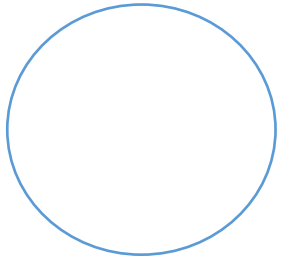
Inheritance & Polymorphism

CSGE601021 Dasar-Dasar Pemrograman 2

Fakultas Ilmu Komputer Universitas Indonesia

We now have some new objects



- color
- filled
- diameter

- color
- filled
- length
- height

Circle
color: String filled: boolean diameter:double
Circle()

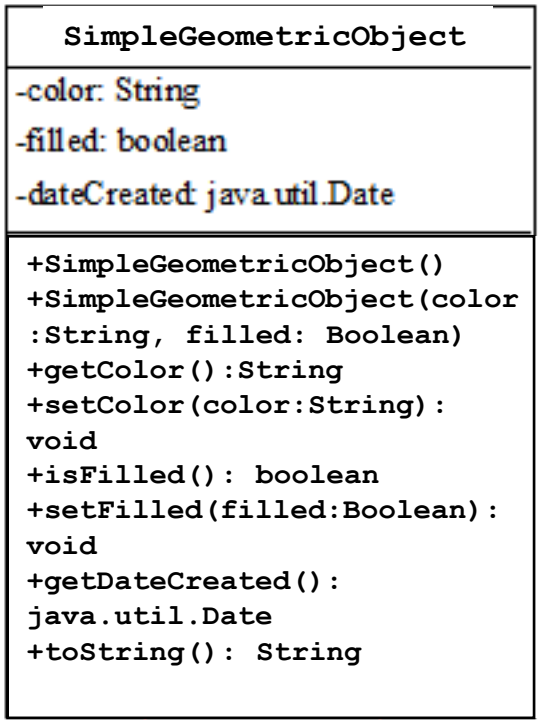
Rectangle
color: String filled: boolean length:double height:double
Rectangle()

These classes have common features.
What is the best way to design these classes so to avoid redundancy?

Inheritance

SimpleGeometricObject

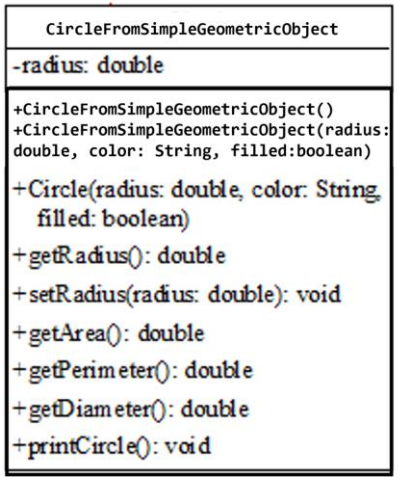
www.cs.armstrong.edu/liang/intro10e/html/SimpleGeometricObject.html




```java
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;
```

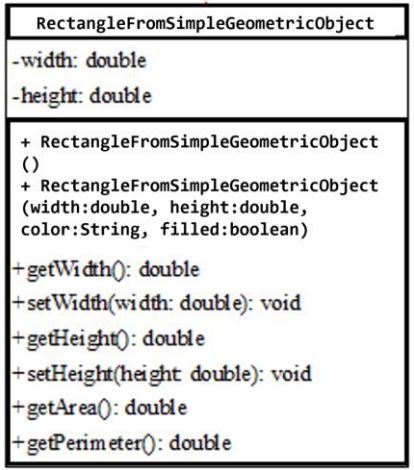
Subclasses of SimpleGeometricObject

https://liveexample.pearsoncmg.com/html/CircleFromSimpleGeometricObject.html



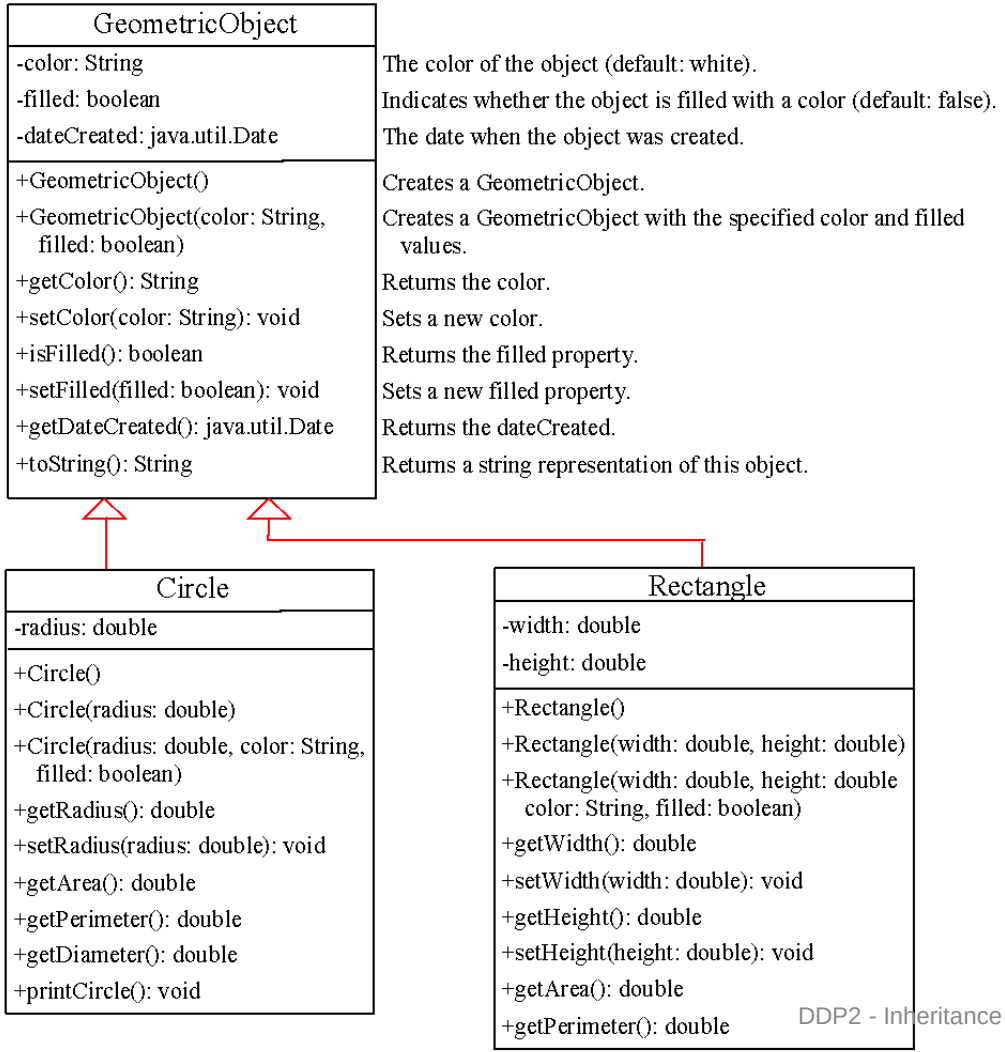

```
public class CircleFromSimpleGeometricObject
extends SimpleGeometricObject {
  private double radius;
```

https://liveexample.pearsoncmg.com/html/RectangleFromSimpleGeometricObject.html



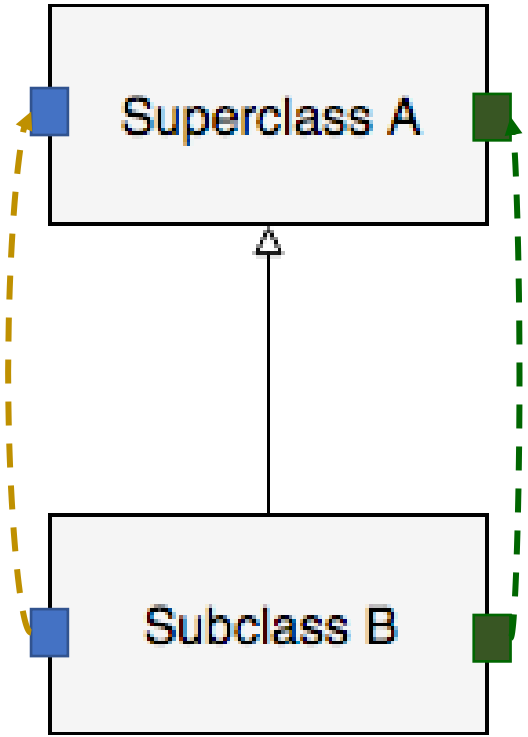

```
public class RectangleFromSimpleGeometricObject
extends SimpleGeometricObject {

  private double width;
  private double height;
```

UML Diagram


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

Inherits all properties and methods



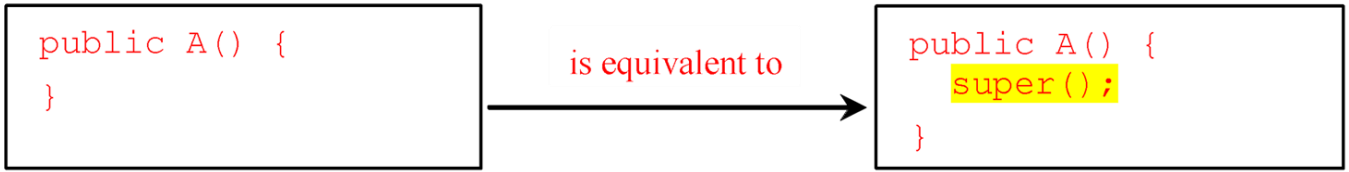
- The superclass's constructors are **not inherited**.
- They can be invoked from the subclasses' constructors, using the keyword **`super`**.

Constructors

- Unlike properties and methods, the superclass’s constructors are not inherited.
- They can only be invoked from the subclasses' constructors, using the keyword super. The call to super must be the first in the subclasses’ constructor.
- If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.

Superclass's Constructor Is Always Invoked

- **Implicitly**. If the keyword **`super()`** is not explicitly used, the superclass's no-arg constructor is automatically invoked. The compiler puts **`super()`** as the first statement in the constructor.



- **Explicitly:** using the keyword **`super()`**. The call to super **must be the first** in the subclasses' constructor.

CircleFromSimpleGeometricObject Constructor (2)

```
public CircleFromSimpleGeometricObject() {
}
```

- The superclass's no-arg constructor is invoked.
- The subclass attribute radius is set to default

```
public CircleFromSimpleGeometricObject(double radius) {
    this.radius = radius;
}
```

- The superclass's no-arg constructor is invoked.
- Sets the subclass attribute radius

CircleFromSimpleGeometricObject Constructor (2)

```
public CircleFromSimpleGeometricObject(double radius,
   String color, boolean filled) {
   super(color,filled);
   this.radius = radius;
}
```

- The superclass's constructor is invoked explicitly
- Sets the subclass attribute radius

```
public CircleFromSimpleGeometricObject(double radius,
   String color, boolean filled) {
   this.radius = radius;
   setColor(color);
   setFilled(filled);
}
```

- The superclass's no-arg constructor is invoked
- Sets the subclass attribute radius
- The superclass attributes are changed via setter

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.

```
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");
  }
}
```

Superclass without no-arg Constructor

- What would happen?

```
public class Apple extends Fruit {
}

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

Defining a Subclass

- A subclass inherits from a 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 follow:

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

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.

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


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

Some Notes!

- An instance method can be overridden only if it is **accessible**. **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.

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.

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

Equivalent

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

Recall: Printing Objects

- The toString() method returns a string representation of the object.
- The default 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@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.

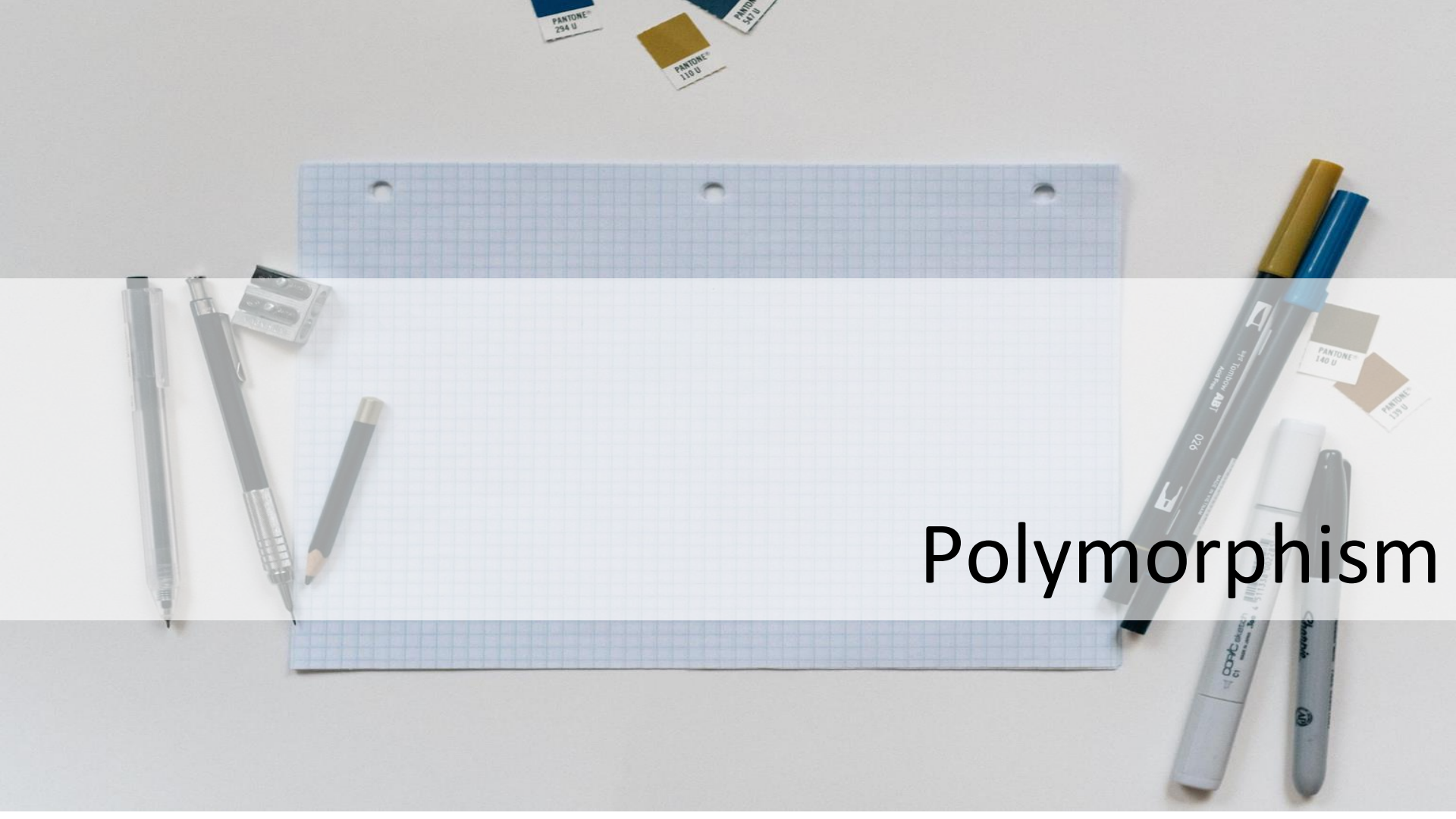
Recall: Printing Objects

```
Cube cube1 = new Cube();
Cube cube2 = new Cube("Blue", 4.0);
System.out.println(cube1);
System.out.println(cube2);
```

Printing objects calls the `toString()`method!

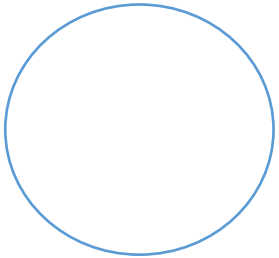
Output:
Cube@15db9742
Cube@6d06d69c

- By default, your class will call the `toString()`**of the `Object` Class**.
- The default returns a string consisting of a class name of which the object is an instance, the at sign (@), and the hashcode representing this object.
- We **usually** override the `toString()` method so that it returns a meaningful representation of the object.



Polymorphism

Polymorphism



Polymorphism is the ability to assume different forms or shapes.

- 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 means that a variable of a supertype can refer to a subtype object.

Polymorphism, Dynamic Binding and Generic Programming

What is the output?

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

Polymorphism, Dynamic Binding and Generic Programming

What is the output?

```
Student
Student
Person
java.lang.Object@15db9742
```

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

Polymorphism

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

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 is known as **polymorphism**

Dynamic Binding

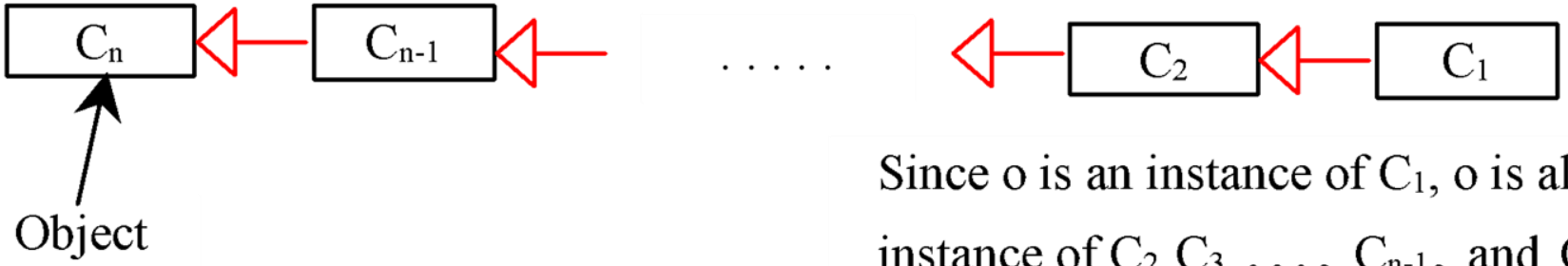
```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
```

When the method m(Object x) is executed, the argument x's toString method is invoked.

- Which implementation is used will be determined dynamically by JVM at runtime. This is known as dynamic binding.

Dynamic Binding

- Suppose object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_n is the most general class, and C_1 is the most specific 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.



Since 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. 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.

Generic Programming

Polymorphism allows methods to be used generically for different object arguments.

- This is known as **generic programming.**

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

Casting Objects

- Casting can be used to convert an object of one class type to another within an inheritance hierarchy.
- Recall:

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

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

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

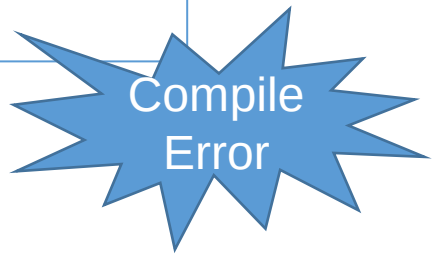
```
Object o = new Student();
m(o);
```

Implicit casting

Why Casting?

- Assigning the object reference o to a variable of the Student type:

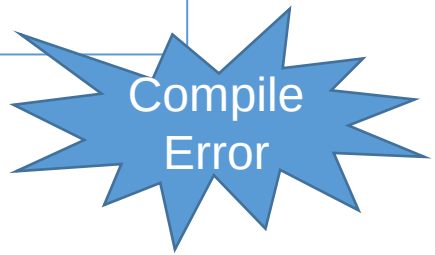
```
Student b = o;
```



Why Casting?

- Assigning the object reference o to a variable of the Student type:

```
Student b = o;
```



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

Why Casting?

- Assigning the object reference o to a variable of the Student type:

```
Student b = o;
```

- Why does the statement **Object o = new Student()** work but the statement **Student b = o** doesn't?
- Because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

Explicit Casting

- To tell the compiler that o is a Student object, use an explicit casting.

```
Student b = (Student)o;
```

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

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

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());
  ...
}
```

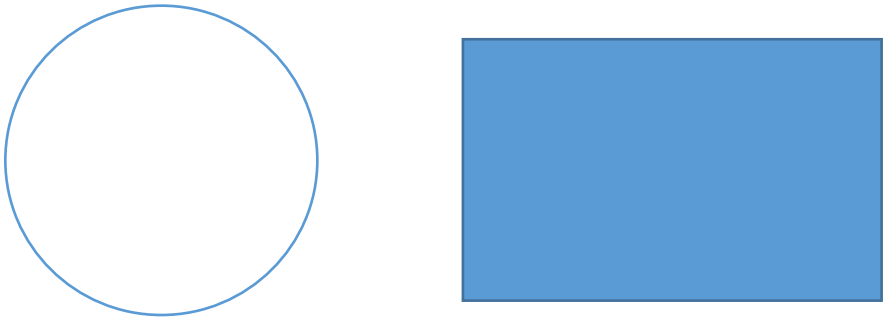
Note

- 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.

Example: Polymorphism and Casting

https://liveexample.pearsoncmg.com/html/CastingDemo.html

- 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.



The `equals` Method

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

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

- The `equals()` method can be overridden (for example in the Circle class)

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

Recall ==

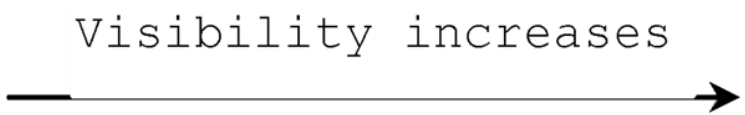
- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.
- The equals method is intended to test whether two objects **have the same contents,** provided that the method is modified in the defining class of the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

Visibility Modifier

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	✓	–	–	–

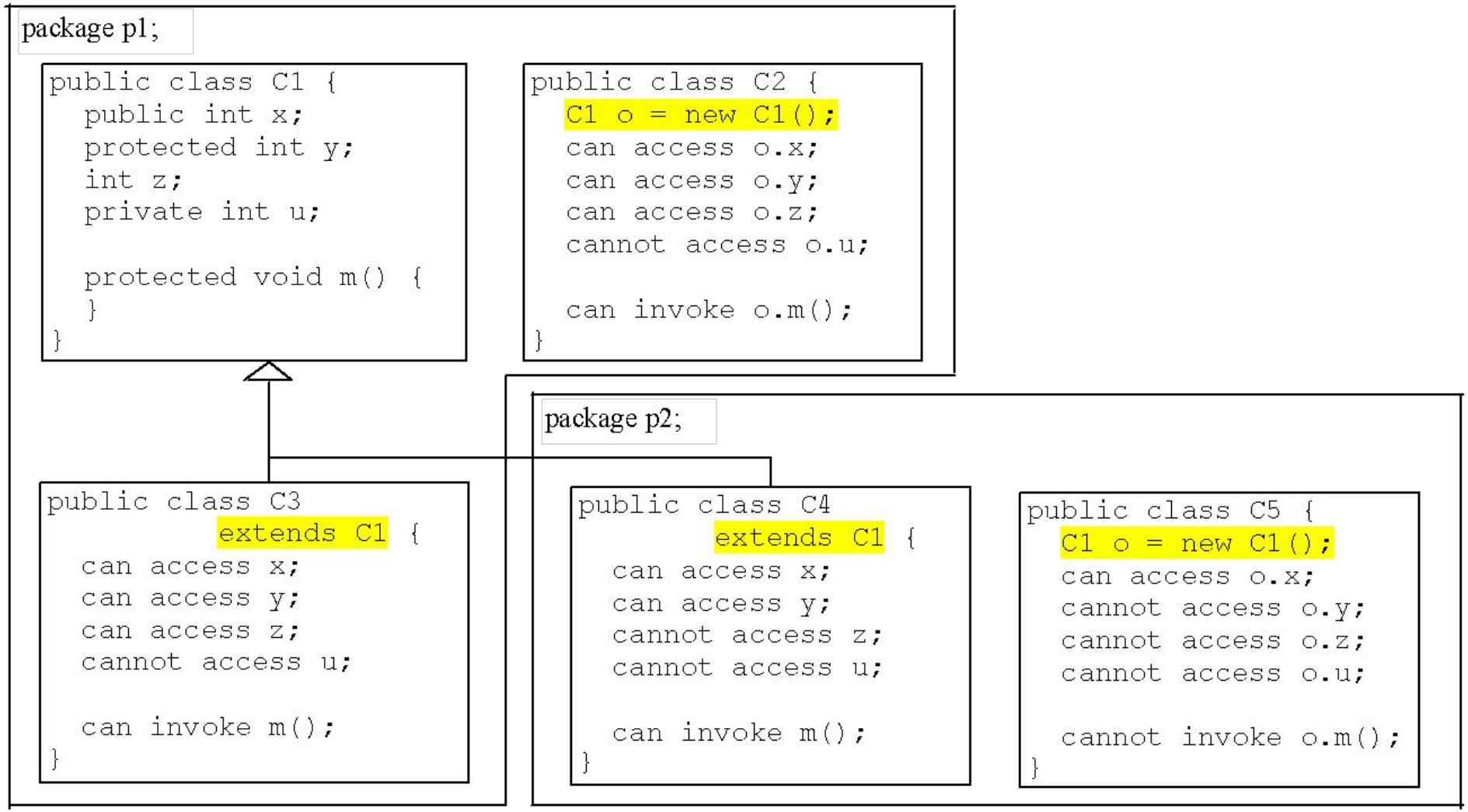
The protected Modifier

- 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, none (if no modifier is used), protected, public

Classes in the same package



Note

- 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.
  - **If a method is public in the superclass, it must be public in the subclass.**
- Modifiers are used on classes and class members (data and methods) except that the `final` modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

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.