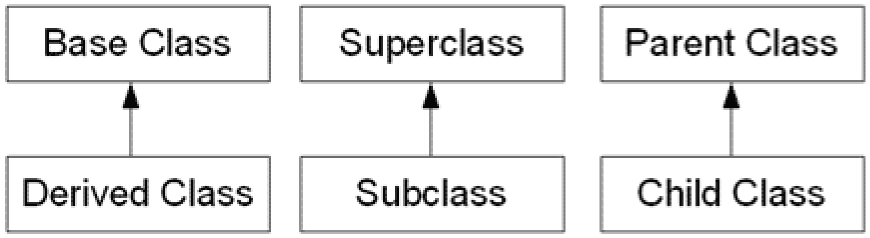
UNIVERSITÉ Concordia UNIVERSITY CENTRE FOR CONTINUING EDUCATION

INTRODUCTION TO PROGRAMMING WITH JAVA - CEJV416

Composition and Inheritance

Inheritance

- Inheritance is a relationship such that a new class is created based upon an existing class
- All of the public and protected attributes of the original class are inherited by the new class
- The new class can then redefine what it has inherited and/or add new members
- This is also called a generalization-specialization relationship
- The general class is called the base or parent or super class
- The specialized class is called the derived or child or sub class



Inheritance Concepts

- *Inheritance* lets you create a new class based on an existing class.
- The new class *inherits* the fields, constructors, and methods of the existing class.
- A class that inherits from an existing class is called a *derived class*, *child class,* or *subclass*.
- A class that another class inherits is called a *base class*, *parent class*, or *superclass*.

Derived Class

- You can directly access fields that have public or protected access in the superclass.
- You can extend the superclass by adding new fields, constructors, and methods.
- You can override methods in the superclass by coding methods that have the same signatures.
- You use the ***super*** keyword to call a constructor or method of the superclass. If necessary, you can call constructors or methods that pass arguments to the superclass.

The syntax for creating subclasses

To declare a subclass

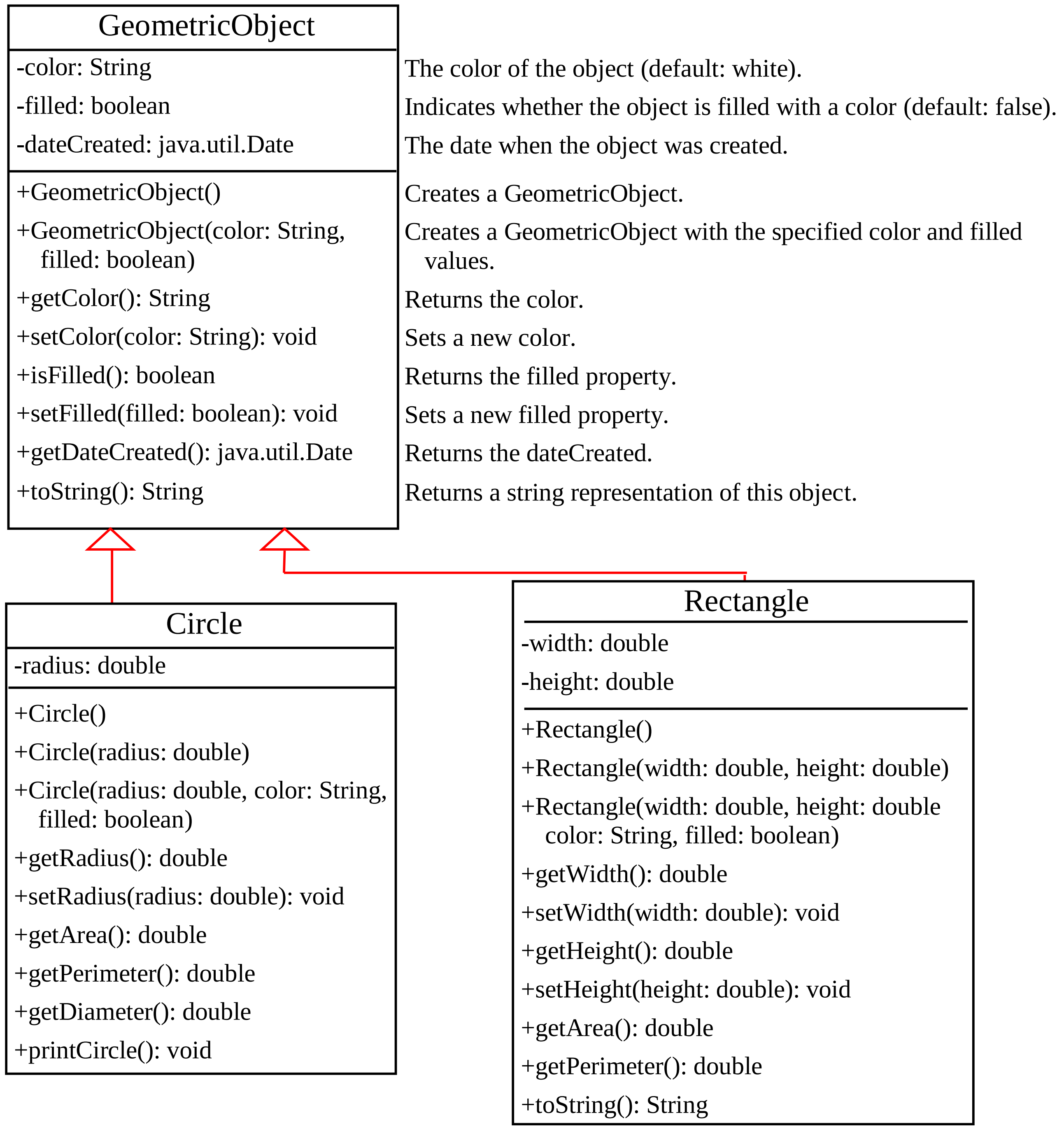
```
public class SubclassName extends SuperClassName{}
```

To call a superclass constructor

```
super(argumentList)
```

To call a superclass method

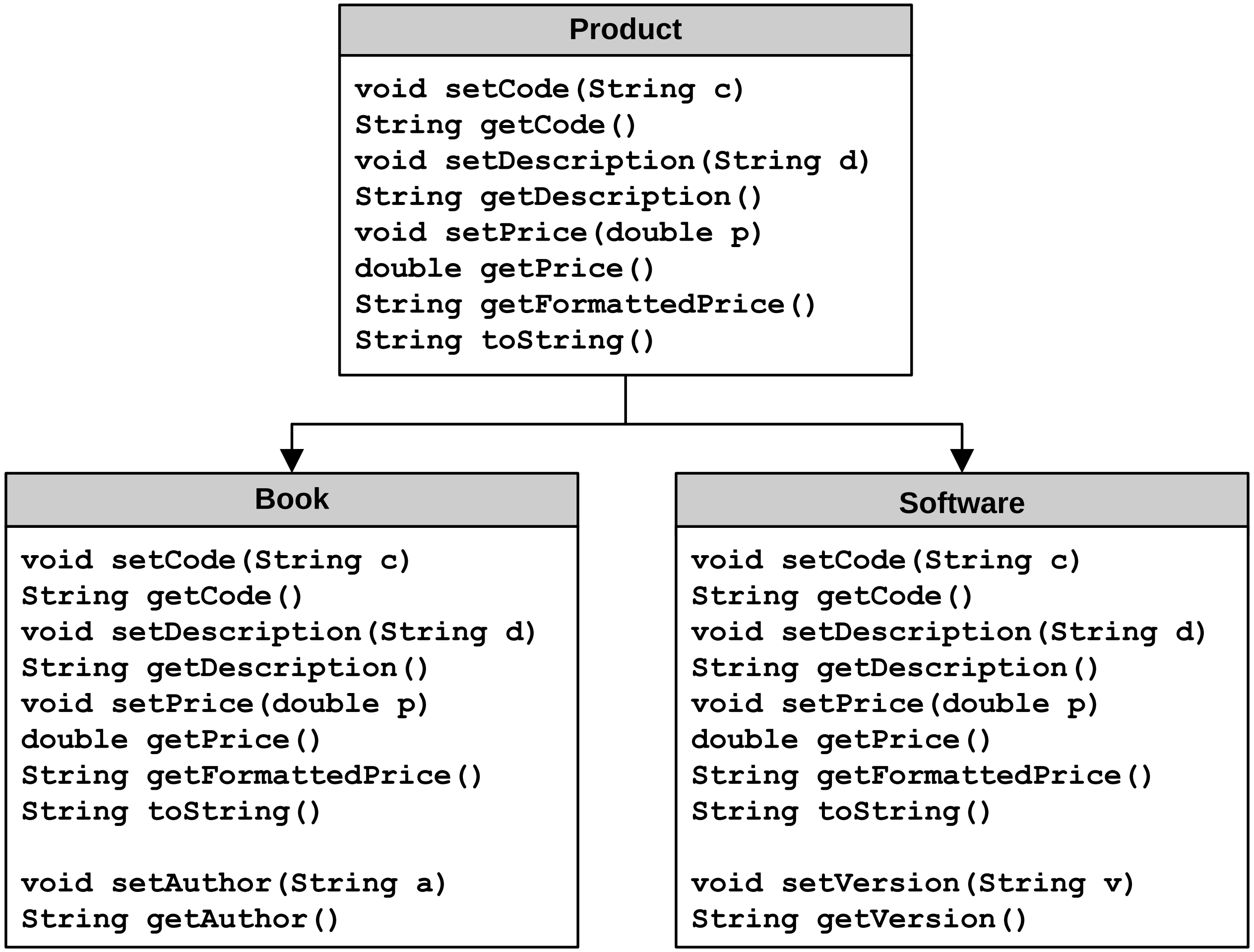
```
super.methodName(argumentList)
```


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
+toString(): String

How polymorphism works

- *Polymorphism* is a feature of inheritance that lets you treat objects of different subclasses that are derived from the same superclass as if they had the type of the superclass.
  - Example: If Book is a subclass of Product, you can treat a Book object as if it were a Product object.
- If you access a method of a superclass object and the method is overridden in the subclasses of that class, polymorphism determines which method is executed based on the object's type.
  - Example: If you call the toString method of a Product object, the toString method of the Book class is executed if the object is a Book object.

Business classes for a Product Maintenance application



Polymorphism: 3 versions of the toString method

The toString method in the Product superclass

```
public String toString()
{
    return "Code:        " + code + "\n" +
           "Description: " + description + "\n" +
           "Price:       " + this.getFormattedPrice() + "\n";
}
```

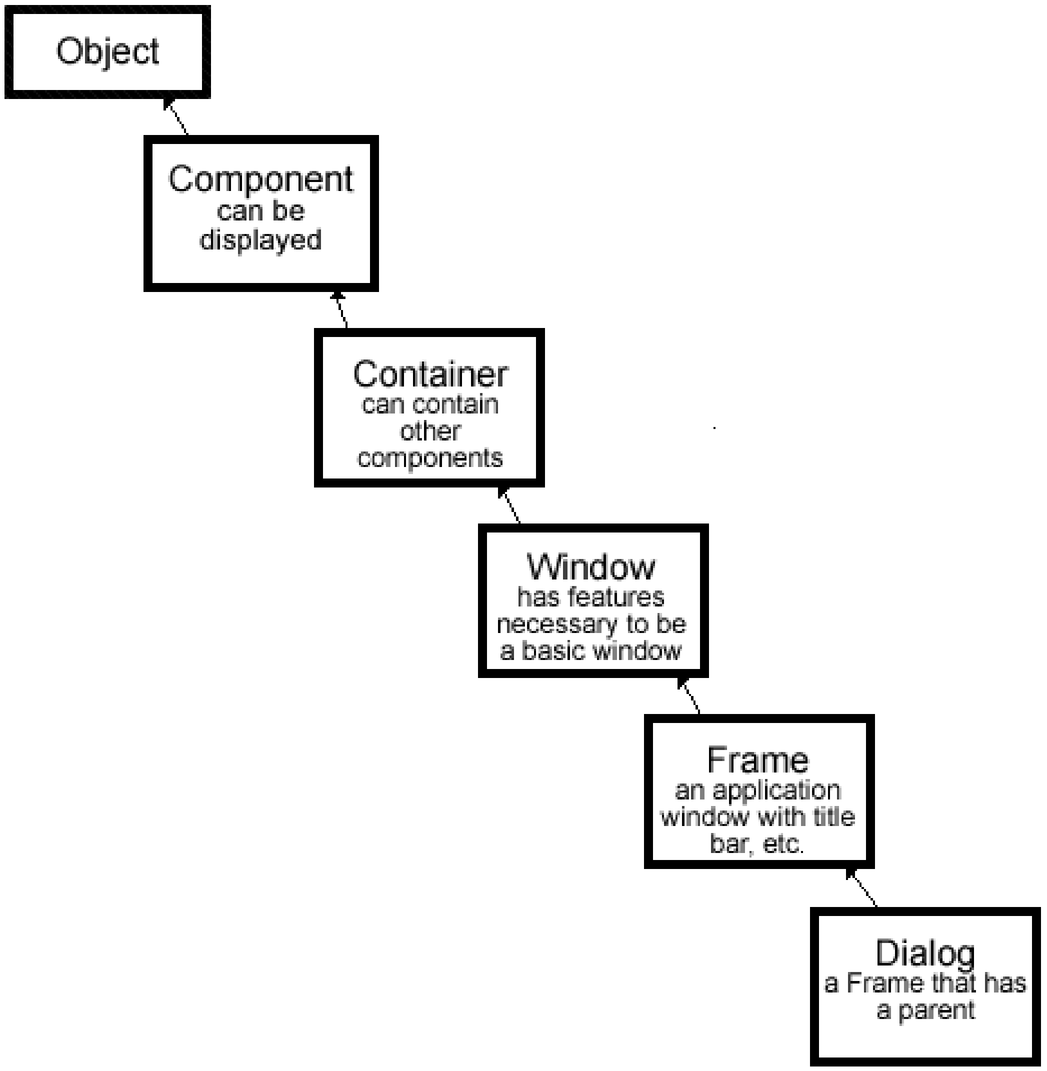
The toString method in the Book subclass

```
public String toString()
{
    return super.toString() +
        "Author:      " + author + "\n";
}
```

The toString method in the Software subclass

```
public String toString()
{
    return super.toString() +
        "Version:     " + version + "\n";
}
```

Java GUI Example


Object
Component can be displayed
Container can contain other components
Window has features necessary to be a basic window
Frame an application window with title bar, etc.
Dialog a Frame that has a parent

The Object class

```
java.lang.Object
```

How the Object class works

- The Object class in the java.lang package is the superclass for all classes. As a result, its methods are available to all classes.
- The *hash code* for an object is a hexadecimal number that identifies the object's location in memory.
- When creating classes, it's a common practice to override the toString and equals methods so they work appropriately for each class.
- In general, you don't need to override the finalize method for an object. That's because the *garbage collector* automatically reclaims the memory of an object whenever it needs to and before it does that, it calls the finalize method of the object.

Methods of the Object class

Method	Description
`toString()`	Returns a String object containing the class name, an @ symbol, and the object's hash code.
`equals(Object)`	Returns true (boolean) if this object points to the same space in memory as the specified object. Otherwise, it returns false, even if both objects contain the same data.
`getClass()`	Returns a Class object that represents the type of this object.
`clone()`	Returns a copy of this object as an Object object (the Cloneable interface must be implemented).
`hashCode()`	Returns the hash code (int) for this object.
`finalize()`	Called by the garbage collector when it determines that there are no more references to the object.

How to compare objects

- To test if two objects point to the same space in memory, you can use the equals method of the Object class.
- To test if two objects store the same data, you can override the equals method in the subclass so it tests whether all instance variables in the two objects are equal.

How the equals method of the Object class works

Example 1: Both variables refer to the same object

```
Product product1 = new Product();
Product product2 = product1;
if (product1.equals(product2)) // expression returns true
```

Example 2: Both variables refer to different objects that store the same data

```
Product product1 = new Product();
Product product2 = new Product();
if (product1.equals(product2)) // expression returns
                               // false
```

How to override the equals method of the Object class

The equals method of the Product class

```
@Override
public boolean equals(Object object)
{
    if (object instanceof Product)
    {
        Product product2 = (Product) object;
        if
        (
            code.equals(product2.getCode()) &&
            description.equals(
                product2.getDescription()) &&
            price == product2.getPrice()
        )
            return true;
    }
    return false;
}
```

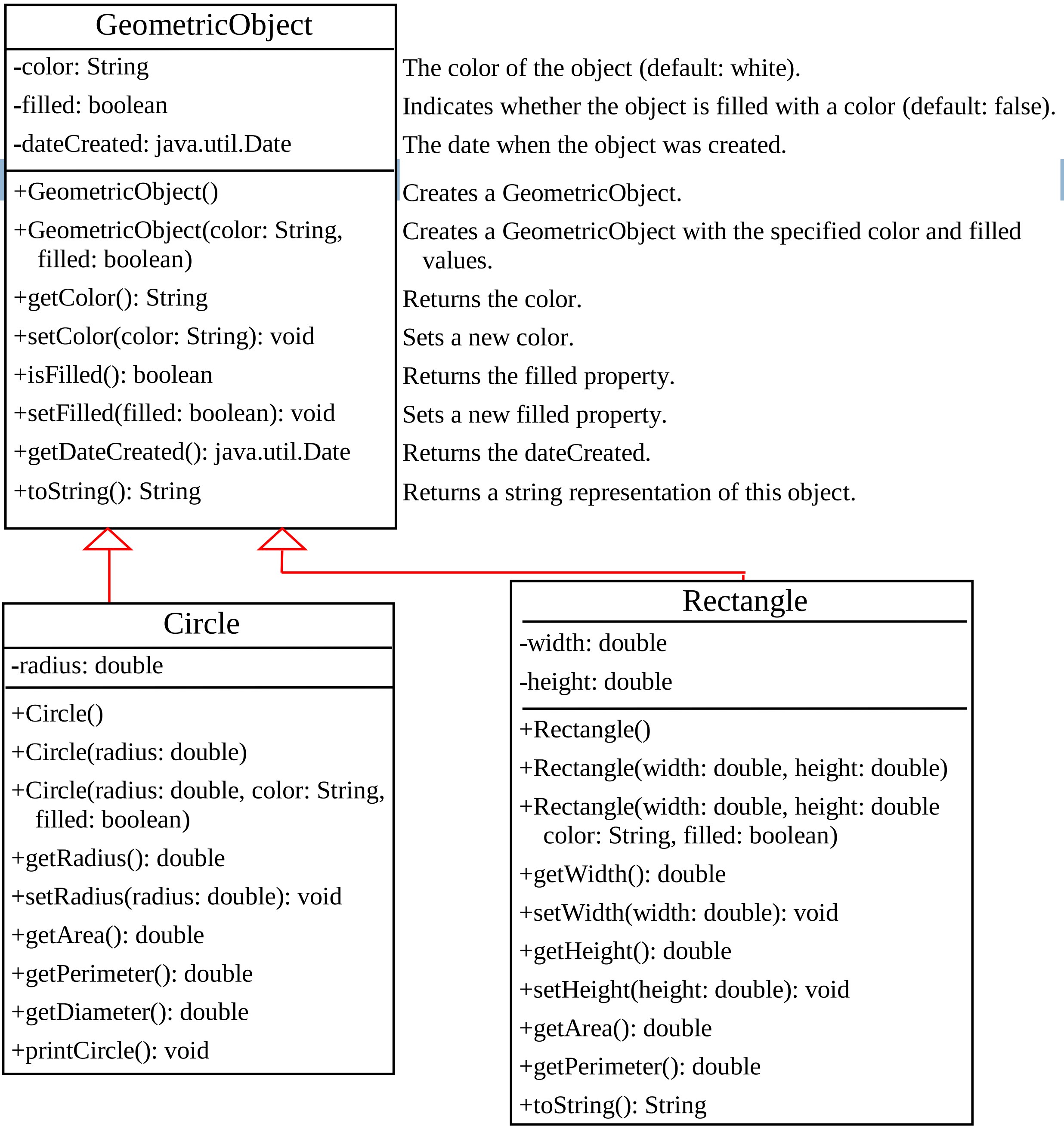
How to override the equals method of the Object class (cont.)

The equals method of the LineItem class

```
@Override
public boolean equals(Object object)
{
    if (object instanceof LineItem)
    {
        LineItem li = (LineItem) object;
        if
        (
            product.equals(li.getProduct()) &&
            quantity == li.getQuantity()
        )
            return true;
    }
    return false;
}
```

Exercise 21

Override the equals method 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
+toString(): String

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

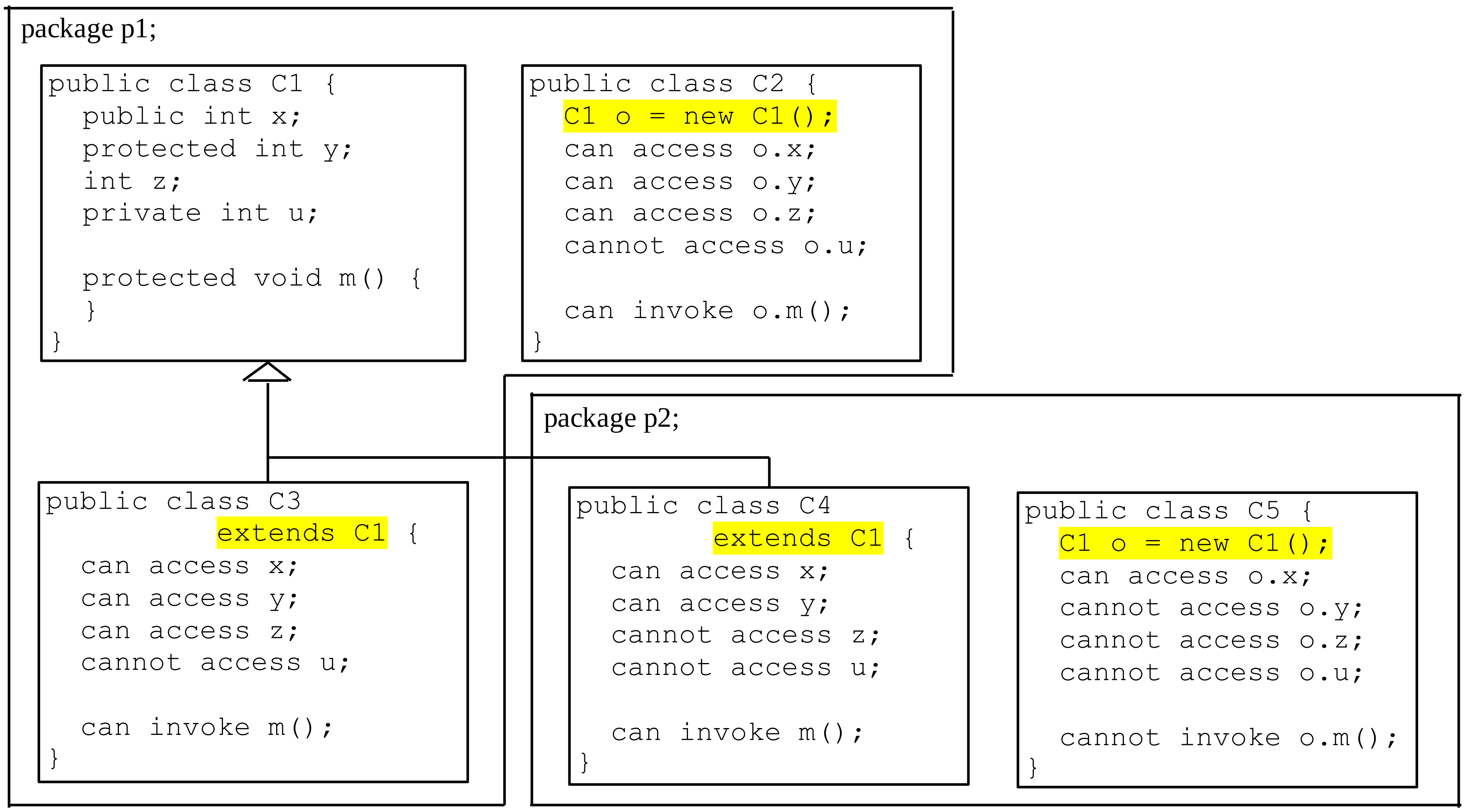
$$\xrightarrow{\texttt{Visibility increases}}$$

```
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();
}
```

How to work with the abstract keyword

- An *abstract class* is a class that can be inherited by other classes but that you can’t use to create an object.
- To declare an abstract class, code the abstract keyword in the class declaration.
- An abstract class can contain fields, constructors, and methods just like other superclasses. It can also contain abstract methods.
- To create an *abstract method*, you code the abstract keyword in the method declaration and you omit the method body.
- Abstract methods cannot have private access. However, they may have protected or default access (no access modifier).
- When a subclass inherits an abstract class, all abstract methods in the abstract class must be overridden in the subclass.
- Any class that contains an abstract method must be declared as abstract.

An abstract Product class

```
public abstract class Product
{
    private String code;
    private String description;
    private double price;

    // regular constructors and methods for instance
    // variables

    @Override
    public String toString()
    {
        return "Code:        " + code + "\n" +
               "Description: " + description + "\n" +
               "Price:       " + this.getFormattedPrice()
                             + "\n";
    }

    // an abstract method
    abstract String getDisplayText();
}
```

A class that inherits the abstract Product class

```
public class Book extends Product
{
    private String author;

    // regular constructor and methods for the Book class

    // implement the abstract method
    @Override
    public String getDisplayText()
    {
        return super.toString() +
            "Author:      " + author + "\n";
    }
}
```

A class that inherits the abstract Product class

```
public class Software extends Product
{
    private String version;

    // regular constructor and methods for the Software
class

    public String getDisplayText()
                         // implement the abstract method
    {
        return super.toString() +
            "Version is:      " + version + "\n";
    }
}
```

How to work with the final keyword

- To prevent a class from being inherited, you can create a *final class*.
- To prevent subclasses from overriding a method of a superclass, you can create a *final method*.
- To prevent a method from assigning a new value to a parameter, you can create a *final parameter*.
- All methods in a final class are automatically final methods.
- Coding the final keyword for classes and methods can result in a minor performance improvement because the compiler doesn’t have to allow for inheritance and polymorphism.

A final class

```
public final class Book extends Product
{
    // all methods in the class are automatically final
}
```

A final method

```
public final String getVersion()
{
    return version;
}
```

A final parameter

```
public void setVersion(final String version)
{
    // version = "new value"; // not allowed
    this.version = version;
}
```

Exercise 22

- Make the GeometricObject as an abstract class.
- Add the following abstract methods to the GeometricObject class:
  - getArea()
  - getPrimeter()
  - toString()
  - getFilledColor().
    - getFilledColor() returns the color of the object if it is filled, otherwise it will return “not filled”.
- Remove the errors in all your classes by adding appropriate methods.