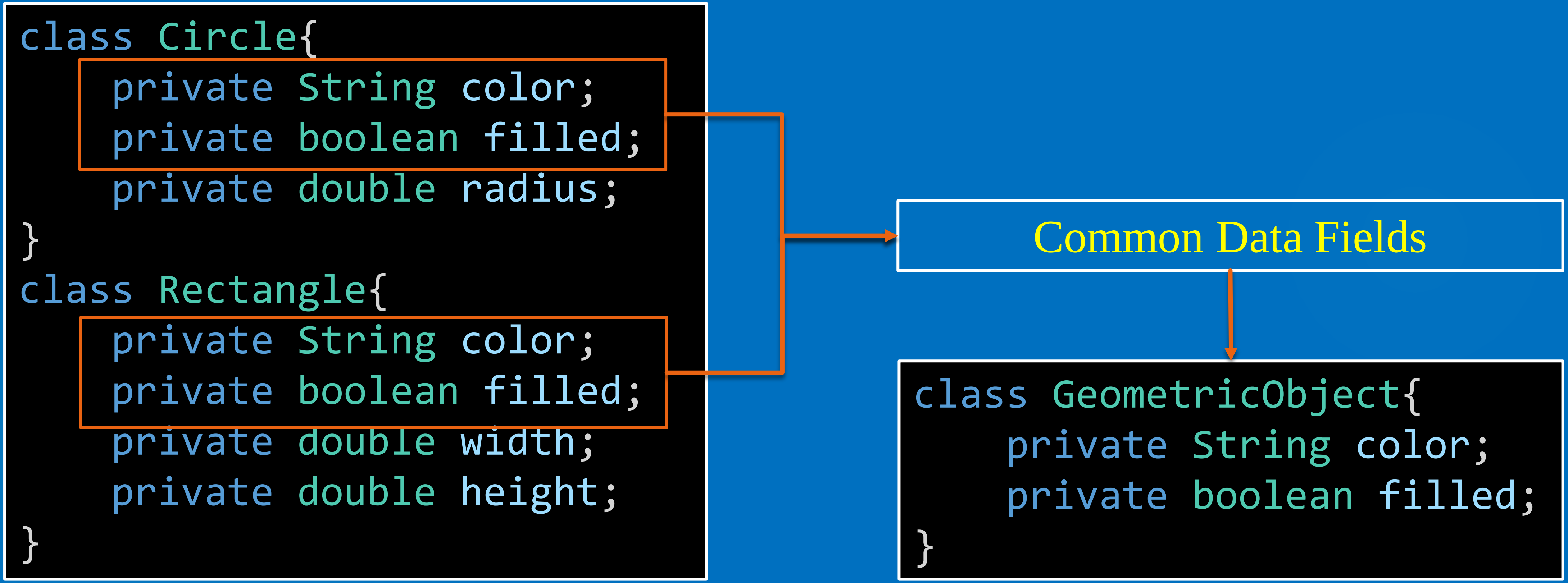
Chapter 07 Inheritance and Polymorphism

Motivations of Inheritance

Object-oriented programming allows you to *define new classes from existing classes*. This is called *inheritance*. *Inheritance is an import and powerful feature for reusing software.* 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*.

Motivations of Inheritance

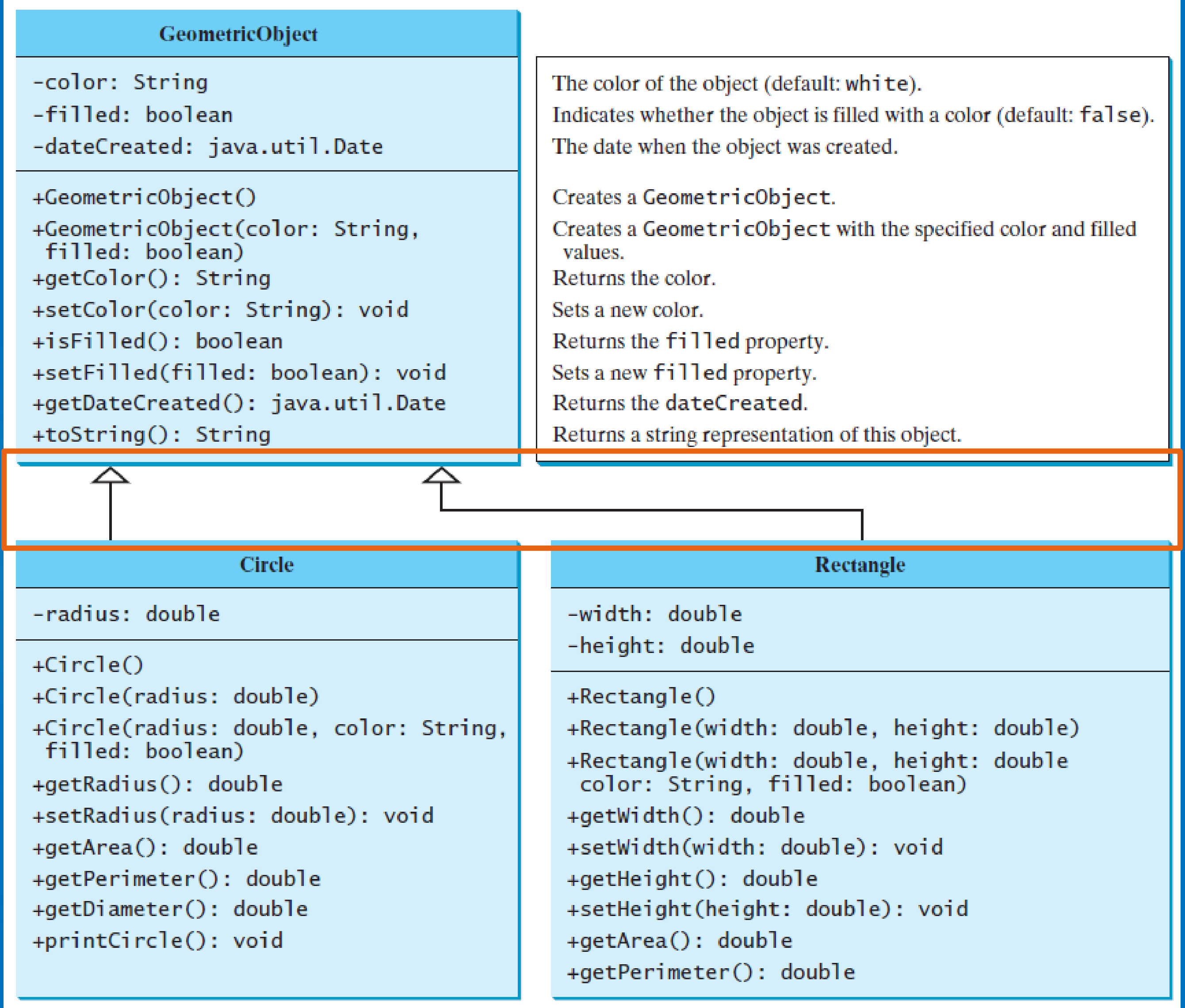


```
class Circle{
    private String color;
    private boolean filled;
    private double radius;
}
class Rectangle{
    private String color;
    private boolean filled;
    private double width;
    private double height;
}
```

Common Data Fields

```
class GeometricObject{
    private String color;
    private boolean filled;
}
```

Inheritance Relationship Definition



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

A class *C1 extended* from another class *C2* is called a *subclass*, and *C2* is called a *superclass*. A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*. A subclass *inherits accessible data fields* and *methods* from its superclass and may *also add new data fields and methods*.

Inheriting Using the Keyword *extends*

```
public class GeometricObject{
    private String color;
    public String getColor(){
        return this.color;
    }
}
public class Circle extends Geometry{
    private double radius; //new property
    public String getRadius(){  //new method
        return this.color;
    }
}
```

Access Members of *Super-classes* By Public Methods

A subclass *does not inherit* the *private* members of its parent class. *However*, if the superclass has *public* or *protected methods* for *accessing* its *private fields*, these can also be used by the subclass. For example:

```
public class Geometry{
    private String color;
    public String getColor(){
        return this.color;
    }
}
```

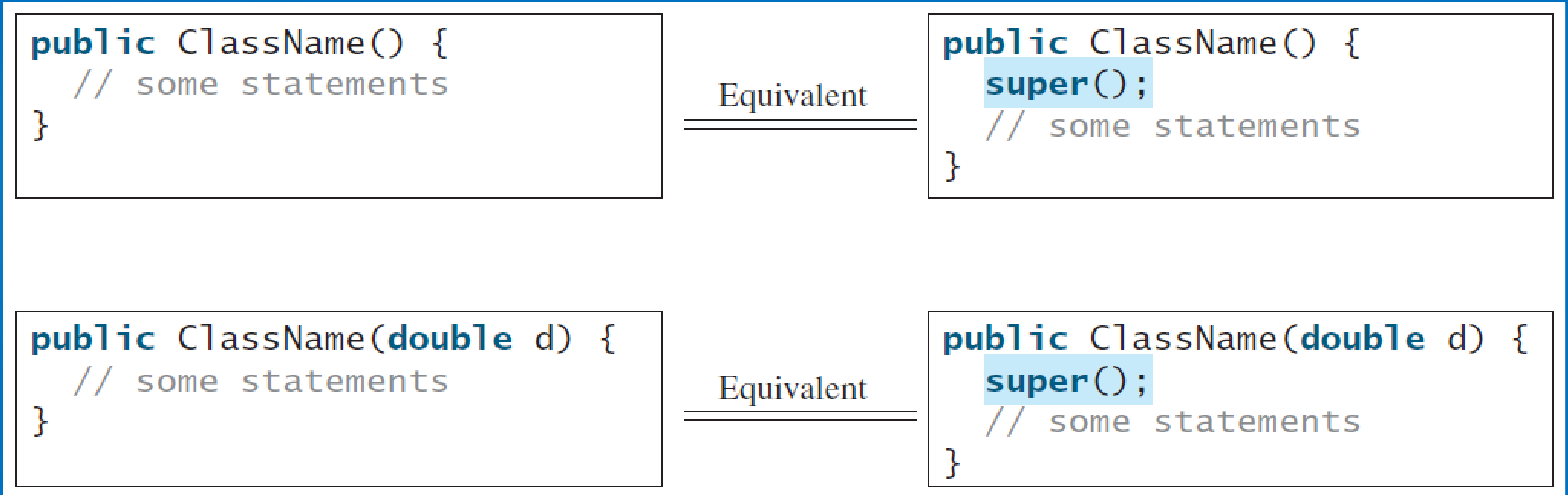
```
public class Circle extends Geometry{
    public String printColor(){
        System.out.print(getColor());
    }
}
```

Constructors of Super-classes

A constructor is used to construct an instance of a class. Unlike properties and methods, *constructors* of a superclass *are* ***not*** *inherited* in the subclass. They can only be *invoked from constructors* of *subclasses*, using the keyword *super*. If the keyword *super* is not explicitly used, the *no-arg constructor* of the *superclass* is automatically invoked.

Default Constructor Invoked

If none of them is invoked explicitly, the compiler puts *super*() as the first statement in the constructor. For example



```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

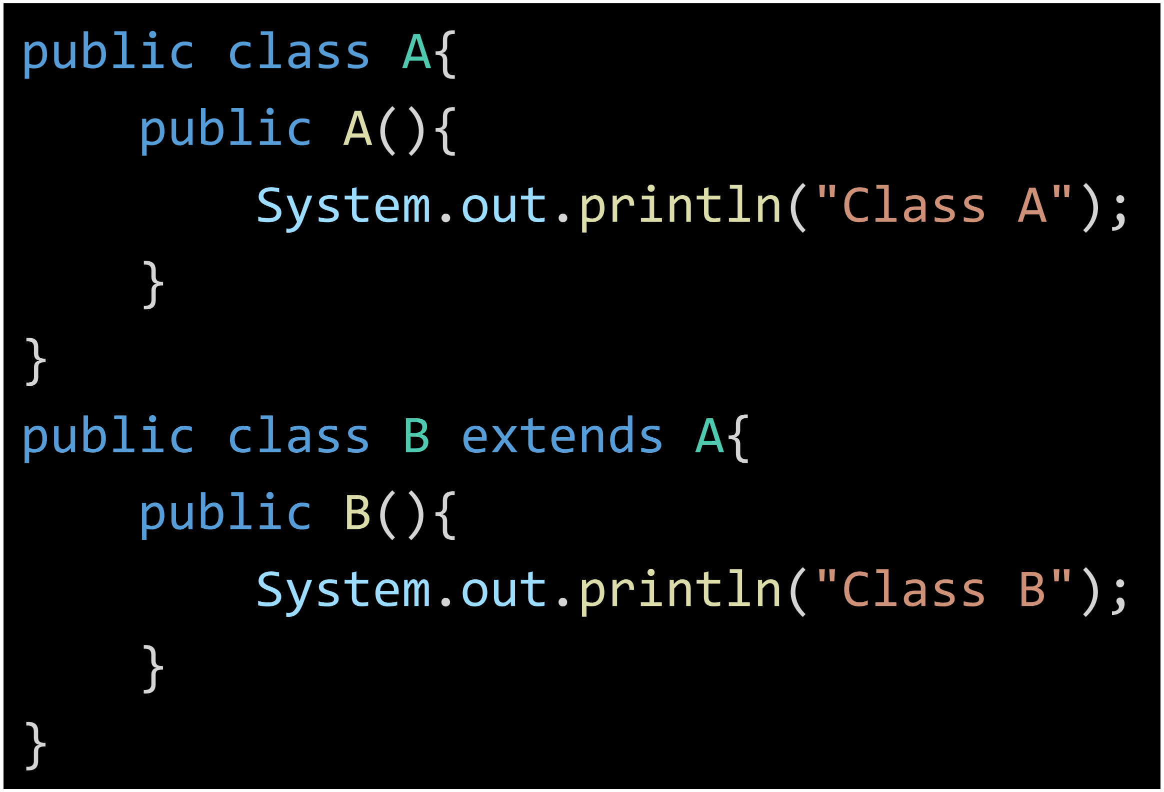
```
public ClassName(double d) {
  // some statements
}
```

Equivalent

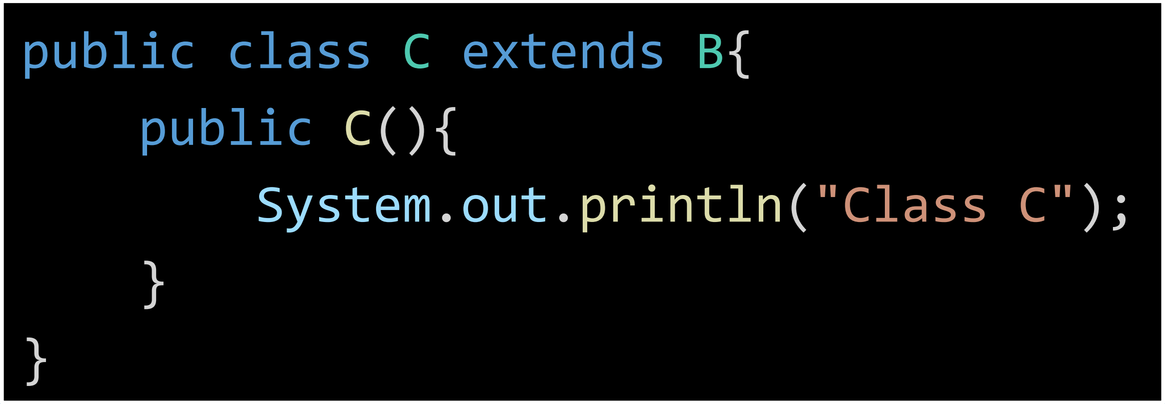
```
public ClassName(double d) {
  super();
  // some statements
}
```

Constructor Chaining

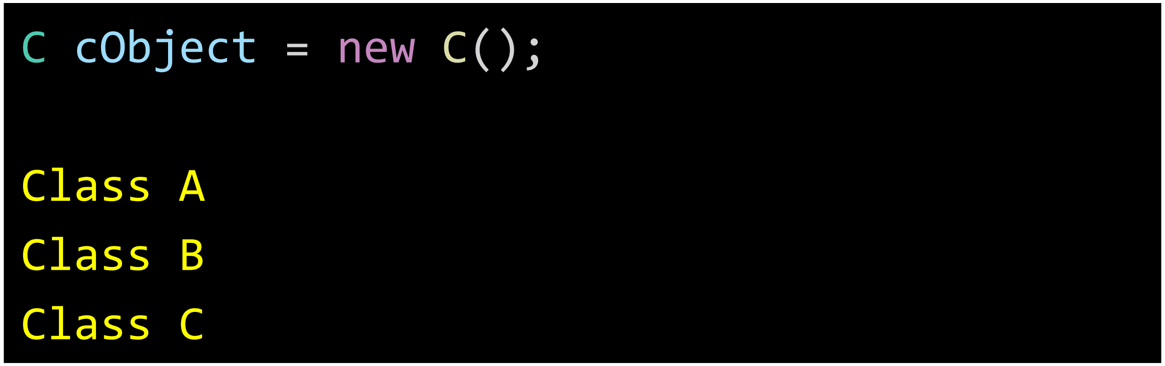
Constructing an instance of a class invokes all the constructors of its superclasses along the inheritance chain. This is called *constructor chaining*.



```
public class A{
    public A(){
        System.out.println("Class A");
    }
}
public class B extends A{
    public B(){
        System.out.println("Class B");
    }
}
```



```
public class C extends B{
    public C(){
        System.out.println("Class C");
    }
}
```



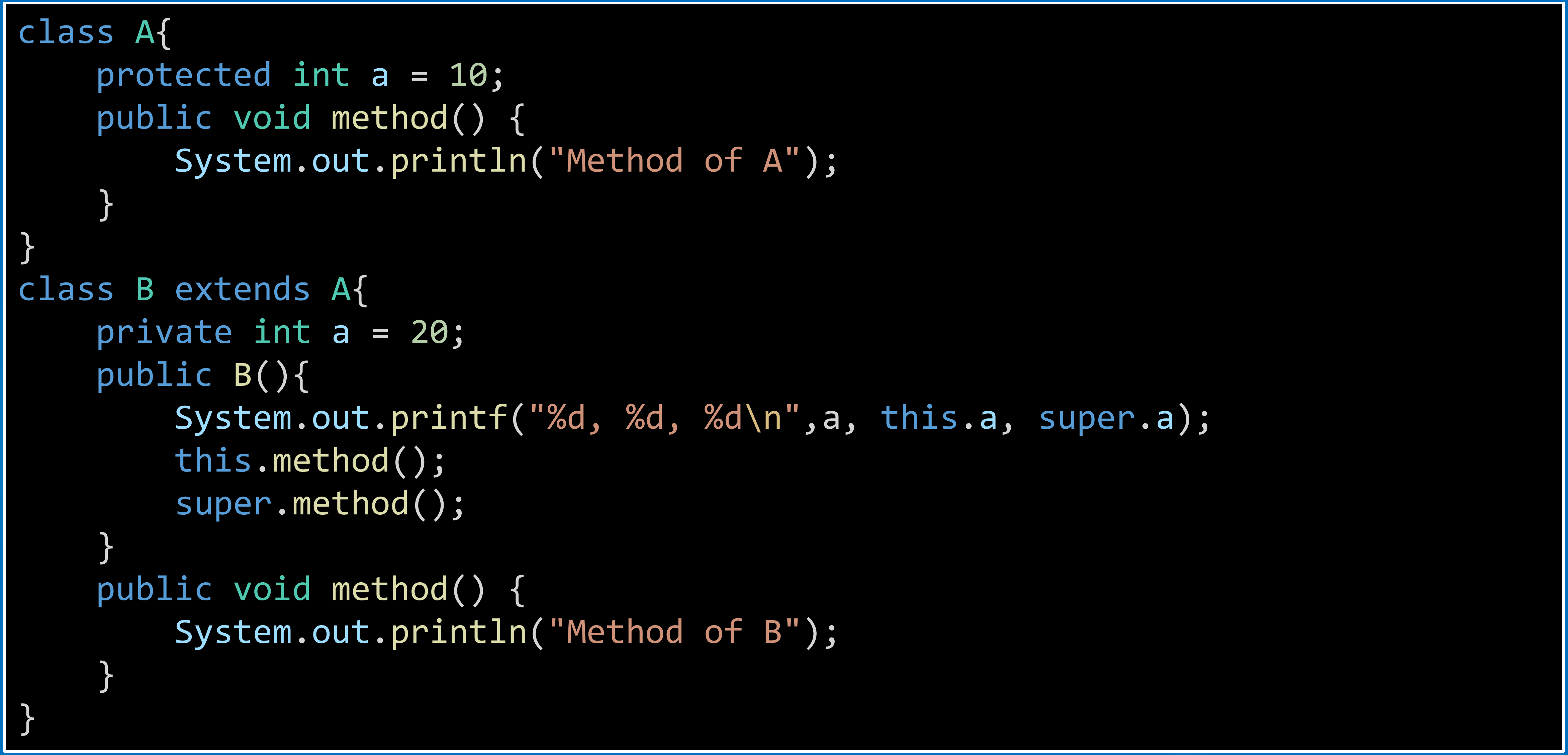
```
C cObject = new C();

Class A
Class B
Class C
```

The Impact of a Superclass without *no-arg* Constructor

```
class Circle extends GeometricObject{
    private double radius;
    public Circle() {}
    public Circle(double radius) {
        this.radius = radius;
    }
}
class GeometricObject{
    private String color;
    private boolean filled;
    public GeometricObject(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
    }
}
```

Compilation Error

Using the Keyword super to access superclass members



```
class A{
    protected int a = 10;
    public void method() {
        System.out.println("Method of A");
    }
}
class B extends A{
    private int a = 20;
    public B(){
        System.out.printf("%d, %d, %d\n",a, this.a, super.a);
        this.method();
        super.method();
    }
    public void method() {
        System.out.println("Method of B");
    }
}
```

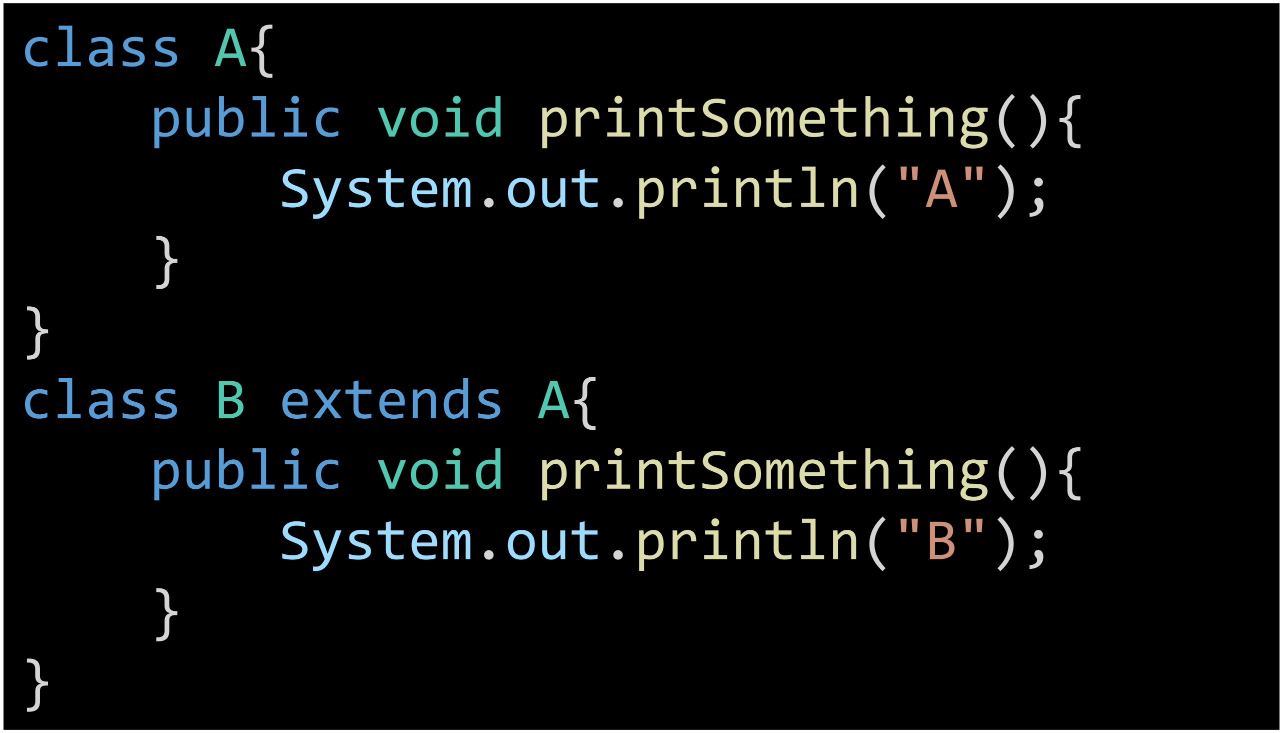
Exercises

What problem arises in compiling the following program.

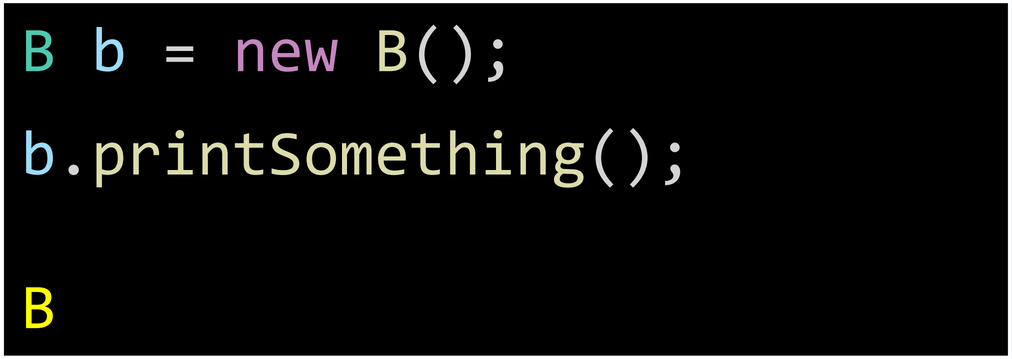
```
class A {
    public A(int x) {
    }
}
class B extends A {
    public B() {
    }
}
public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

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*.



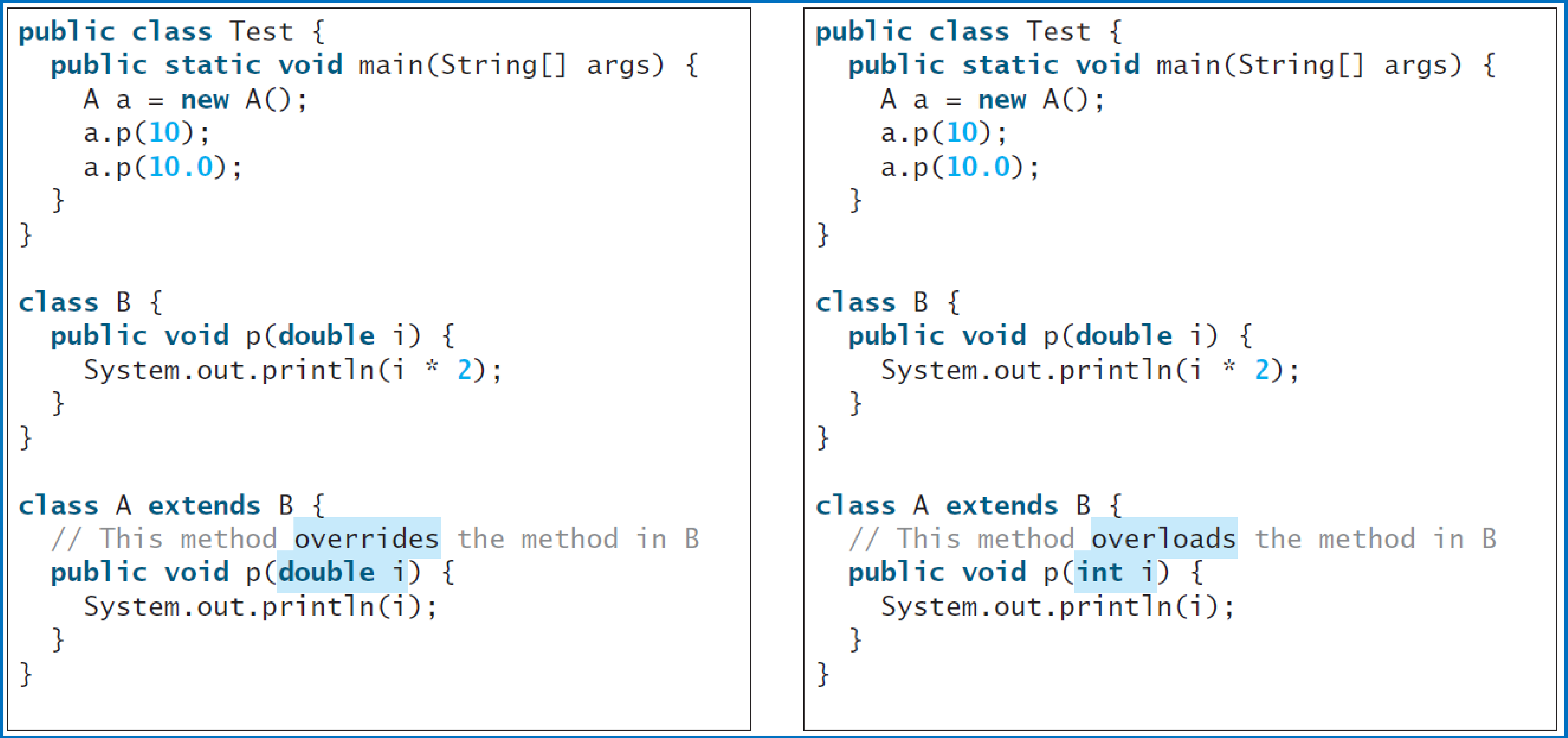
```java
class A{
    public void printSomething(){
        System.out.println("A");
    }
}
class B extends A{
    public void printSomething(){
        System.out.println("B");
    }
}
```



```java
B b = new B();
b.printSomething();
```

```
B
```

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);
  }
}
```

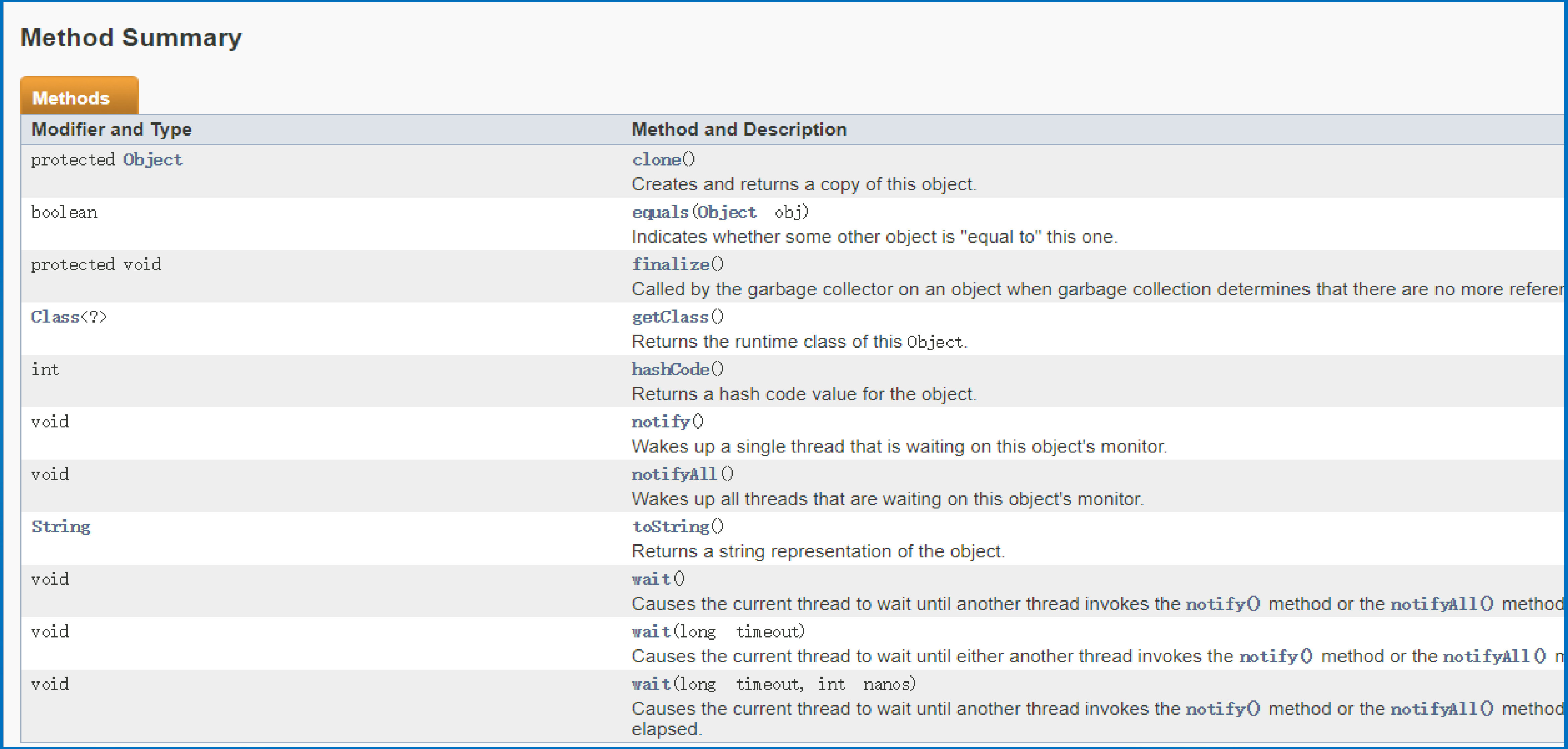
```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

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 Object Class

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*.

```java
public class A{
    public static void main(String[] args){
        A a = new A();
        a.printSuperClassName();
    }
    public void printSuperClassName(){
        System.out.println(this.getClass().getSuperclass().getName());
    }
}
```

The Methods of the Object Class



Method Summary

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more referen
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() m
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method elapsed.

The `toString()` method of the Object Class

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.

```
public class B extends A{
    public static void main(String[] args) {
        B a = new B();
        //getClass().getName() + '@' + Integer.toHexString(hashCode())
        System.out.println(a.toString());
    }
}
```

The code displays something like B@15db9742. 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.

Override equals Method of Object

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.

Override equals Method of Object

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 comparing areas of two *circles*, 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 for Overriding

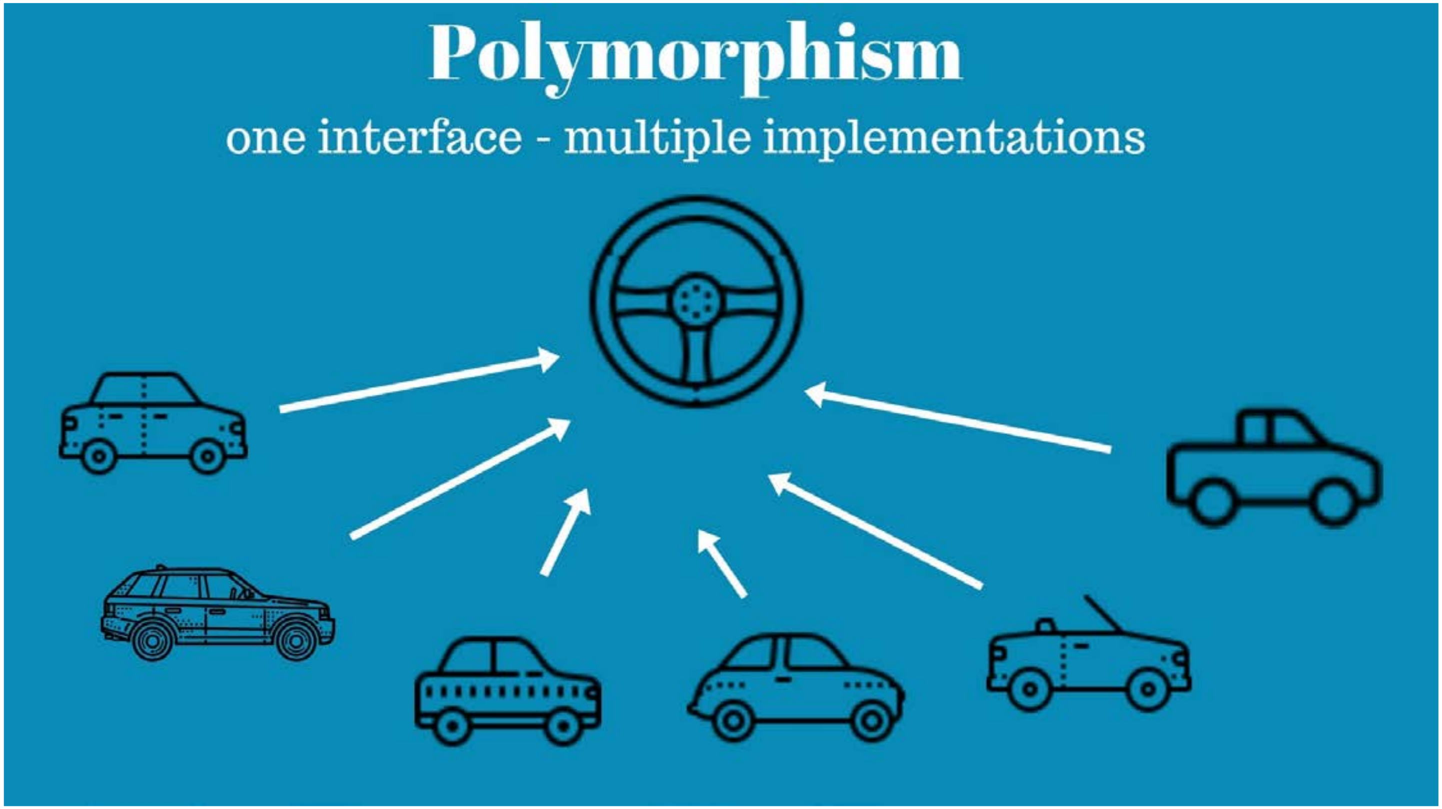
- An *instance method* can be *overridden* only if it is *accessible*. Thus a *private* method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

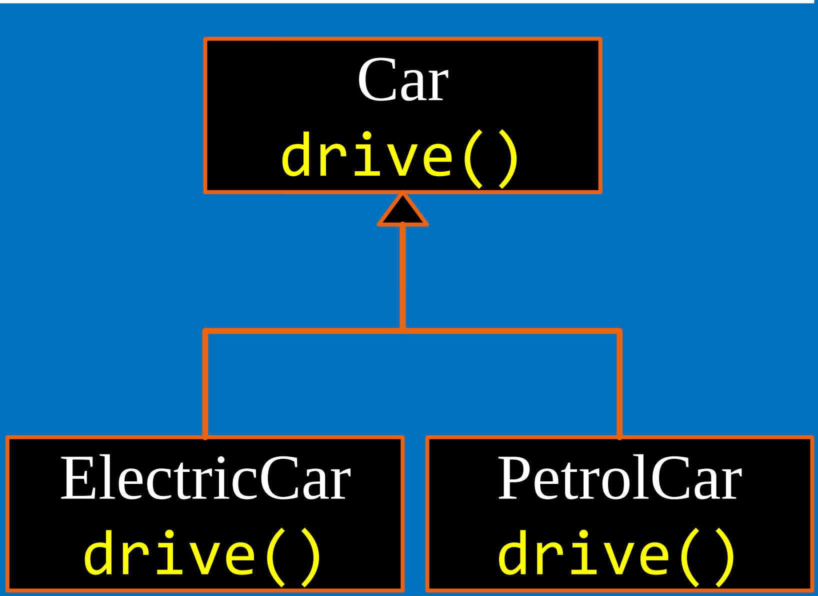
- Like an instance method, a *static method can* be *inherited*. However, a static method *cannot be overridden*. If a static method defined in the superclass is *redefined* in a subclass, the method defined in the superclass is *hidden*.

- 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.

Polymorphism

Polymorphism: Subclasses of a class can define their own *unique behaviors* and yet share some of the *same functionality* of the parent class. polymorphism deals with *decoupling* in terms of types.





Car
`drive()`

ElectricCar
`drive()`

PetrolCar
`drive()`

A Polymorphism Example: `toString()`

```
class A{
    public String toString(){
        return "toString method of Class A";
    }
}
class B extends A{
    public String toString(){
        return "toString method of Class B";
    }
}
Object o1 = new Object();
Object o2 = new A();
Object o3 = new B();
System.out.println(o1.toString()); // java.lang.Object@15db9742
System.out.println(o2.toString()); // toString method of Class A
System.out.println(o3.toString()); // toString method of Class B
```

Casting Objects (Upcasting)

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*. For example, the statement `Object o2 = new A()`, known as *implicit casting*, is legal because an instance of A is *automatically* an instance of Object.

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.

```
Object o = new Object();
A a = (A) o;  //error

Object o = new A();
A a = (A)o;  //correct
```

The `instanceof` Operator

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

```java
Object o1 = new Object();
Object o2 = new A();

if (o1 instanceof A) {
    System.out.println((A).toString());
}
if (o2 instanceof A) {
    System.out.println((A).toString());
}
```

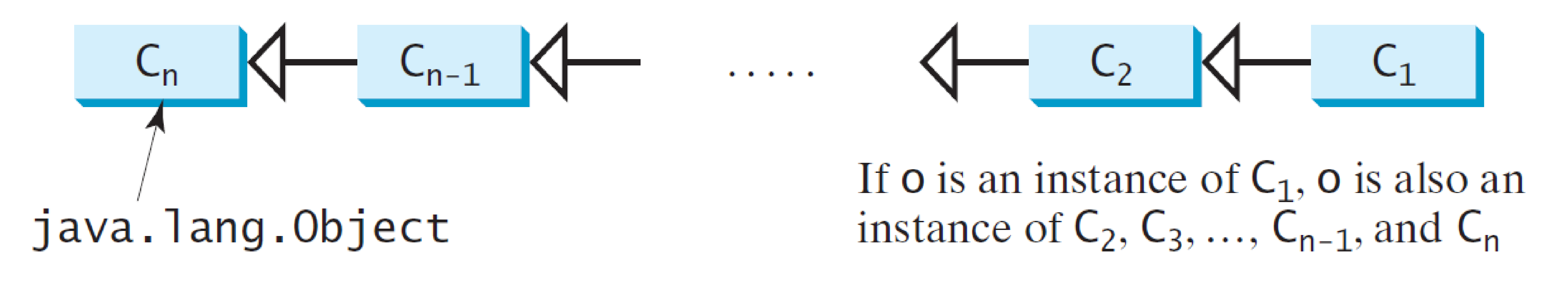
Understand Casting

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the `Fruit` class as the superclass for `Apple` and `Orange`. An apple is a fruit, so you can always safely assign an instance of `Apple` to a variable for `Fruit`. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of `Fruit` to a variable of `Apple`.

```
Apple a = new Apple();
Fruit f = a; //Assign an instance of Apple to a variable for Fruit
Apple b = (Apple)f; //Have to use explicit casting
```

Dynamic Binding: Which method to be invoked

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., 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, ..., 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.



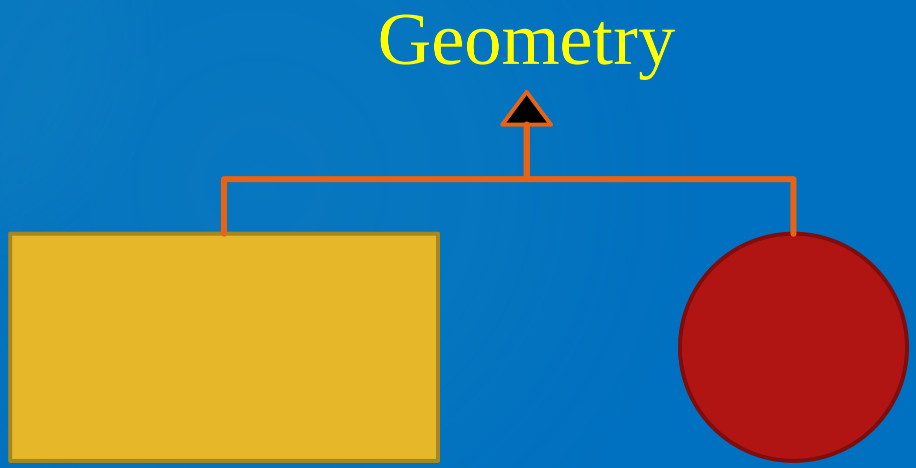
If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n

Method Matching vs. Dynamically 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*.

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.



An example: Answer questions using pen in exam

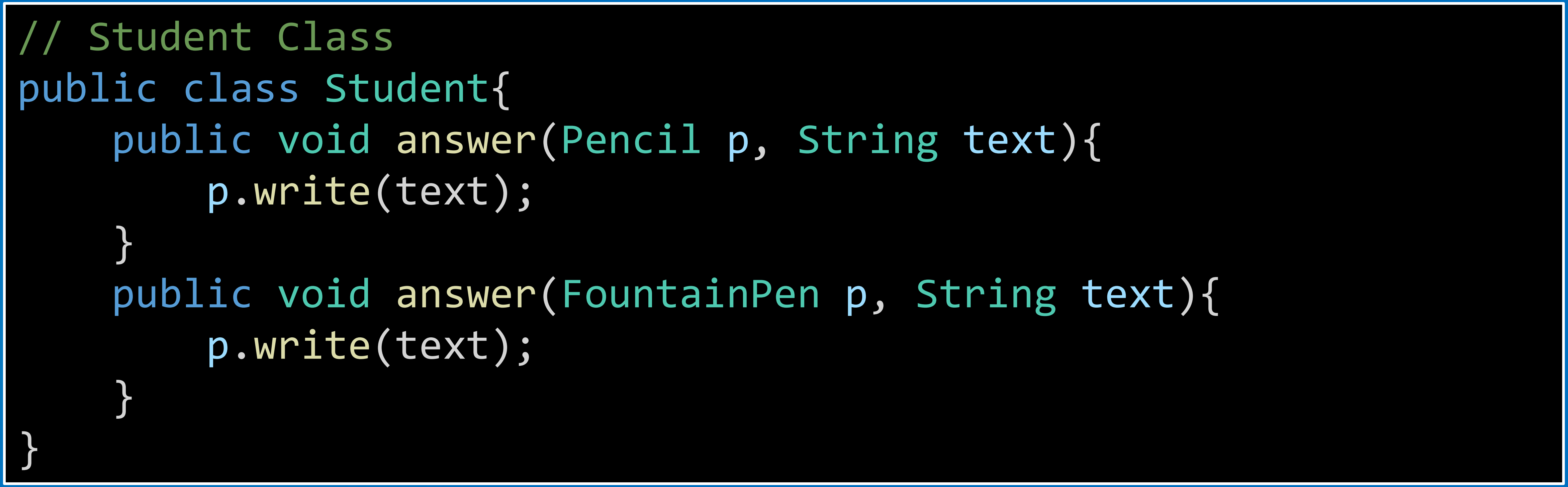
Fountain Pen

Pencil

A Common Solution

```
public class FountainPen{
    public void write(String text){
        System.out.println("Write [%s] with a fountain pen.", text);

    }
}

public class Pencil{
    public void write(String text){
        System.out.println("Write [%s] with a Pencil.", text);
    }
}
```

A Common Solution



```
// Student Class
public class Student{
    public void answer(Pencil p, String text){
        p.write(text);
    }
    public void answer(FountainPen p, String text){
        p.write(text);
    }
}
```

A Common Solution

```
// Simulation of a test process
public class Test{
    public static void main(String[] args){
        Student s = new Student();
        Pencil p = new Pencil();
        FountainPen fp = new FountainPen();
        s.answer(p, "C");
        s.answer(fp, "The result is 50.");
    }
}
```

Thinking

If there are more types of pen used for exam, how to update the program? Add new pen classes? If so, the `Student` class also needs to be modified. Obviously, it is not a good design. So how should we improve the program?

A Improved Design Using Polymorphism

```
public class Pen{
    // It is more suitable for defining a abstract method
    public void write(String text){}
}
public class FountainPen extends Pen{
    public void write(String text){
        System.out.println("Write [%s] with a fountain pen.", text);
    }
}
public class Pencil  extends Pen{
    public void write(String text){
        System.out.println("Write [%s] with a Pencil.", text);
    }
}
```

A Improved Design Using Polymorphism

```
public class Student{
    public void answer(Pen p, String text){
        p.write(text);
    }
}
public class Test{
    public static void main(String[] args){
        Student s = new Student();
        Pen p = new Pencil();
        Pen fp = new FountainPen();
        s.answer(p, "A");
        s.answer(fp, "The result is 666.");
    }
}
```

A Improved Design Using Polymorphism

If there is a new type of pen. We just need to define a new pen class. The Student class does not need to be modified.

```
public class NewPen extends Pen{
    public void write(String text){
        System.out.println("Write [%s] with a NewPen.", text);
    }
}
```

```
public static void main(String[] args){
    Student s = new Student();
    Pen np = new NewPen();
    s.answer(np, "There are something to be written.");
}
```

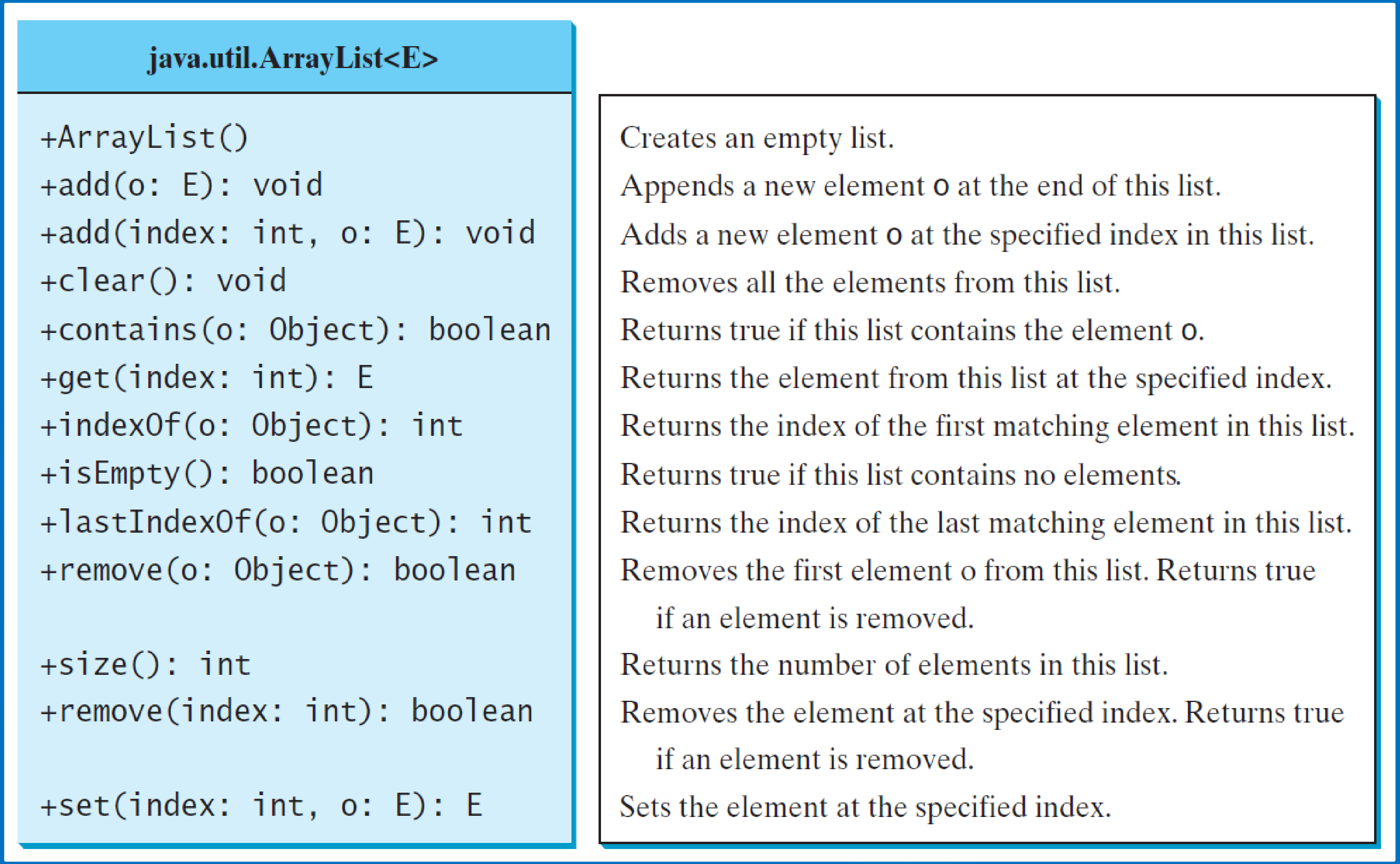
Preventing Extending and Overriding: `final`

The final class cannot be extended. The final variable is a constant. The final method cannot be overridden by its subclasses. For example:

```java
public final class Math {
    public static final double E = 2.718281828459045235;
}

class A{
    public final void method(){}
}
class B extends A{
    public void method(){} //Error
}
```

The ArrayList Class

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



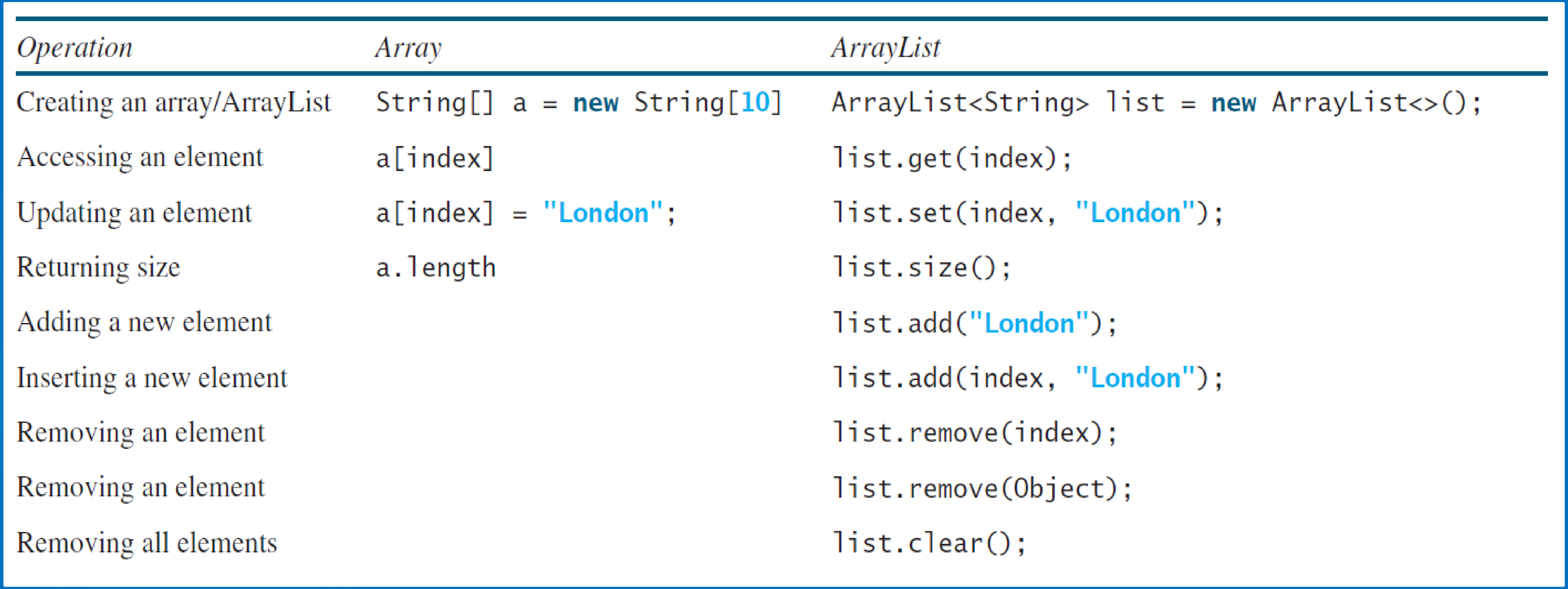
java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E): void`	Appends a new element `o` at the end of this list.
`+add(index: int, o: E): void`	Adds a new element `o` at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): 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 first element o from this list. Returns true if an element is removed.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int): boolean`	Removes the element at the specified index. Returns true if an element is removed.
`+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.

```
public class ArrayList<E> extends AbstractList<E>
     implements List<E>, RandomAccess, Cloneable, java.io.Serializable{
}

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

Arrays vs. 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();`

Array Lists from/to Arrays

Creating an *ArrayList* from an *array* of objects:

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

Creating an *array* of objects from an *ArrayList*:

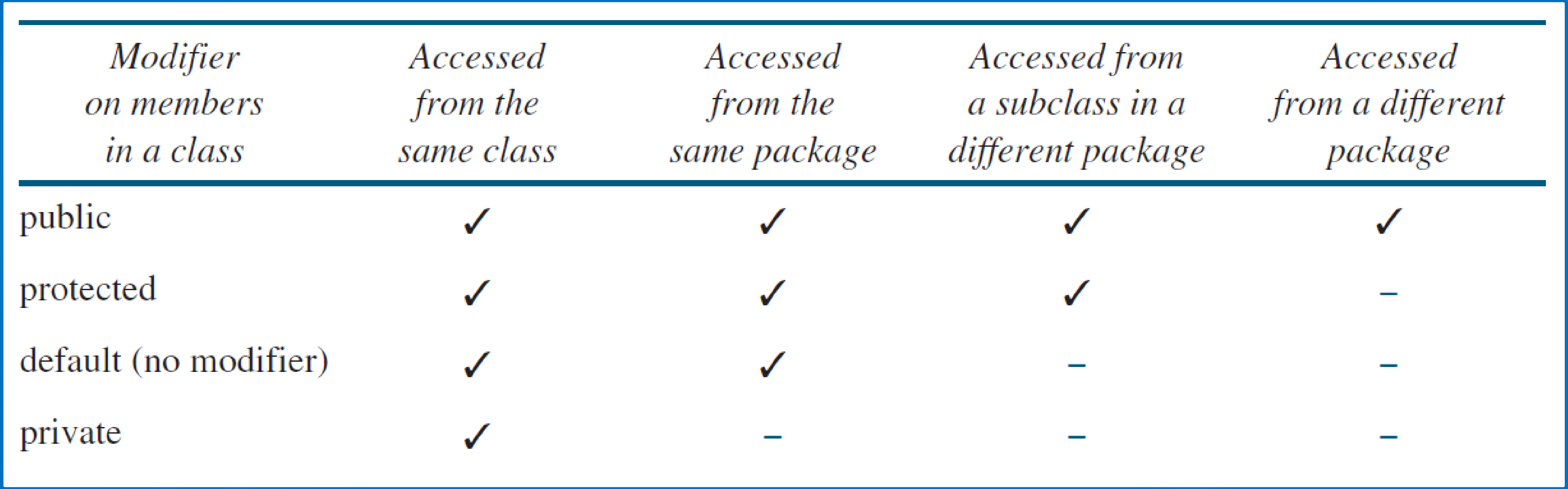
```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

ArrayList Processing by The Collections Class

```
// max and min
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<String>(Arrays.asList(array));
System.out.pritnln(java.util.Collections.max(list));
System.out.pritnln(java.util.Collections.min(list));

// shuffling
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);
```

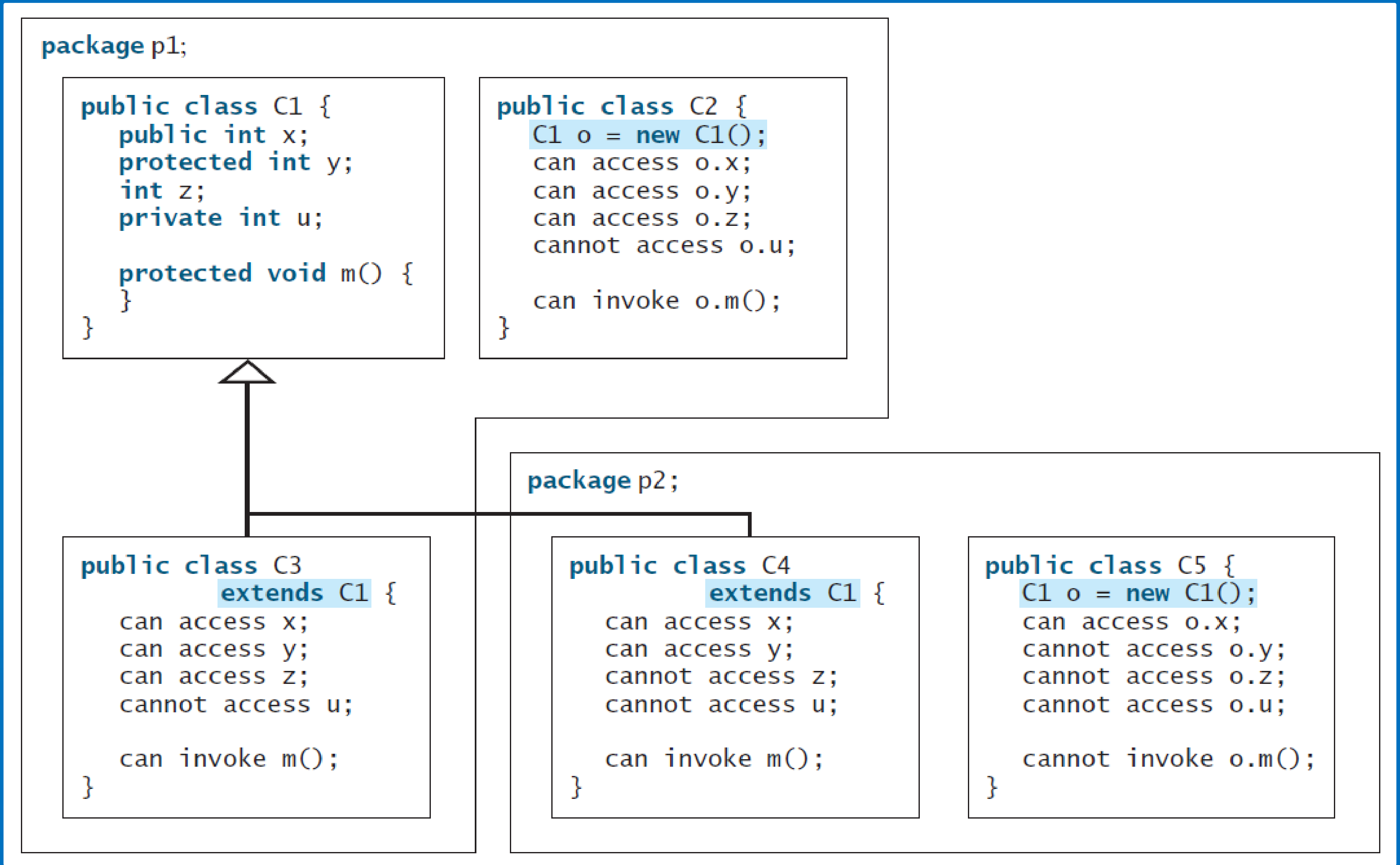
The `protected` Modifier

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.



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



```
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();
}
```

Make the members ***private*** if they are not intended for use from outside the class. Make the members ***public*** if they are intended for the users of the class. Make the fields or methods ***protected*** if they are intended for the extenders of the class but not for the users of the class.

Exercises

Can you assign `new int[50]`, `new Integer[50]`, `new String[50]`, or `new Object[50]`, into a variable of `Object[]` type?

True or false?

1. A subclass is a subset of a superclass.
2. When invoking a constructor from a subclass, its superclass's no-arg contructor is always invoked.
3. You can override a private method defined in a superclass.
4. You can override a static method defined in a superclass.

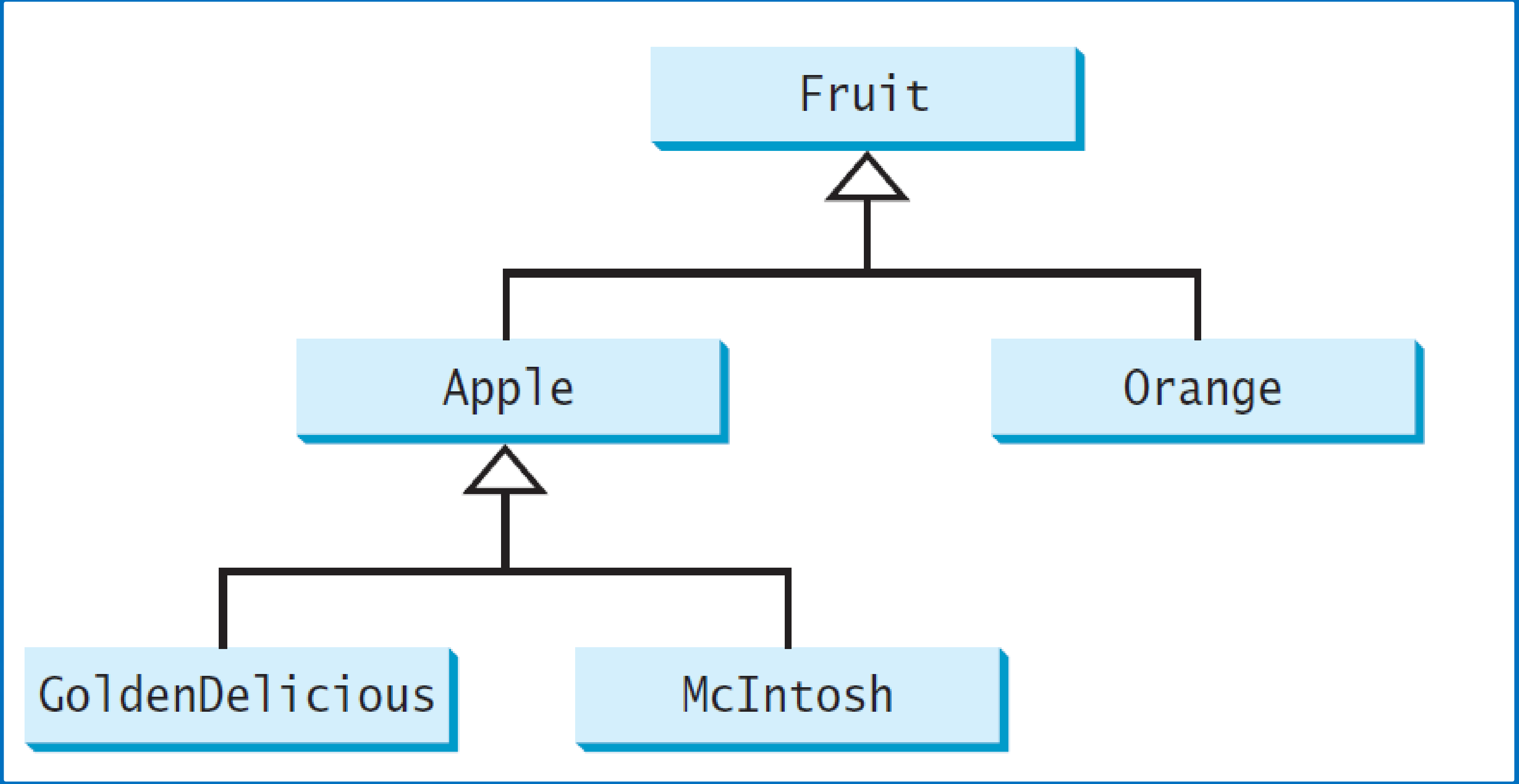
Exercises

Show the output of following program. Is the no-arg constructor of Object invoked when new A(3) is invoked?

```
public class Test{
    public static void main(String[] args){
        A a = new A(3);
    }
}
class A extends B {
    public A(int t) {System.out.println("A constructor is invoked");}
}
class B {
    public B() {System.out.println("B constructor is invoked");}
}
```

Exercises

Suppose that **Fruit**, **Apple**, **Orange**, **GoldenDelicious**, and **McIntosh** are defined in the following inheritance hierarchy:



```
Fruit fruit = new G
oldenDelicious();
Orange orange = new
 Orange();
```

Exercises

Answer the following questions:

a. Is **fruit `instanceof` Fruit**?

b. Is **fruit `instanceof` Orange**?

c. Is **fruit `instanceof` Apple**?

d. Is **fruit `instanceof` GoldenDelicious**?

e. Is **fruit `instanceof` McIntosh**?

f. Is **orange `instanceof` Orange**?

g. Is **orange `instanceof` Fruit**?

h. Is **orange `instanceof` Apple**?

i. Suppose the method **makeAppleCider** is defined in the **Apple** class. Can **fruit** invoke this method? Can **orange** invoke this method?

j. Suppose the method **makeOrangeJuice** is defined in the **Orange** class. Can **orange** invoke this method? Can **fruit** invoke this method?

k. Is the statement `Orange p = new Apple()` legal?

l. Is the statement `McIntosh p = new Apple()` legal?

m. Is the statement `Apple p = new McIntosh()` legal?

Practices

(*Sum **ArrayList***) Write the following method that returns the sum of all numbers in an **ArrayList**:

```
public static double sum(ArrayList<Double> list)
```

Write a test program that prompts the user to enter 5 numbers, stores them in an array list, and displays their sum.