NYU

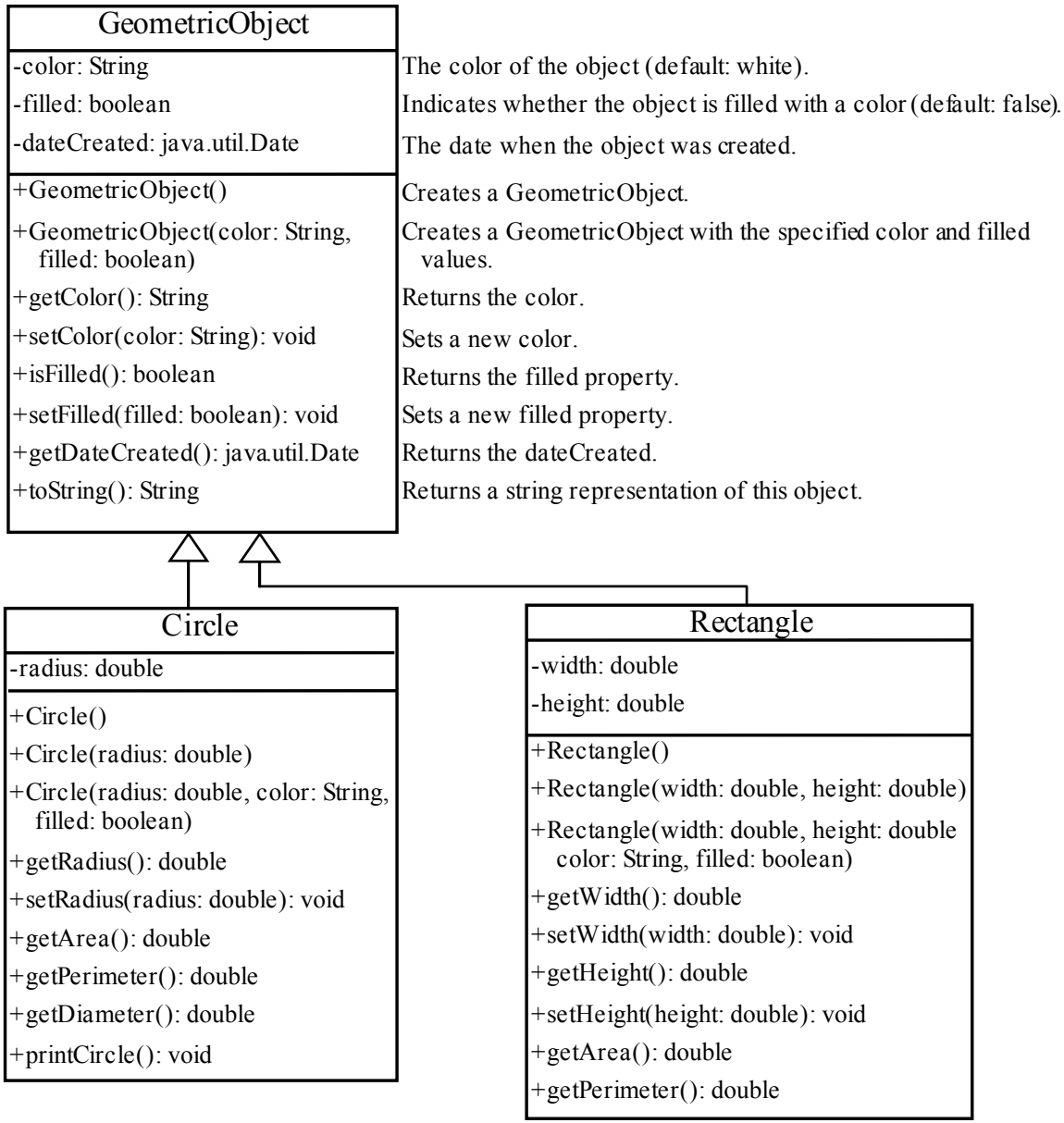
Lecture 9

Inheritance and Polymorphism

Motivations

- Suppose you will define classes to model circles, rectangles, and triangles.
- These classes have many common features.
- What is the best way to design these classes so to avoid redundancy?
  - The answer is to use inheritance.

Superclasses and Subclasses



Defining a Subclass

- A subclass inherits from a superclass.

- You can also:
  - Add new properties
  - Add new methods
  - **Override** the methods of the superclass

Example

```
//Superclass
public class GeometricObject
{
   public int x, y;
   public String getDateCreated() { ... }
   public double area() { ... }
}
//Subclass
public class Circle extends GeometricObject {
   public double radius;

   public double getDiameter() { ... }
   public void printCircle() { ... }
}
```

Using the Keyword super

- The keyword `super` refers to the superclass of the class in which `super` appears.

- This keyword can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method

Calling Superclass Methods

- You could implement the `printCircle()` method in the `Circle` class as follows:

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

Overriding Methods in the 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***.

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the area method defined in GeometricObject */
  public double area() {
    return radius*radius*Math.Pi();
  }
}
```

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

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 {
  // 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 {
  // overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Note

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

Note

- 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.

The protected Modifier

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

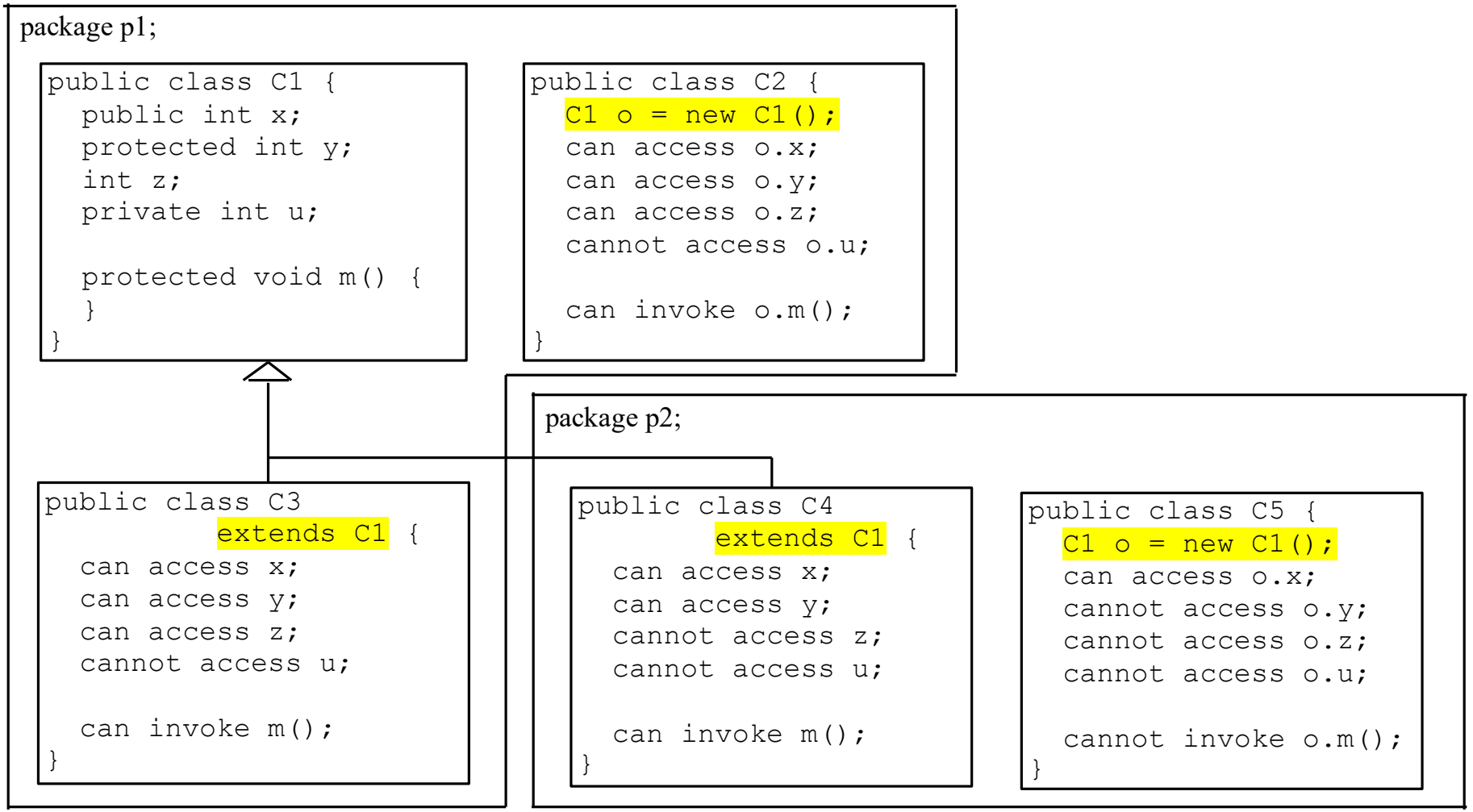
- private, default, protected, public

Visibility increases

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



Accessibility Weakening

- 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 be overridden by its subclasses.

The Object Class and Its Methods

- Every class in Java is descended from the `java.lang.Object` class.

- If no inheritance is specified when a class is defined, the superclass of the class is Object.

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

equivalent

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

The toString() method in Object

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

- Example

```
Circle circle = new Circle();
System.out.println(circle.toString());
```

The code displays something like Circle@15037e5.

The toString() method in Object

- 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.

```
public String toString() {
  return
     "The circle is created " +
     super.getDateCreated() +
     " and the radius is " + radius);
}
```

The equals Method

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

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

- Better

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

Note

- The == comparison operator is used for comparing
  - two primitive data type values or
  - for determining whether two objects have the same references.

- The **equals method** is intended to test whether two objects have the **same contents**, provided that the method is modified in the defining class of the objects.
- The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

Polymorphism

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

- 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**.

Example

```
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";
  }
}
```

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***.

Example

- 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

- Suppose an object `o` is an instance of classes `C1`, `C2`, ..., and `Cn`
- `C1` is a subclass of `C2`, `C2` is a subclass of `C3`, ..., and `Cn-1` is a subclass of `Cn` (`Cn` is the most general class, and `C1` is the most specific class).
- In Java, `Cn` is the Object class.
- If `o` invokes a method `p`, the JVM searches the implementation for the method `p` in `C1`, `C2`, ..., `Cn`, 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.
- 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

```
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";
  }
}
```

Generic Programming

- 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

- We 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.
- Example:

```
Object o = new Student(); // Implicit casting
m(o);
```

- This statement, known as implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

Why Casting Is Necessary?

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

- Suppose you want to assign the object reference `o` to a variable of the Student type using the following statement:

```
Student b = o;
```

- A compile error would occur.

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

Why Casting Is Necessary?

- 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 like the one used for casting among primitive data types.

```
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


// Cast 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.

Superclass Constructor Inherited

- No. They are **not** inherited.
- They are invoked explicitly or implicitly.
- Explicitly using the `super` keyword.

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

Superclass Constructor Invoked

- A constructor may invoke an overloaded constructor or its superclass's constructor.
- If none of them is invoked explicitly, the compiler puts `super()` as the first statement in the constructor.

```
public A() {
}
```

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

```
public A() {
   super();
}
```

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

Caution

- You must use the keyword `super` to call the superclass constructor.

- Invoking a superclass constructor's name in a subclass causes a syntax error.
- Java requires that the statement that uses the keyword `super` appear first in the constructor.

Constructor Chaining

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

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");
  }
}
```

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");
  }
}
```

Start from the main method

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");
  }
}
```

Invoke Faculty constructor

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");
  }
}
```

Invoke Employee's no-arg constructor

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");
  }
}
```

Invoke Employee(String) constructor

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 construct        invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overload    constructor");
    System.out.println("(3) Employee'    o-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");
  }
}
```

Invoke Person() constructor

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 i      d");
  }
}

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

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

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

Execute println

Constructor Chaining

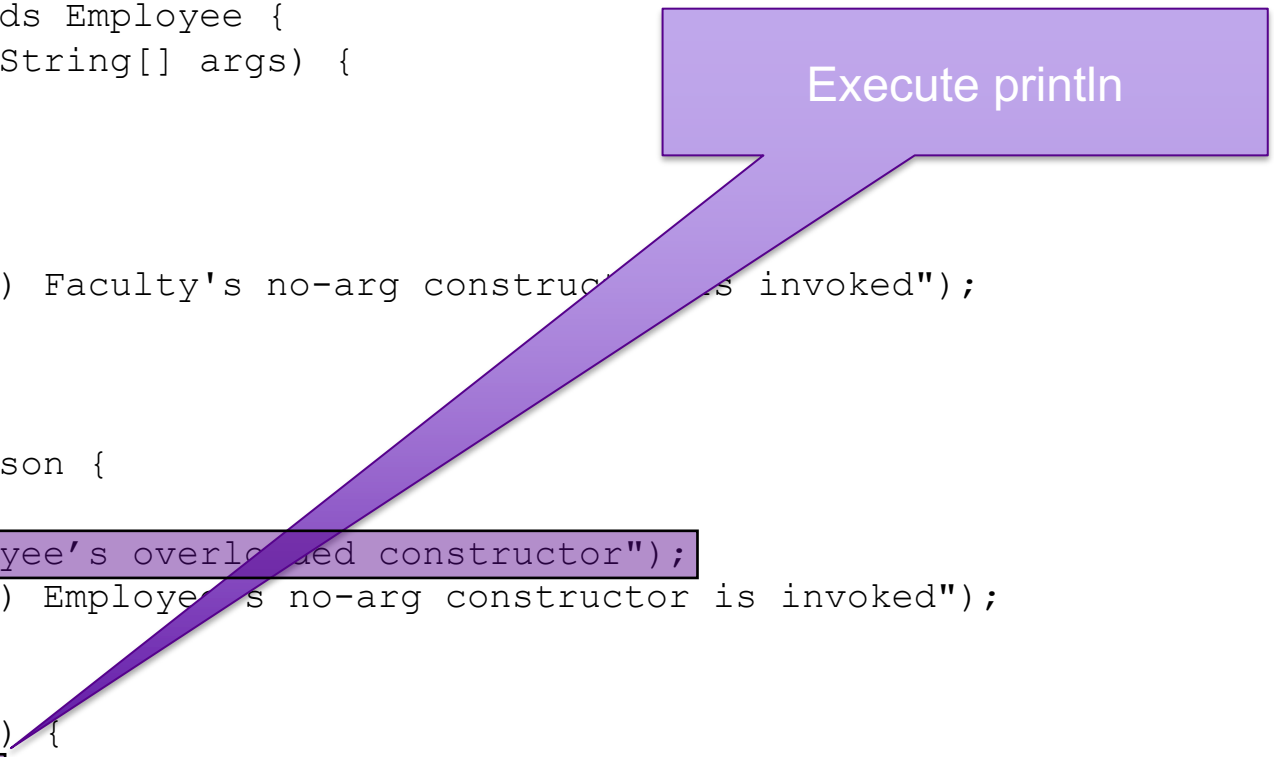
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  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");
  }
}
```



Constructor Chaining

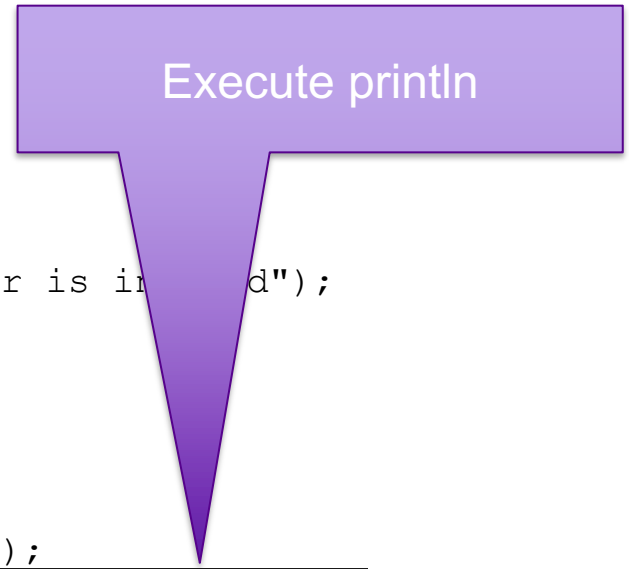
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  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");
  }
}
```



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");
  }
}
```

Execute println

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");
  }
}
```

Done

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");
  }
}
```

The ArrayList Class

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

The ArrayList Class

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 element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+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.

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

Arrays vs ArrayList

Operation	Array	ArrayList
Creation	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "String";`	`list.set(index, "String");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("String");`
Inserting a new element		`list.add(index, "String");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove("String");`
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<>(Arrays.asList(array));
```

- Creating an array of objects from an `ArrayList`:

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

max and min in an Array List

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(
  java.util.Collections.max(
     new ArrayList<String>(
          Arrays.asList(array))));
```

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

Shuffling an Array List

```
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);
```