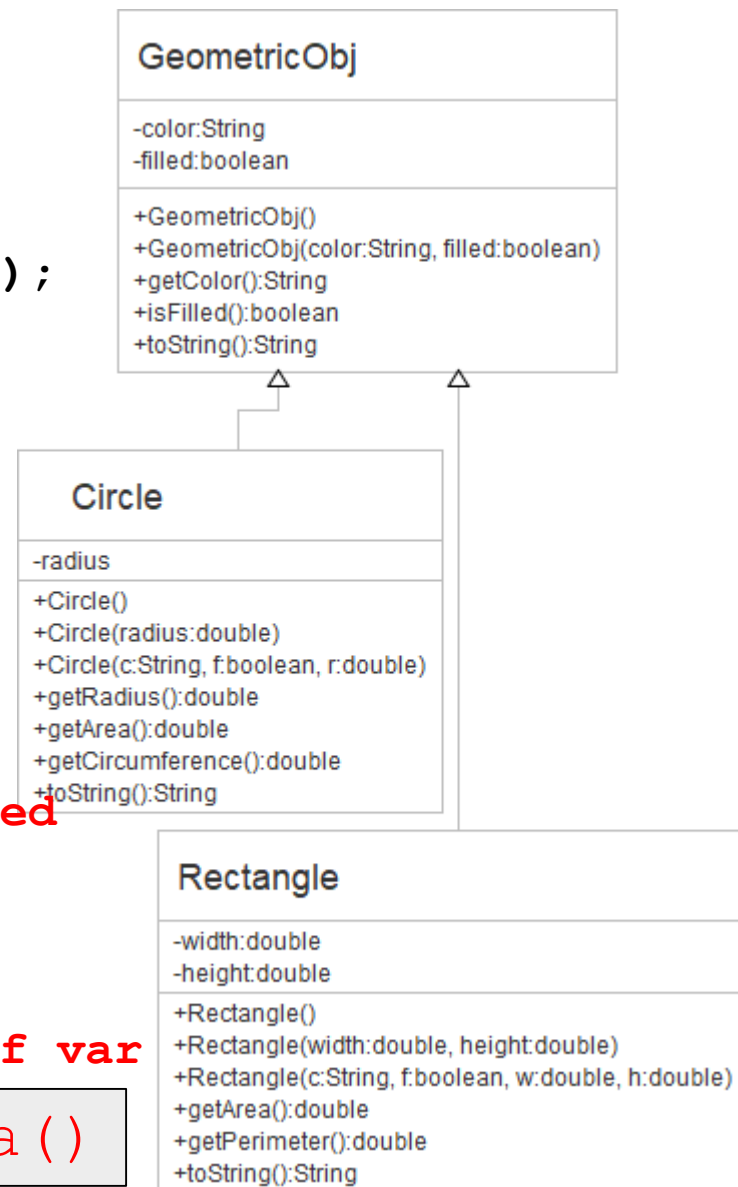
```
String s3 = g1.toString();
```

```
//Circle's toString() will be invoked
```

```
double a = g1.getArea(); X
```

```
// Can't access using superclass ref var
```

```
((Circle)g1).getArea()
```



# The instanceof Operator

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

```
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());  
    ...  
}
```



Explicit casting

# 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;  
}
```



# NOTE

The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.

# (Generic Programming) 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	
+ArrayList()	Creates an empty list.
+add(o: Object) : void	Appends a new element o at the end of this list.
+add(index: int, o: Object) : 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) : Object	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) : Object	Removes the element at the specified index.
+set(index: int, o: Object) : Object	Sets the element at the specified index.

# The MyStack Classes

A stack to hold objects.

MyStack

MyStack	
-list: ArrayList	
+isEmpty(): boolean	
+getSize(): int	
+peek(): Object	
+pop(): Object	
+push(o: Object): void	
+search(o: Object): int	

A list to store elements.

Returns true if this stack is empty.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the first element in the stack from the top that matches the specified element.

# 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`

Visibility increases  
—————→  
`private`, `none` (if no modifier is used), `protected`, `public`

# Accessibility Summary

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

# Visibility Modifiers

package p1;

```
public class C1 {  
    public int x;  
    protected int y;  
    int z;  
    private int u;  
  
    protected void m() {  
    }  
}
```

```
public class C2 {  
    C1 o = new C1();  
    can access o.x;  
    can access o.y;  
    can access o.z;  
    cannot access o.u;  
  
    can invoke o.m();  
}
```



```
public class C3  
    extends C1 {  
    can access x;  
    can access y;  
    can access z;  
    cannot access u;  
  
    can invoke m();  
}
```

package p2;

```
public class C4  
    extends C1 {  
    can access x;  
    can access y;  
    cannot access z;  
    cannot access u;  
  
    can invoke m();  
}
```

```
public class C5 {  
    C1 o = new C1();  
    can access o.x;  
    cannot access o.y;  
    cannot access o.z;  
    cannot access o.u;  
  
    cannot invoke o.m();  
}
```

# 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

- 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.