# INHERITANCE & POLYMORPHISM

ITX 2001, CSX 3002, IT 2371

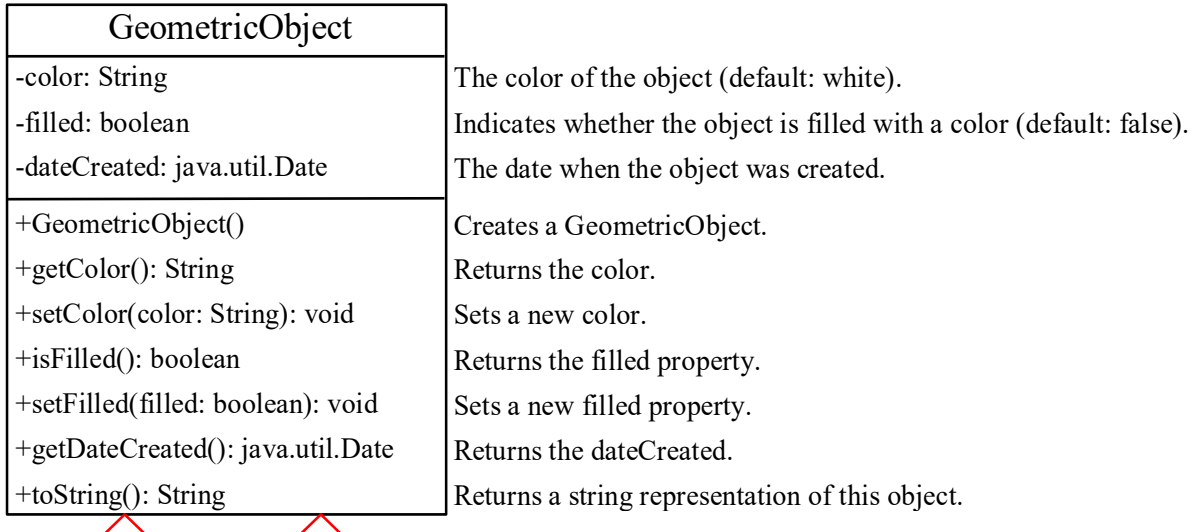
# OBJECTIVES

- To develop a subclass from a superclass through inheritance
- To invoke the superclass's constructors and methods using the super keyword
- To override methods in the subclass
- To distinguish differences between overriding and overloading
- To comprehend polymorphism, dynamic binding



[1]

# SUPERCLASSES AND SUBCLASSES

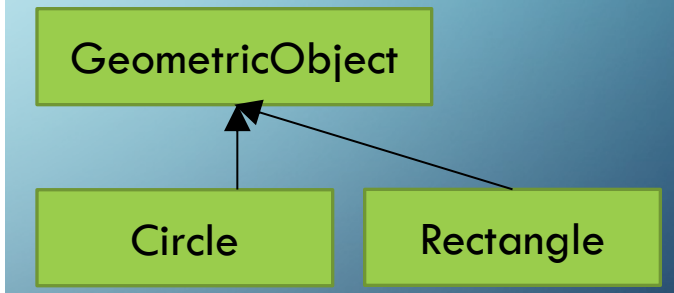



GeometricObject

Circle

Rectangle

TestCircleRectangle





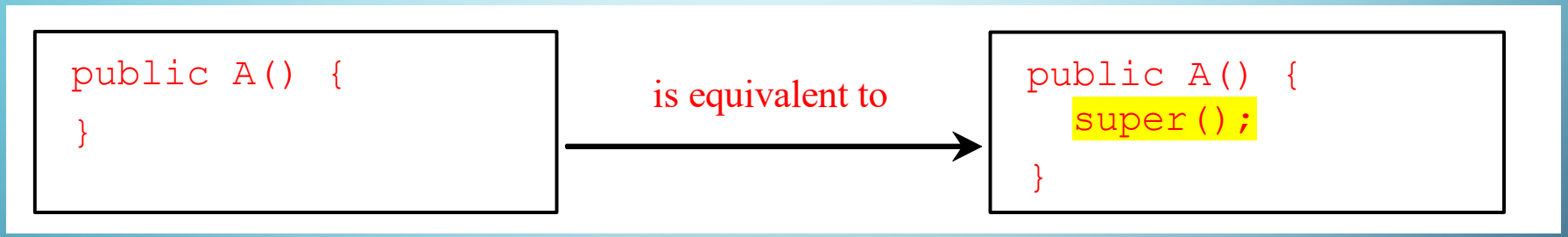

[1]

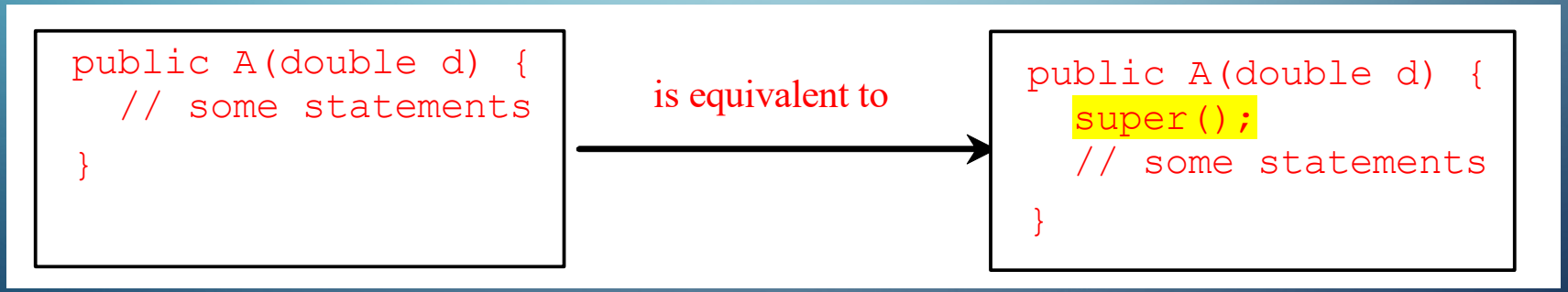
# WHICH PART OF SUPERCLASS ARE INHERITED?

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



# 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:

1. To call a superclass constructor
2. To call a superclass method

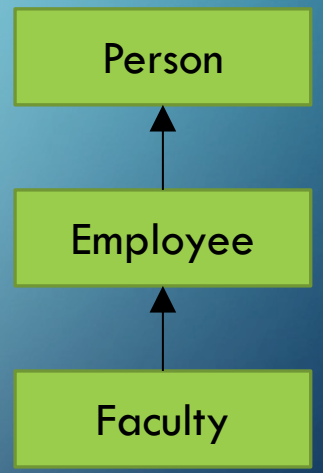
# CAUTION

- 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 super classes' constructors along the inheritance chain. This is called *constructor chaining*.

```
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");
  }
}
```

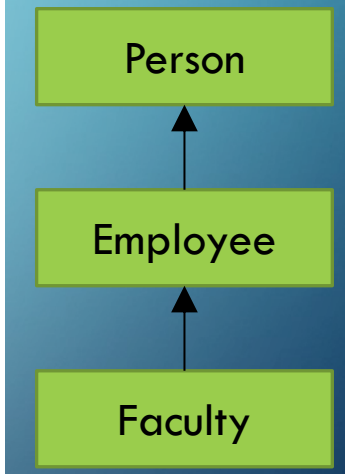


animation

# TRACE EXECUTION

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

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

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

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

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

1. Start from the main method

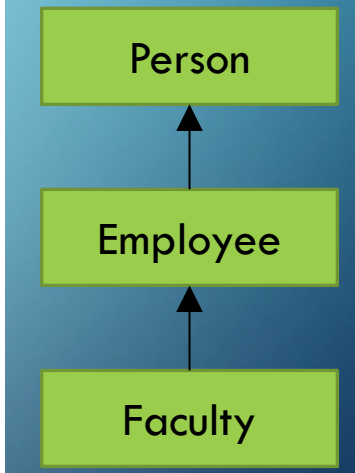


animation

# TRACE EXECUTION

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

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

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

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

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

2. Invoke Faculty constructor

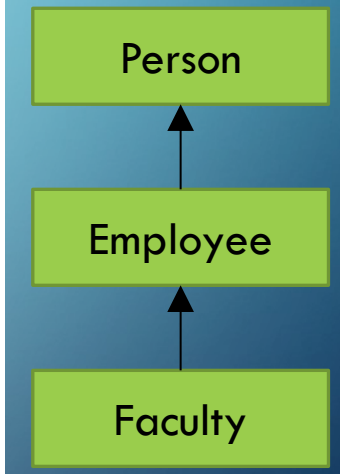


animation

# TRACE EXECUTION

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

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

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

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

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

3. Invoke Employee's no-arg constructor

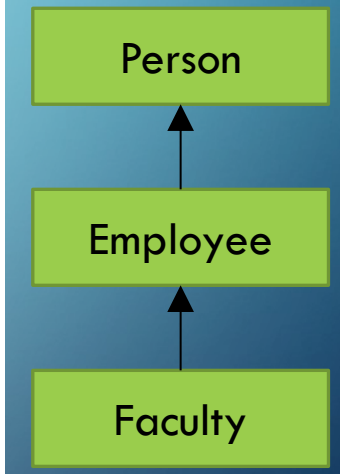


animation

# TRACE EXECUTION

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

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

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

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

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

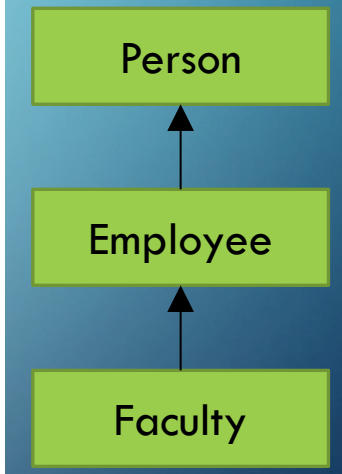
4. Invoke Employee(String) constructor



animation

# TRACE EXECUTION

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

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

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

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

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

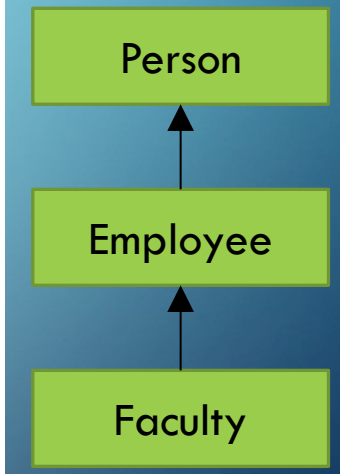



5. Invoke Person() constructor

animation

# TRACE EXECUTION

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

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

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

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

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

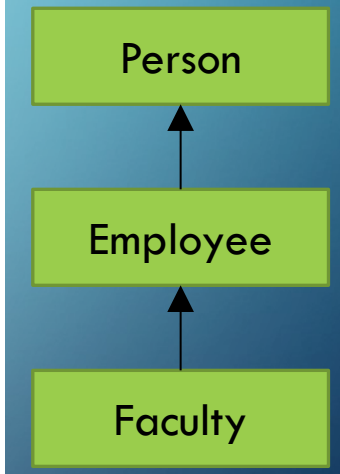



6. Execute println

animation

# TRACE EXECUTION

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

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

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

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

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



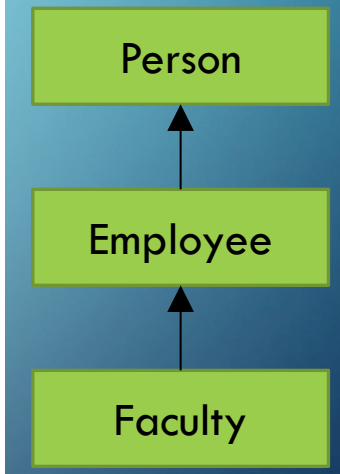

7. Execute println

animation

# TRACE EXECUTION

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

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

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

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

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

8. Execute println

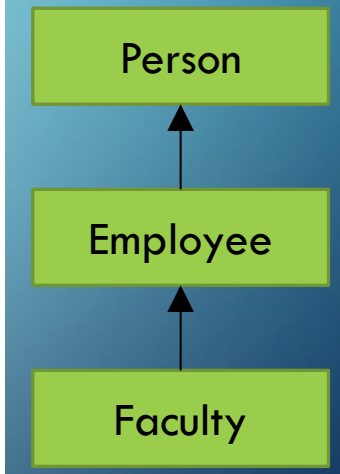


animation

# TRACE EXECUTION

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

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

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

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

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

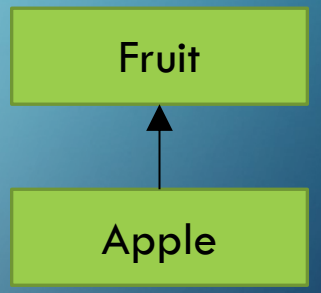
9. Execute println



# EXAMPLE ON THE IMPACT OF A SUPERCLASS WITHOUT NO-ARG CONSTRUCTOR

Find out the errors in the program:

```
public class Apple extends Fruit {
}

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






[1]

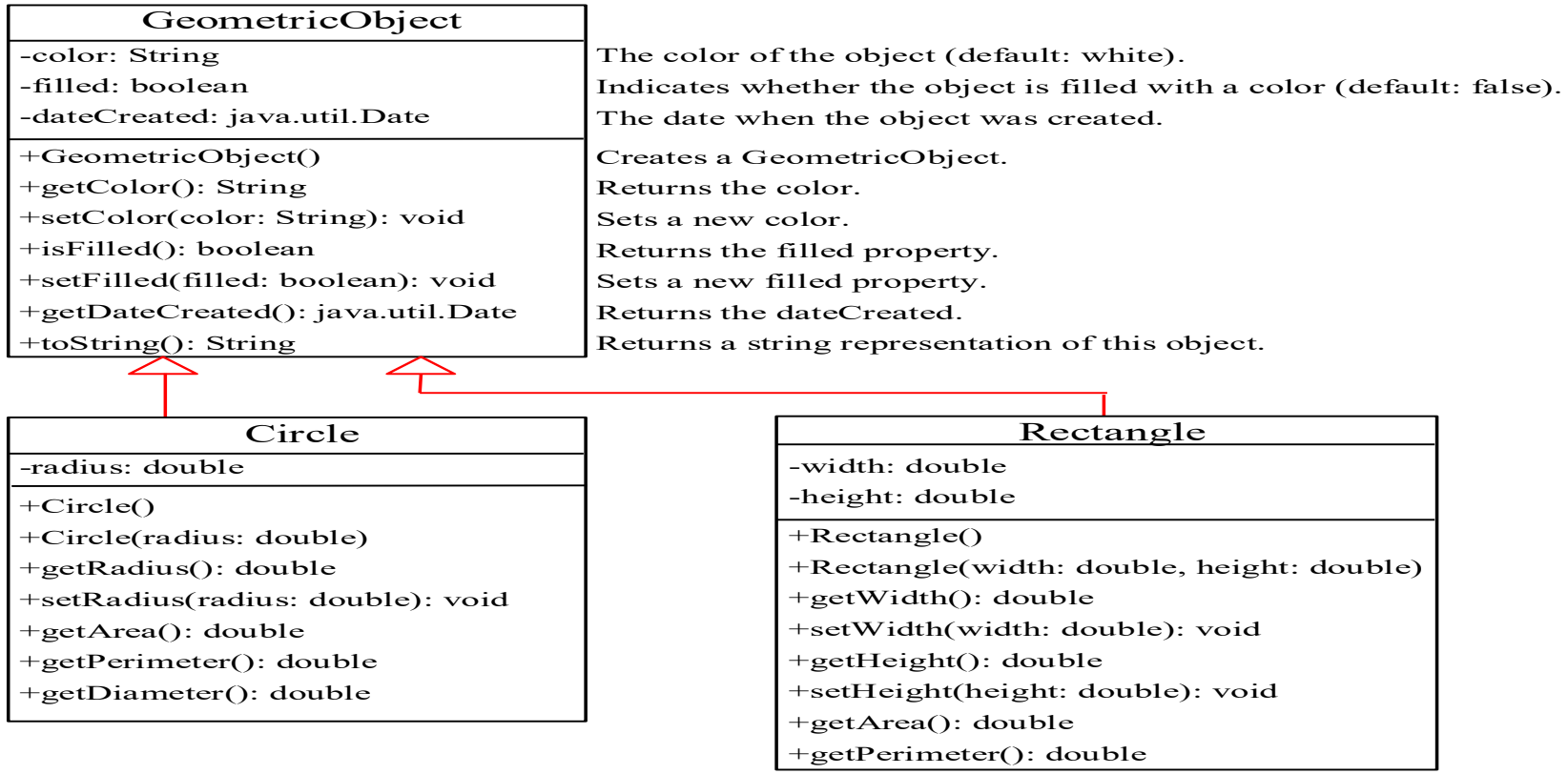
# DECLARING A SUBCLASS

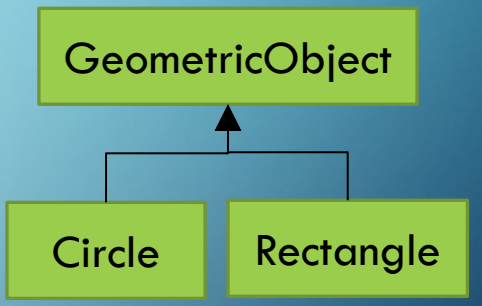
- A subclass extends properties and methods from the superclass.
- You can also:
  - Add new properties
  - Implement overloaded properties
  - Add new methods
  - Implement overloaded methods
  - Override the methods of the superclass

# CALLING SUPERCLASS METHODS

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

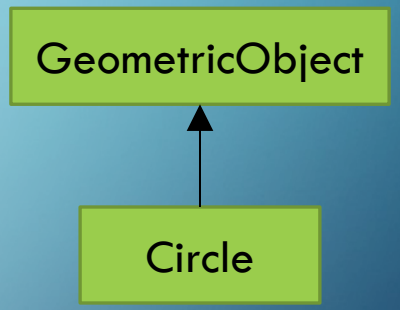
```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```



# OVERRIDING SUPERCLASS METHODS

- 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

- 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.

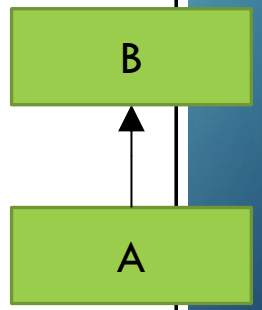
# NOTE

- 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 & OVERLOADING

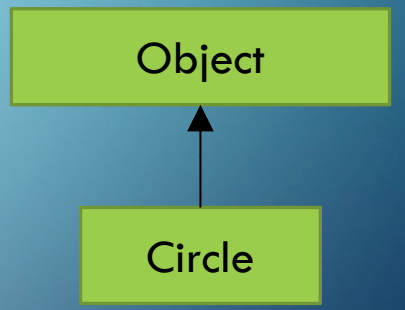
```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

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

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

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(double 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 {
   ...
}
```

# 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.

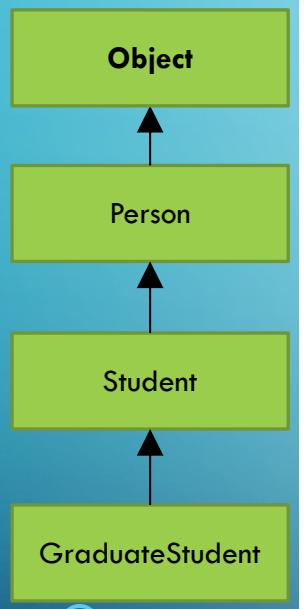
```
Loan loan = new Loan();
System.out.println(loan.toString());
```

# THE TOSTRING() METHOD IN 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




```
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";
  }
}
```

PolymorphismDemo

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

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

When the method m(Object x) is executed,

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

# DYNAMIC BINDING

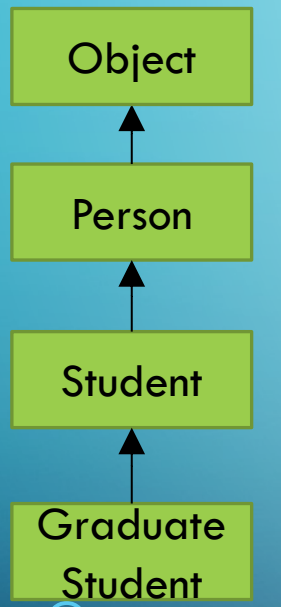
Dynamic binding works as follows: Suppose an object o is an instance of classes $C_1$, $C_2$, ..., $C_{n-1}$, and $C_n$, where $C_1$ is a subclass of $C_2$, $C_2$ is a subclass of $C_3$, ..., and $C_{n-1}$ is a subclass of $C_n$. That is, $C_n$ is the most general class, and $C_1$ is the most specific 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.

# METHOD MATCHING VS. BINDING

Matching a method signature and binding a method implementation are two issues.

1. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
2. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

# 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

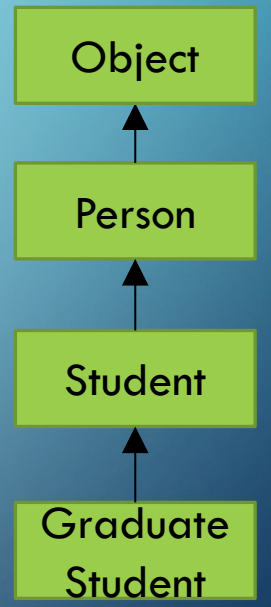
- You have already used the casting operator to convert variables of one primitive type to another.
- *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy.
  - In the preceding section, the statement

```
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.

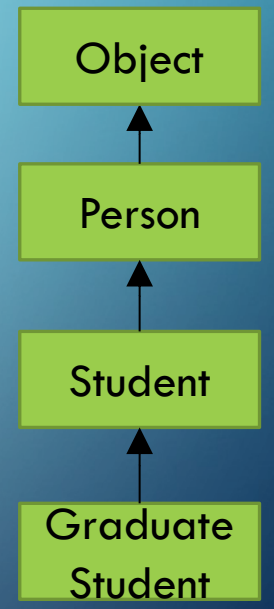


# WHY CASTING IS NECESSARY?

• 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 !!! but
**Student b = o** doesn't work?

- 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

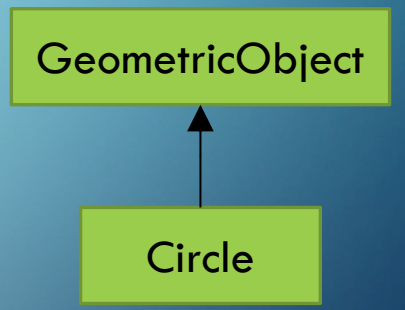
- Explicit casting must be used when casting an object from a superclass to a subclass.
- This type of casting may not always succeed.

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



# TIP

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: 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.

TestPolymorphismCasting

# THE ARRAYLIST AND VECTOR CLASSES

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.

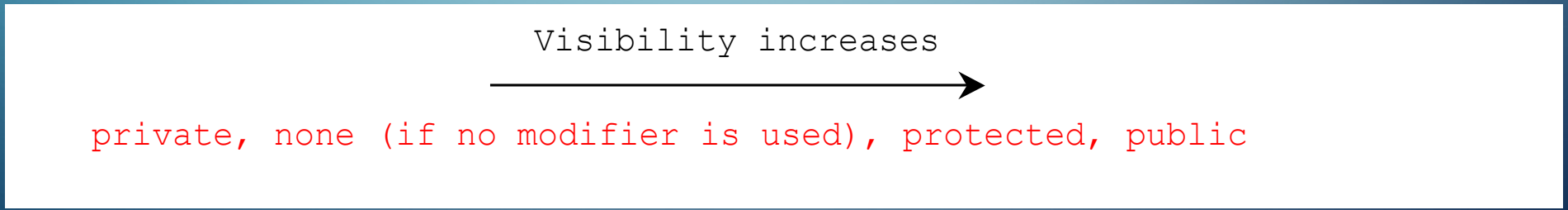
| java.util.ArrayList | |
|---|---|
| +ArrayList() | Creates an empty list. |
| +add(o: Object) : void | Appends a new element o at the end of this list. |
| +add(index: int, o: Object) : void | Adds a new element o at the specified index in this list. |
| +clear(): void | Removes all the elements from 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. |



[1]

# THE PROTECTED MODIFIER

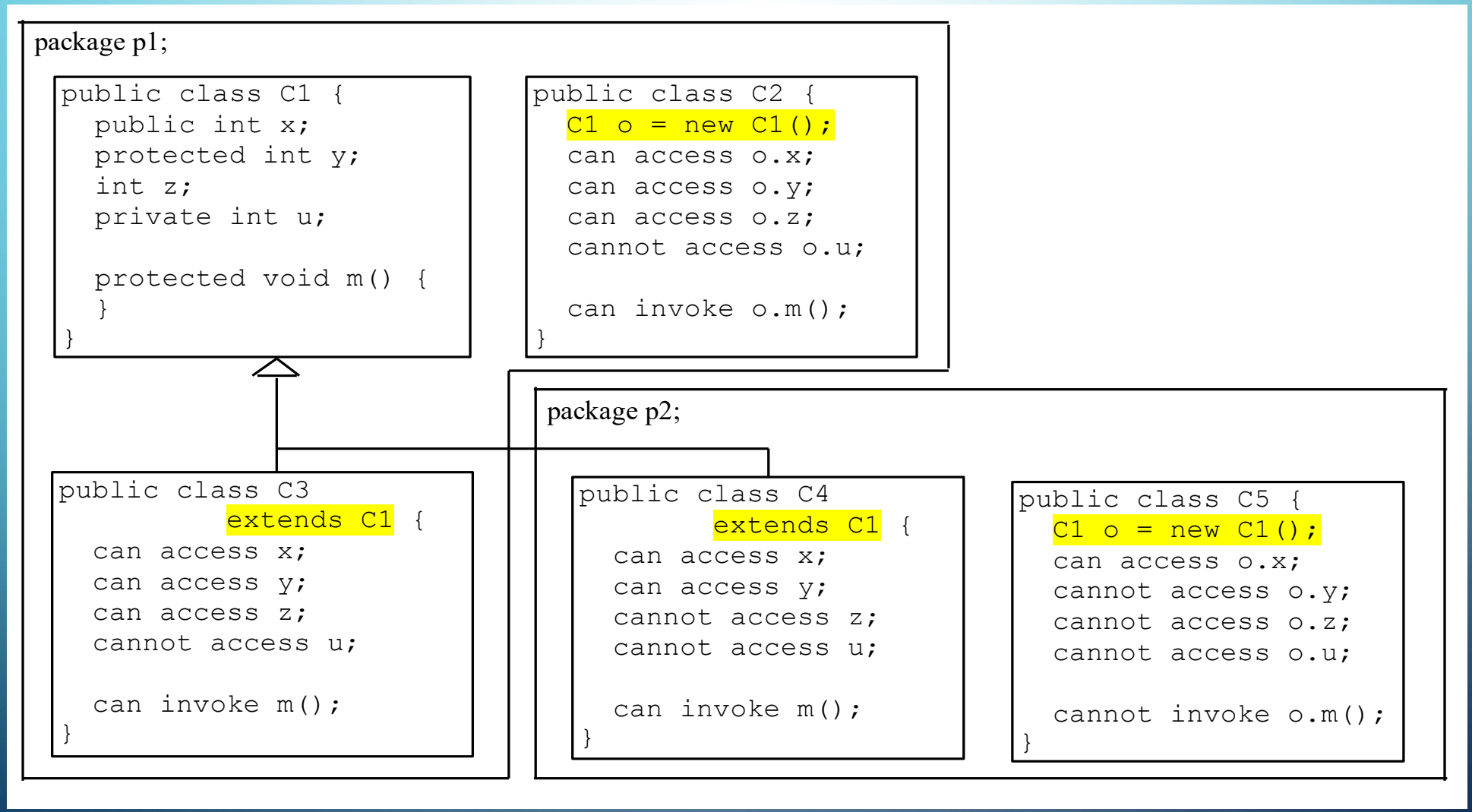
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by:
  - any class in the same package or
  - its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



# ACCESSIBILITY SUMMARY

| Modifier on members in a class | Accessed from the same class | Accessed from the same package | Accessed from a subclass | Accessed from a different package |
|---|---|---|---|---|
| public (+) | ✓ | ✓ | ✓ | ✓ |
| protected (#) | ✓ | ✓ | ✓ | – |
| default | ✓ | ✓ | – | – |
| private (–) | ✓ | – | – | – |

# VISIBILITY MODIFIERS



# A SUBCLASS CANNOT WEAKEN THE ACCESSIBILITY

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

# NOTE

- The 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 beoverridden by its subclasses.



[1]

Optional

# HIDING FIELDS AND STATIC METHODS

You can override an instance method, but you cannot override a field (instance or static) or a static method. If you declare a field or a static method in a subclass with the same name as one in the superclass, the one in the superclass is hidden, but it still exists. The two fields or static methods are independent. You can reference the hidden field or static method using the super keyword in the subclass. The hidden field or method can also be accessed via a reference variable of the superclass's type.



[1]

Optional

# HIDING FIELDS AND STATIC METHODS, CONT.

When invoking an instance method from a reference variable, the actual class of the object referenced by the variable decides which implementation of the method is used at runtime. When accessing a field or a static method, the declared type of the reference variable decides which method is used at compilation time.

HidingDemo



[1]

Optional

# INITIALIZATION BLOCK

Initialization blocks can be used to initialize objects along with the constructors. An initialization block is a block of statements enclosed inside a pair of braces. An initialization block appears within the class declaration, but not inside methods or constructors. It is executed as if it were placed at the beginning of every constructor

```
public class Book {
  private static int numOfObjects;
  private String title
  private int id;

  public Book(String title) {
    this.title = title;
  }

  public Book(int id) {
    this.id = id;
  }

  {
    numOfObjects++;
  }
}
```

Equivalent

```
public class Book {
  private static int numOfObjects;
  private String title;
  private int id;

  public Book(String title) {
    numOfObjects++;
    this.title = title;
  }

  public Book(int id) {
    numOfObjects++;
    this.id = id;
  }
}
```



[1]

# REFERENCES

[1] Liang, “Introduction to Java Programming”, 6th Edition, Pearson Education, Inc.