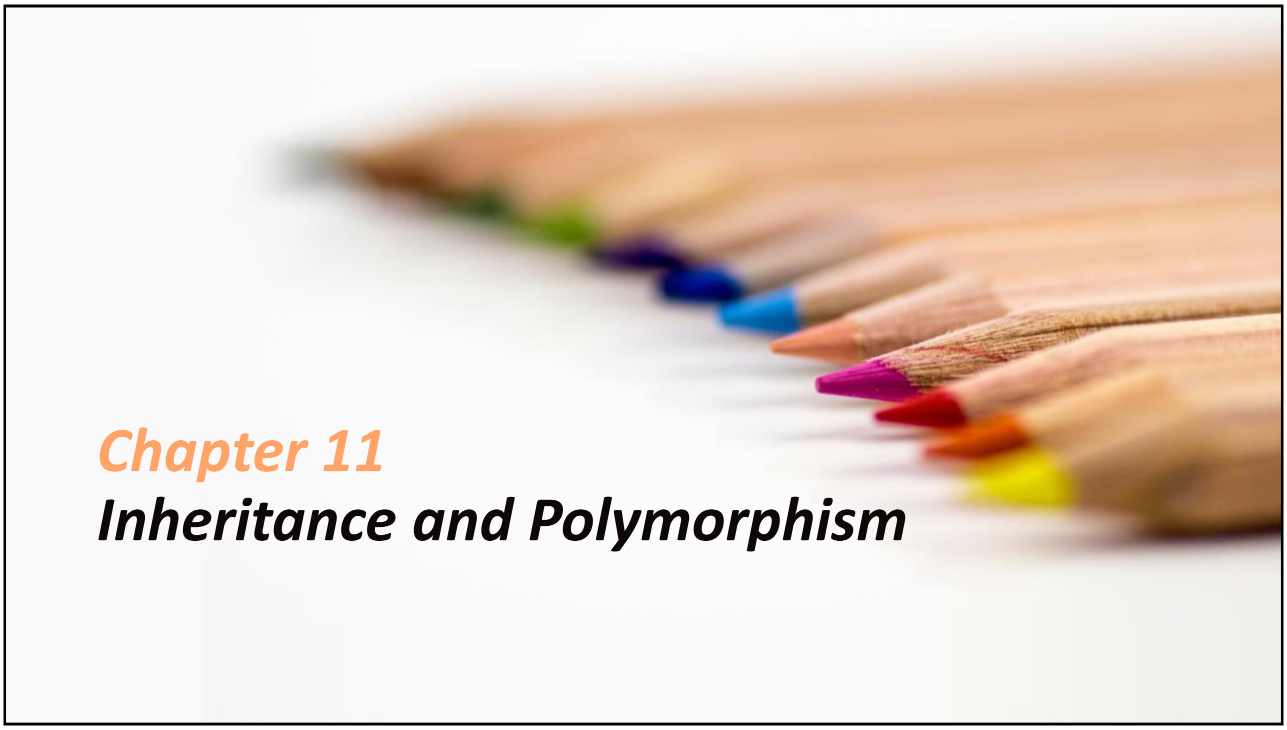
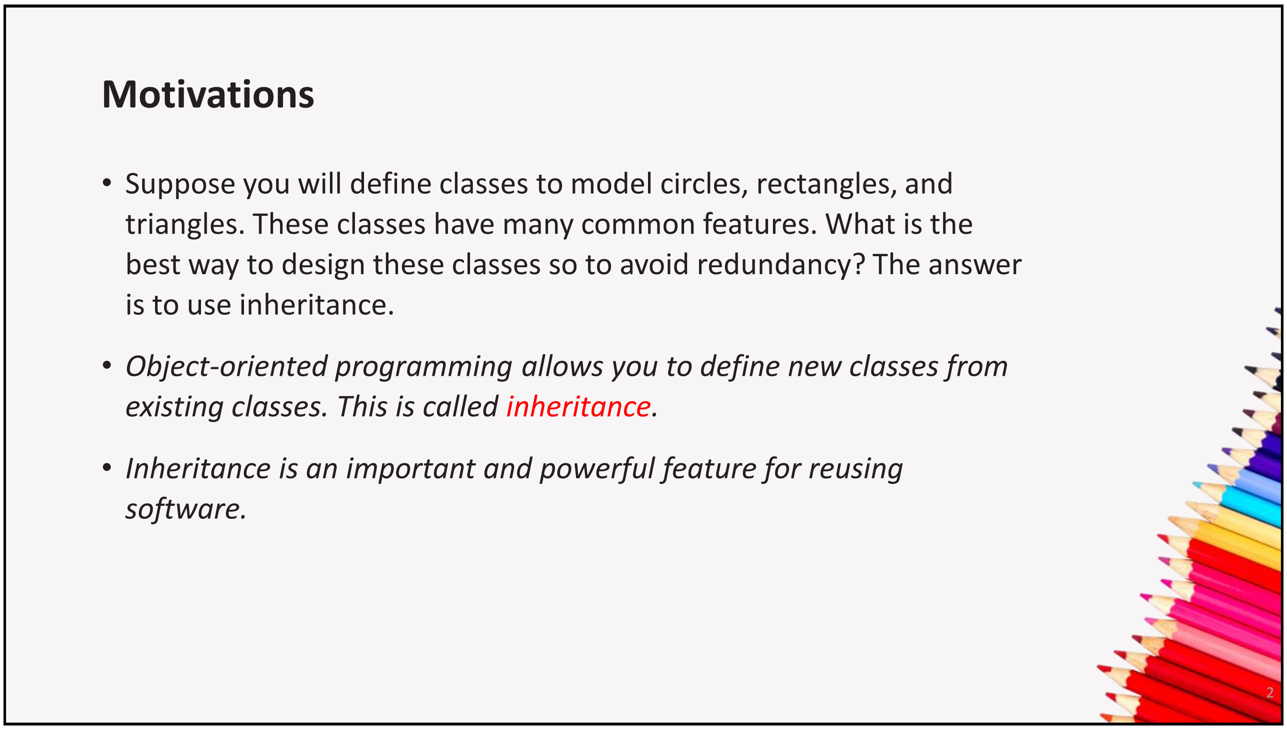


Chapter 11
Inheritance and Polymorphism



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

2

Objectives(1)

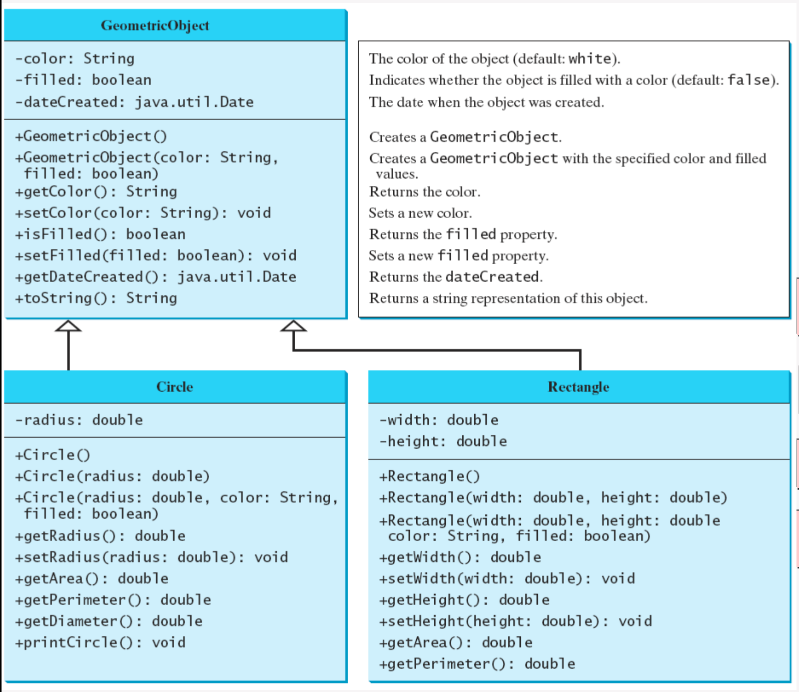
- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the *super* keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the **toString() method** in the Object class (§11.6).
- To discover *polymorphism* and *dynamic binding* (§§11.7–11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).

Objectives(2)

- To explore the equals method in the **Object** class (§11.10).
- To store, retrieve, and manipulate objects in an **ArrayList** (§11.11).
- To construct an array list from an array, to sort and shuffle a list, and
- to obtain max and min element from a list (§11.12).
- To implement a Stack class using ArrayList (§11.13).
- To enable data and methods in a superclass accessible from subclasses using the protected visibility modifier (§11.14).
- To prevent class extending and method overriding using the *final* modifier (§11.15).

11.2 Superclasses and Subclasses

- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes.
- You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.
- *Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).*



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

GeometricObject

CircleFromSimpleGeometricObject

RectangleFromSimpleGeometricObject

TestCircleRectangle

Run

6

Superclasses and Subclasses

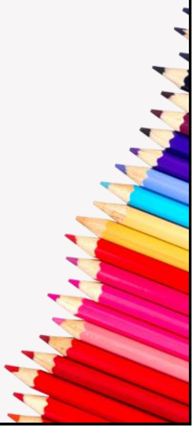
- In Java terminology, a class C1 extended from another class C2 is called a *subclass, and C2* is called a *superclass.*
- *A superclass is also referred to as a parent class or a base class, and a* subclass as a *child class, an extended class, or a derived class.*
- *A subclass inherits accessible*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

NOTE

- More or less in subclass?
- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass.
- Not all is-a relationships should be modeled using inheritance.
- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods.
- Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as *single inheritance.*

11.3 Using the super Keyword

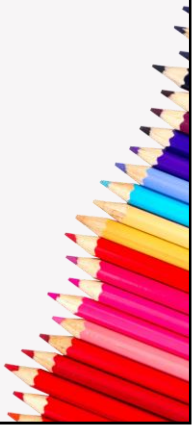
- Are Superclass's Constructor Inherited? Can the superclass's constructors be invoked from a subclass?
- No. They are not inherited. They are invoked explicitly or implicitly.
- Explicitly using the **`super`** keyword.
  - A constructor is used to construct an instance of a class. Unlike properties 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.*

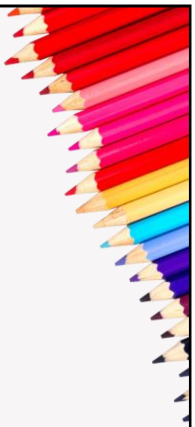


Calling Superclass Constructors

- *The keyword* **super** *refers to the superclass and can be used to invoke the superclass's methods and constructors.*
  1. To call a superclass constructor.
  2. To call a superclass method.
- The syntax to call a superclass's constructor is:
  - **super(),** or **super(parameters);**

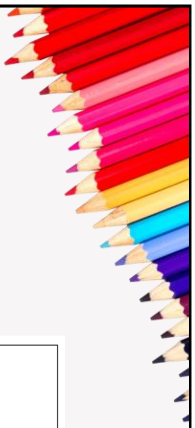
```
public CircleFromSimpleGeometricObject(
    double radius, String color, boolean filled) {
  super(color, filled);
  this.radius = radius;
}
```





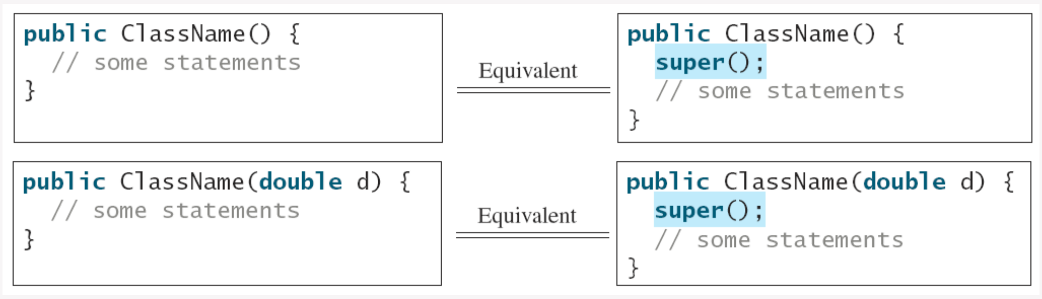
CAUTION

- 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.



Superclass's Constructor Is Always 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. 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
}
```

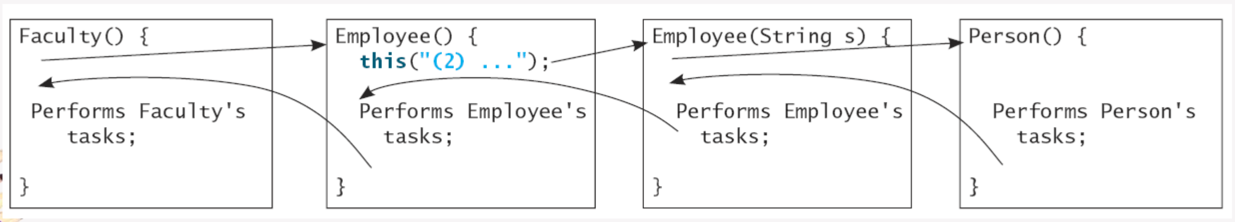
Constructor Chaining

- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.
- When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks.
- If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.
- This process continues until the last constructor along the inheritance hierarchy is called.
- This is called *constructor chaining.*

Example

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

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invokes Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Performs Person's tasks");
  }
}
```

Example



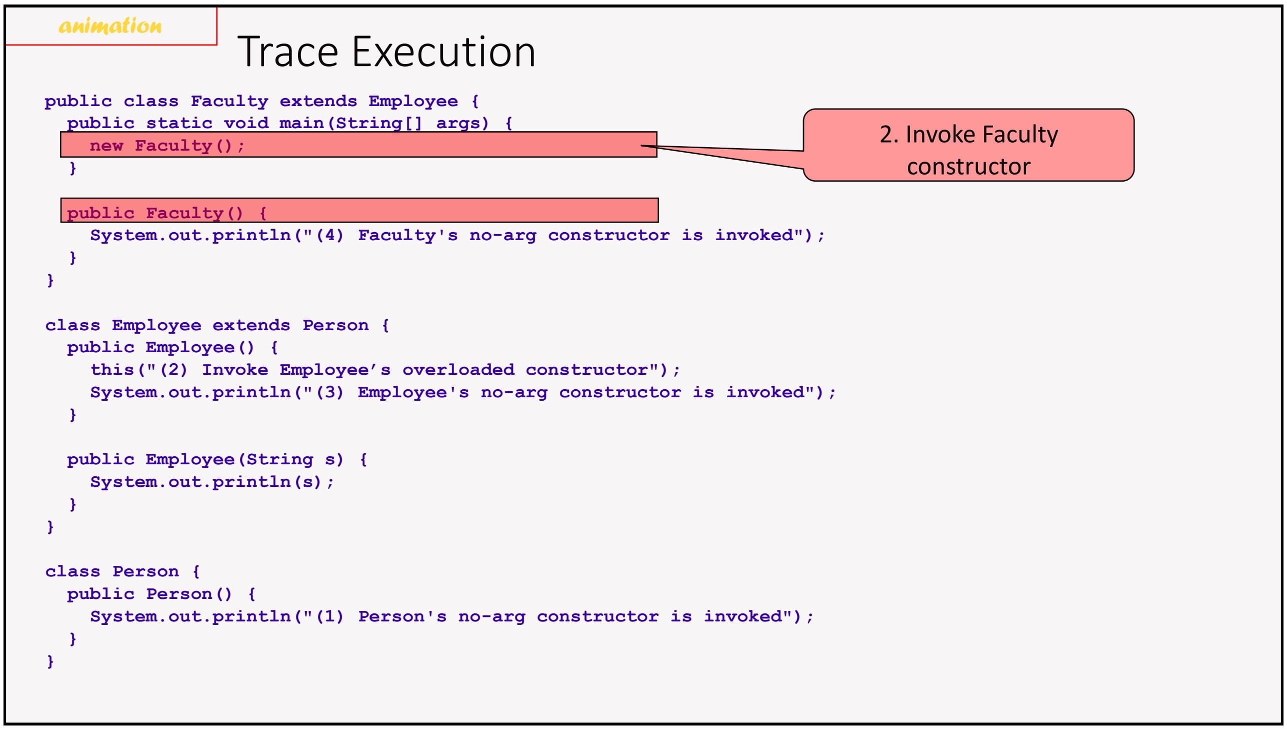
animation

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

16

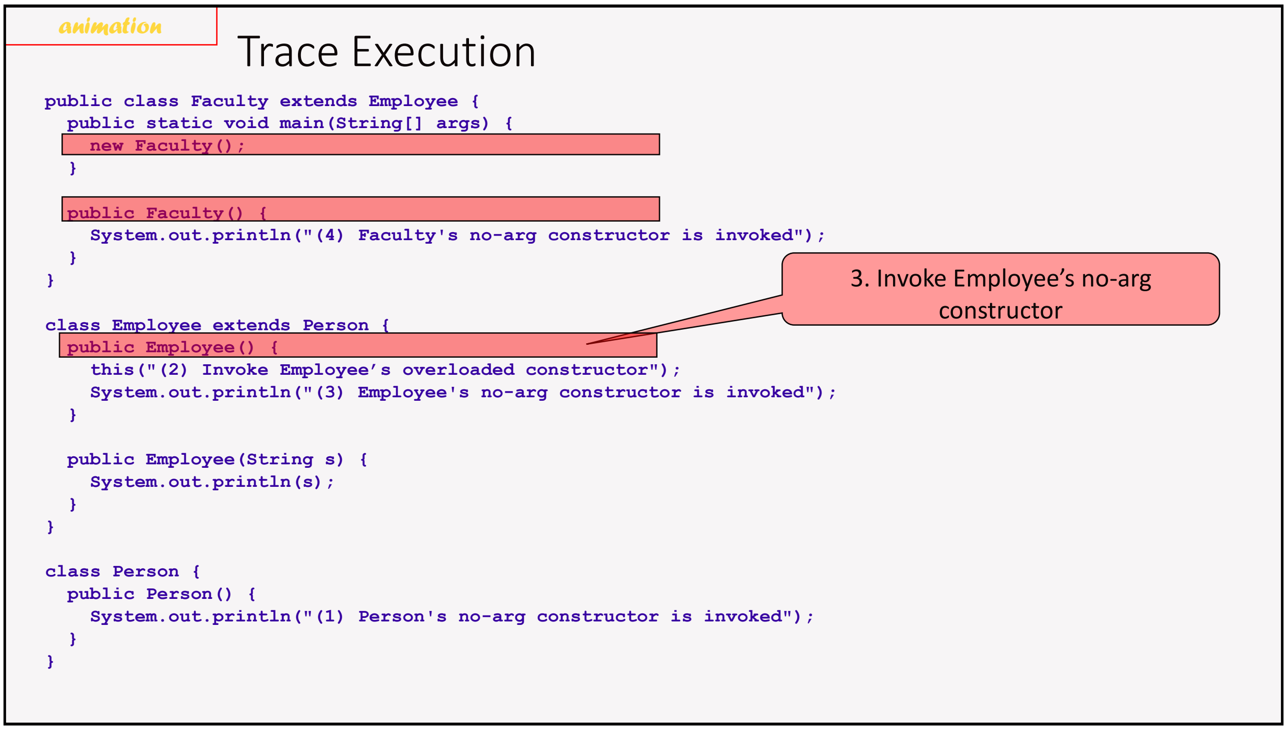


animation

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");
  }
}
```

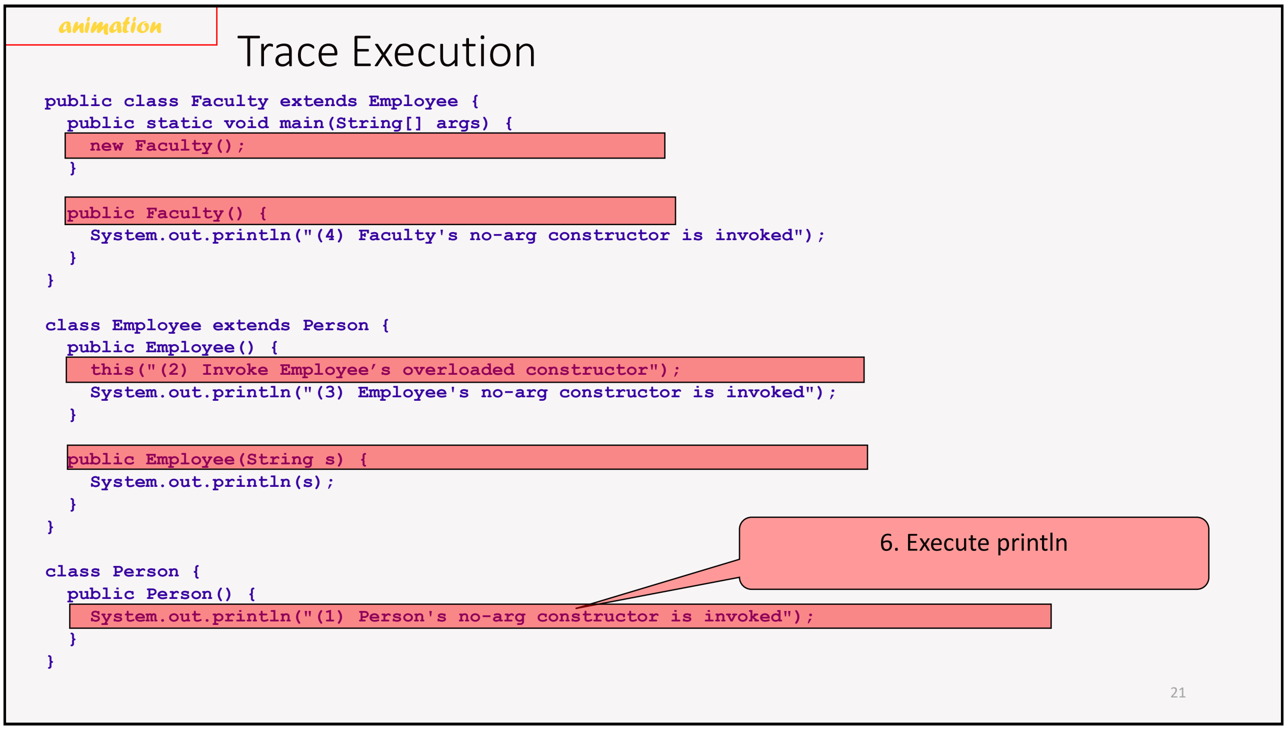
4. Invoke Employee(String) constructor

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



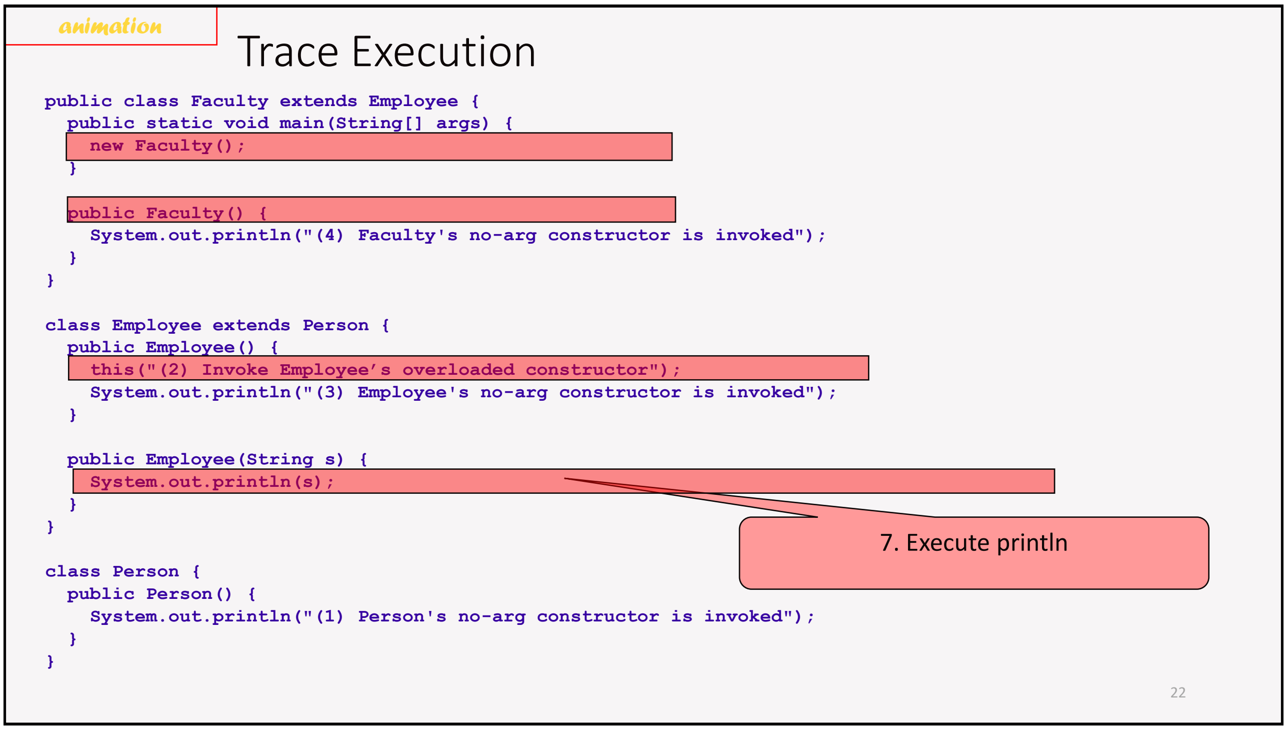
animation

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

21



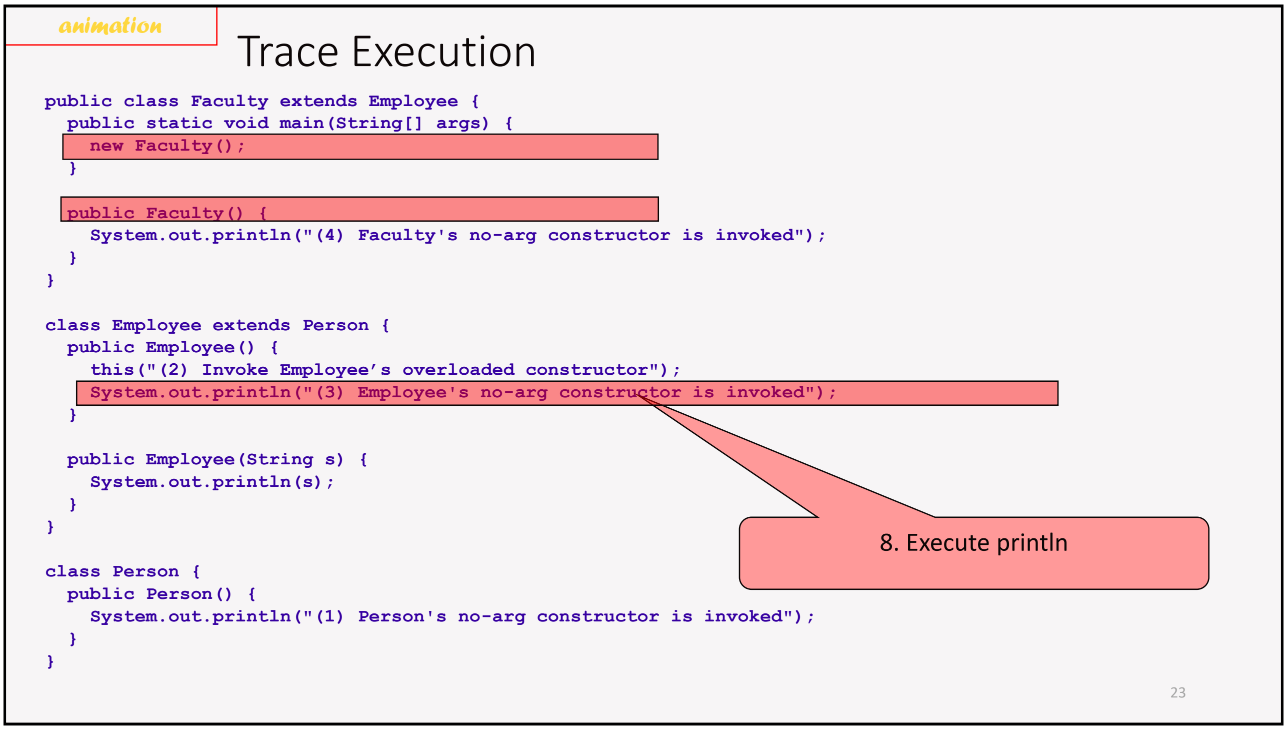
animation

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

22



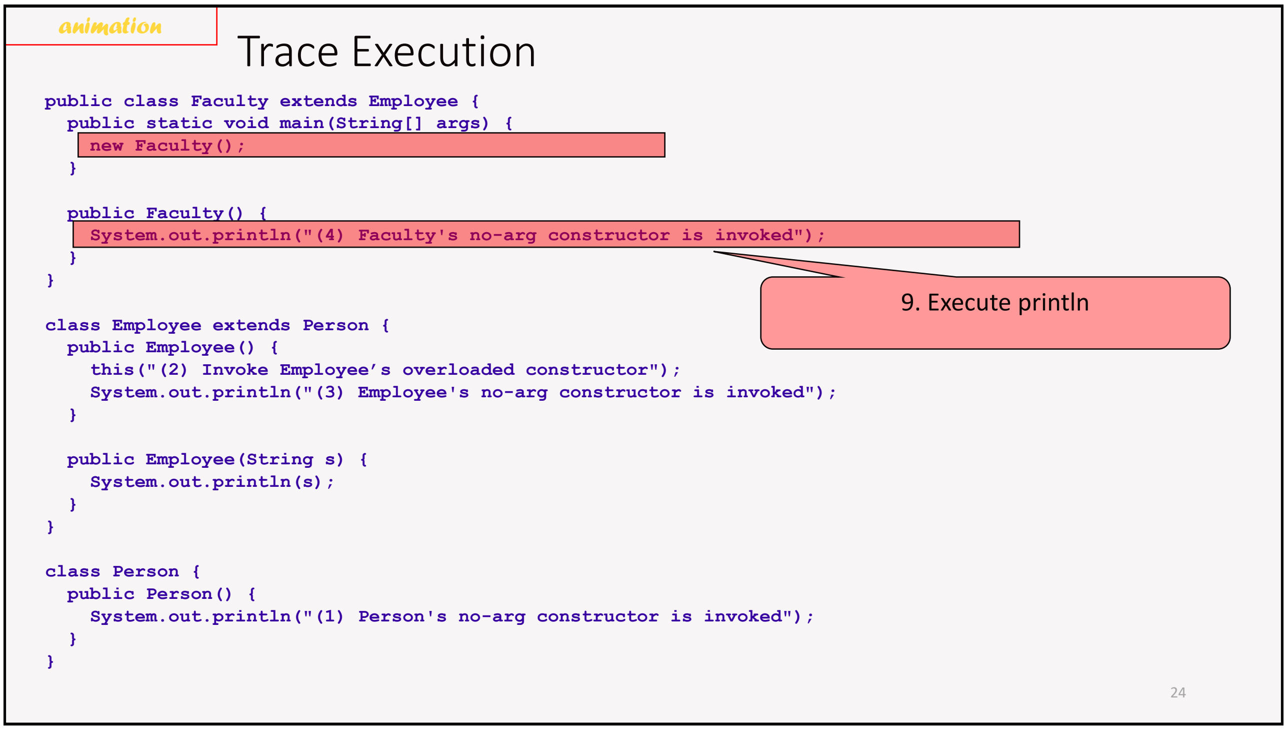
animation

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

23



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

24

Example on the Impact of a Superclass without no-arg Constructor

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.

```
public class Apple extends Fruit {
}

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

Calling Superclass Methods

- The keyword **super** can also be used to reference a method other than the constructor in the superclass.
- The syntax is:

```
super.method(parameters);
```

- For example, 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);
}
```

11.4 Overriding Methods

- A subclass inherits from a superclass. You can also:
  - Add new properties
  - Add new methods
  - Override the methods of the superclass
- Method overriding
  - 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*.

Overriding Methods in the Superclass

- Overriding the **toString** method in the GeometricObject class in the Circle class.

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

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

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.
- 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 hidden static methods can be invoked using the syntax SuperClassName.staticMethodName.

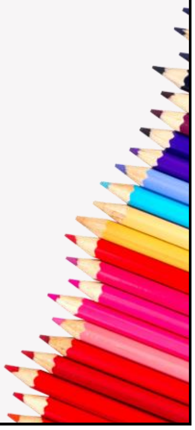
11.5 Overriding vs. Overloading

```
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);
  }
}
```

Overriding vs. Overloading

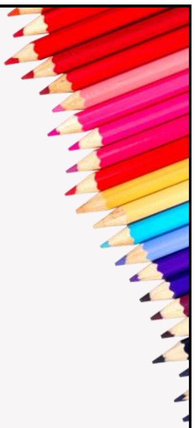
- *Overloading means to define multiple methods with the same name but different signatures.*
- *Overriding means to provide a new implementation for a method in the subclass.*
- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.



@Override

- To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the method in the subclass. For example:

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

  @Override
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```



11.6 The Object Class and Its toString() Method

- *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** by default. For example, the following two class definitions are the same:

```
public class  ClassName {
  ...
}
```

Equivalent

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

The `toString()` method in Object

- The signature of the toString() method is:

  `public String toString()`

- Invoking toString() on an object returns a string that describes the object.
  - By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@),and the object's memory address in hexadecimal.
- Usually you should override the **toString** method so that it returns a descriptive string representation of the object.

```
public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
    " and filled: " + filled;
}
```

11.7 Polymorphism

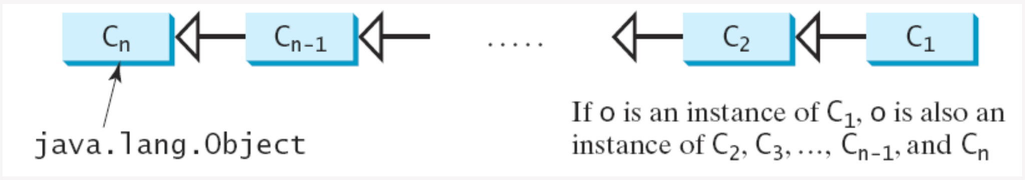
- The three pillars of object-oriented programming are *encapsulation*, *inheritance*, and *polymorphism*.
- A class defines a type. A type defined by a subclass is called a *subtype, and a type defined by its superclass is called* a *supertype.*
- *Polymorphism means that a variable of a supertype can refer to a subtype object.*
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

Polymorphic Call Demo

```java
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new Circle(1, "red", false));
    displayObject(new Rectangle(1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(GeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

11.8 Dynamic Binding

- Dynamic binding works as follows: Suppose an object o is an instance of classes C1, C2, ..., Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, ..., and Cn-1 is a subclass of Cn. That is, 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-1 and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.


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

Dynamic Binding Demo

```
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" ;
  }
}
```

DynamicBindingDemo

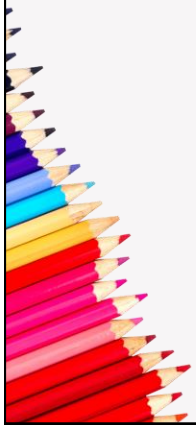
Run

Dynamic Binding

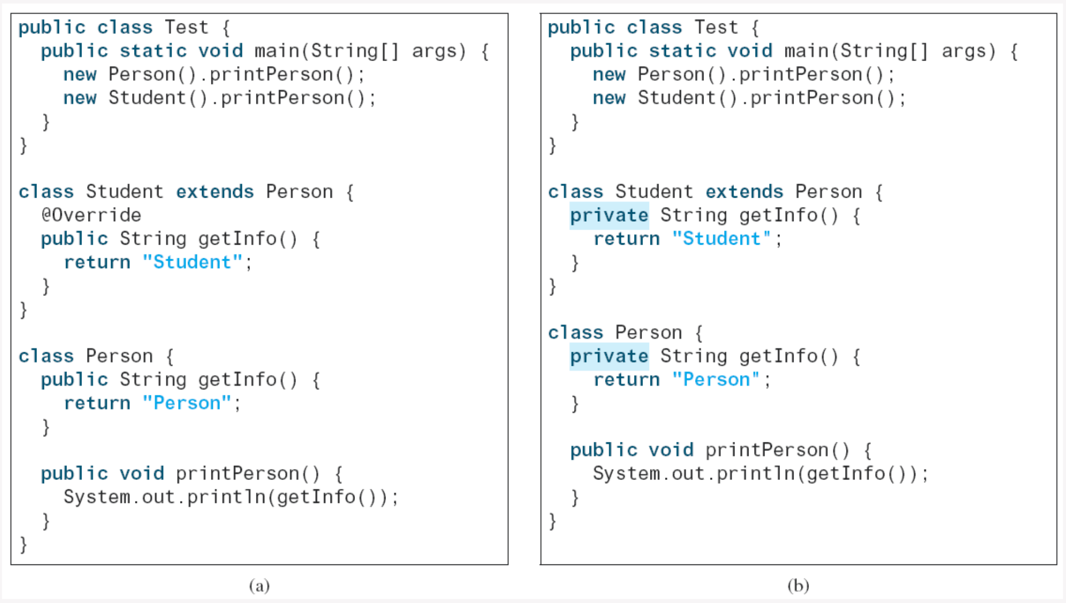
- *A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.*
- A variable must be declared a type. The type that declares a variable is called the variable's *declared type.*
- A variable of a reference type can hold a null value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype. The *actual type* of the variable is the actual class for the object referenced by the variable.
- The declared type of the reference variable decides which method to match at *compile time*. The JVM dynamically binds the implementation of the method at *runtime*, decided by the actual type of the variable.

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.



Check Point

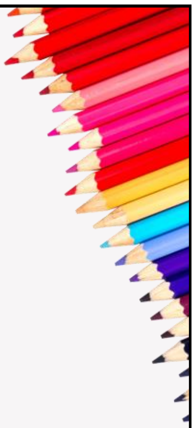


```java
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  private String getInfo() {
    return "Student";
  }
}

class Person {
  private String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(b)



11.9 Casting Objects and the instanceof Operator

- One object reference can be typecast into another object reference. This is called casting object.
- Implicit casting / Upcasting

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

  - It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting), because an instance of a subclass is always an instance of its superclass.
- Explicit casting / Downcasting

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

  - When casting an instance of a superclass to a variable of its subclass (known as downcasting), explicit casting must be used to confirm your intention to the compiler with the (SubclassName)cast notation.

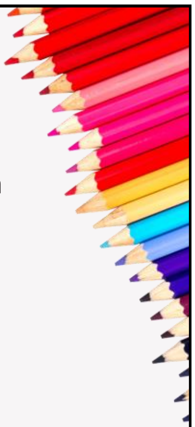
Casting Objects and the instanceof Operator

- Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?
- 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.
- 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.*
- It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting.

the instanceof Operator

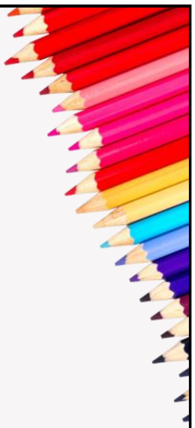
- Use the `instanceof` operator to test whether an object is an instance of a class:

```
void someMethod(Object myObject) {
  ... // 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());
    ...
  }
}
```



Why Casting Is Necessary?

- The variable myObject is declared Object. The declared type decides which method to match at compile time.
- Using myObject.getDiameter() would cause a compile error, because the Object class does not have the getDiameter method. The compiler cannot find a match for myObject.getDiameter().
- Therefore, it is necessary to cast myObject into the Circle type to tell the compiler that myObject is also an instance of Circle.
- Why not define myObject as a Circle type in the first place?
- To enable generic programming, it is a good practice to define a variable with a supertype, which can accept an objectof any subtype.



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.

CastingDemo

Run

11.10 The Object's equals Method

- equals method tests whether two objects are equal. Its signature is

  `public boolean equals(Object o)`

- The syntax for invoking it is: `object1.equals(object2);`
- The default implementation of the equals method in the Object class is:

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

- This implementation checks whether two reference variables point to the same object using the **== operator.** You should override this method in your custom class to test whether two distinct objects have the same content.

The equals Method

- The equals() method compares the contents of two objects. For example, the equals method is overridden in the Circle class.

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

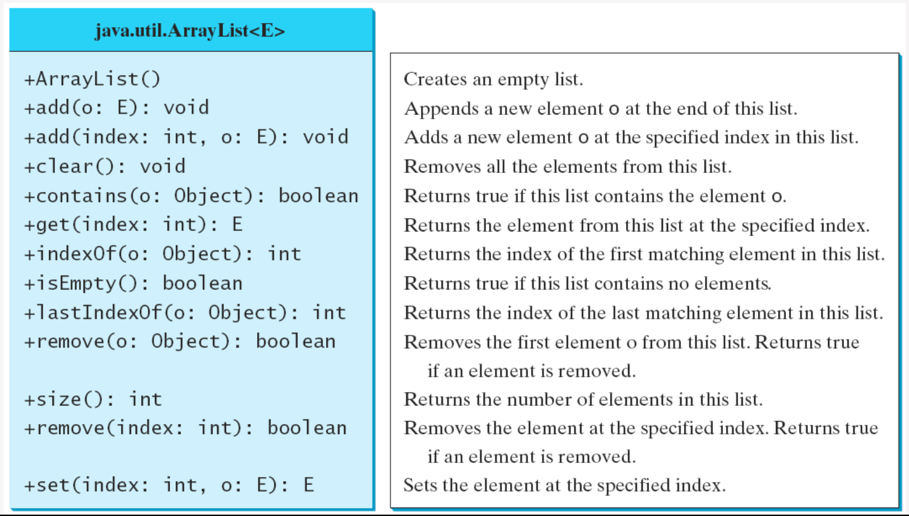
48

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.

11.11 The ArrayList Class

- *An **ArrayList** object 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.

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

TestArrayList   Run

NOTE

- Since JDK 7, the statement

```
ArrayList <AConcreteType> list = new
ArrayList<AConcreteType>();
```

can be simplified by

```
ArrayList<AConcreteType> list = new ArrayList<>();
```

- The concrete type is no longer required in the constructor, thanks to a feature called *type inference*. The compiler is able to infer the type from the variable declaration.

Differences and Similarities between Arrays and 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();`

DistinctNumbers

Run

11.12 Useful Methods for Lists

- *Java provides the methods for creating a list from an array, for sorting a list, and finding maximum and minimum element in a list, and for shuffling a list.*
- methods in:
  - Arrays class
  - ArrayList class
  - java.util.Collections

ArrayLists to/from Arrays

- Create an array list from an array: Arrays.asList

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

  - The static method **asList** in the **Arrays** class returns a list that is passed to the ArrayList constructor for creating an ArrayList.

- Create an array of objects from an array list: toArray

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

  - Invoking **list.toArray(array1)** copies the contents from list to array1.

java.util.Collections.sort method

- If the elements in a list are comparable such as integers, double, or strings, you can use the static **sort** method in the **java.util.Collections** class to sort the elements.

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.sort(list);
System.out.println(list);
```

max and min methods

- You can use the static **max** and **min** in the **java.util.Collections** class to return the maximum and minimal element in a list.

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
System.out.println(java.util.Collections.max(list));
System.out.println(java.util.Collections.min(list));
```

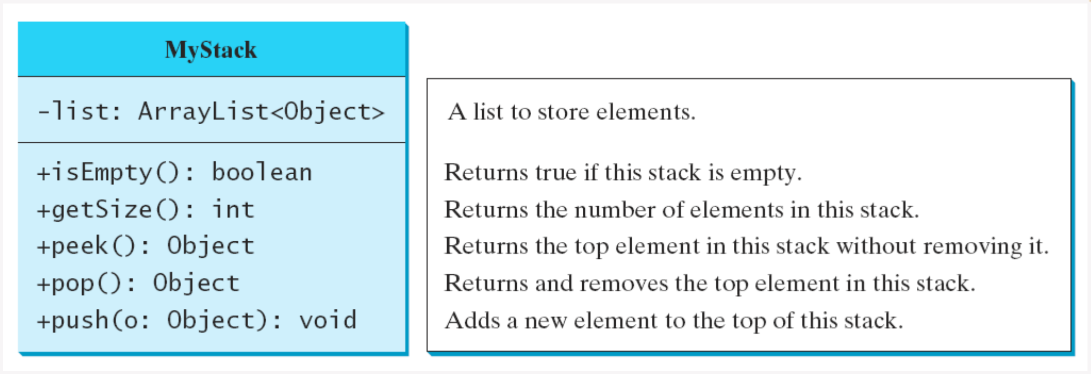
shuffle method

- You can use the static **shuffle** method in the **java.util.Collections** class to perform a random shuffle for the elements in a 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);
```

11.13 Case Study: A Custom Stack Class

- A stack to hold objects.



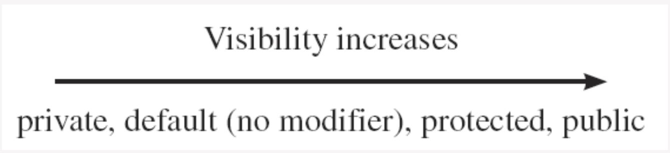
MyStack	
-list: ArrayList<Object>	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack without removing it.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.

MyStack

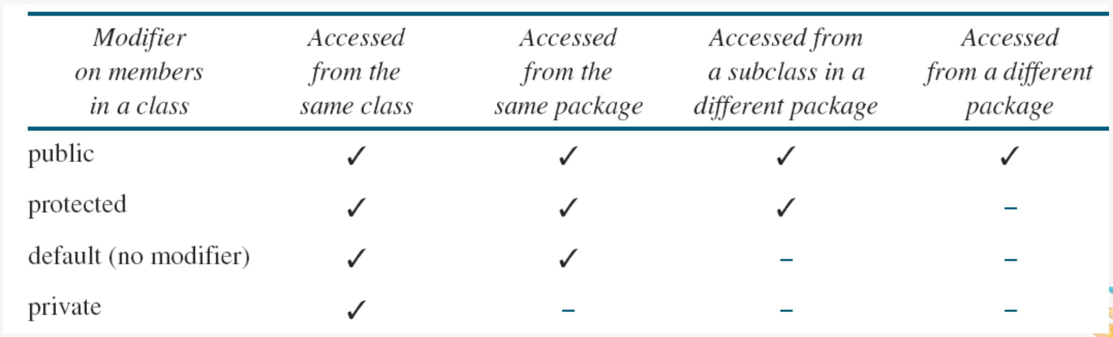
59

11.14 The protected Data and Methods

- The **`protected`** modifier can be applied on *data and methods* in a class.
- *A protected member of a class can be accessed from a subclass.*
  - 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`**



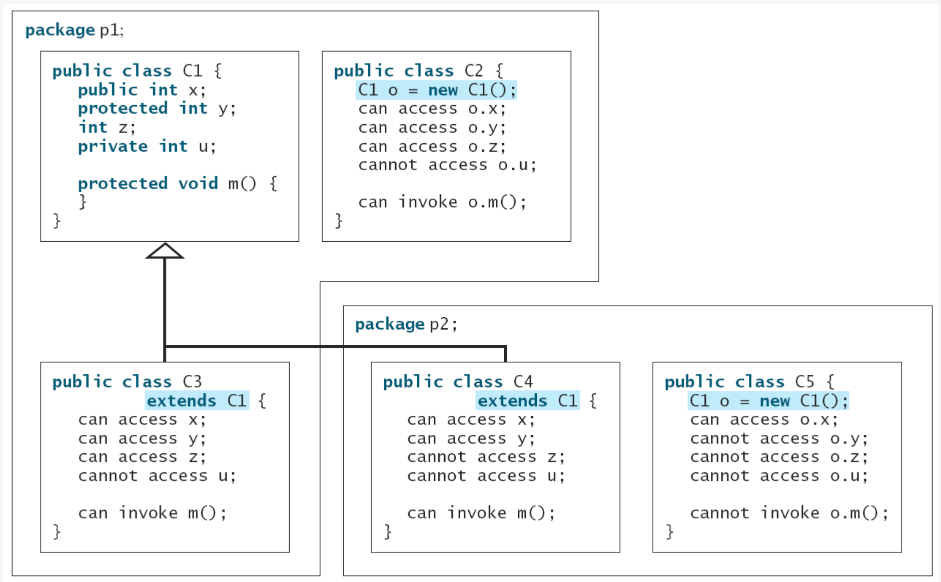
Accessibility Summary



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

61

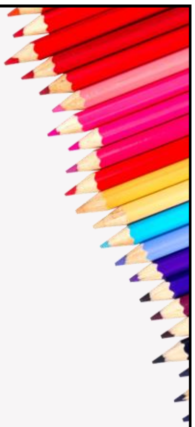
Visibility Modifiers



```
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();
}
```

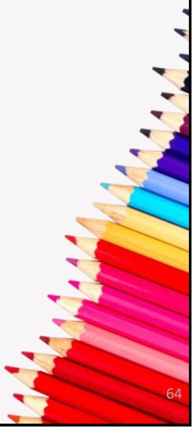
```
package p2;

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();
}
```

62



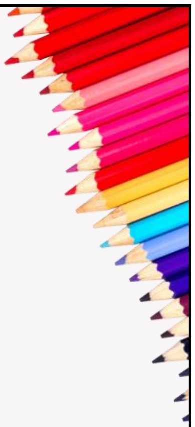
Visibility Modifiers

- 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.
- The **private and protected** modifiers can be used only for members of the class.
- The **public** modifier and the default modifier (i.e., no modifier) can be used on members of the class as well as on the class.



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.



11.15 Preventing Extending and Overriding

- *Neither a final class nor a final method can be extended. A final data field is a constant.*
- 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.

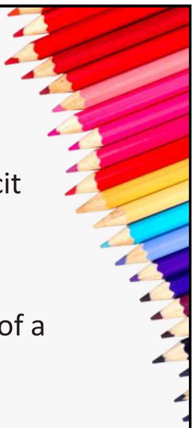
Chapter Summary

Chapter Summary

- You can define a new class from an existing class. This is known as class *inheritance.* The new class is called a *subclass, child class, or extended class. The existing class is* called a *superclass, parent class, or base class.*
- Every class in Java is descended from the **java.lang.Object** class.
- The constructors of a superclass are not inherited in the subclass. They can be invoked only from the constructors of the subclasses, using the keyword **super.**
- A constructor may invoke an overloaded constructor or its superclass's constructor. The call must be the first statement in the constructor. If none of them is invoked explicitly, the compiler puts **super()** as the first statement in the constructor, which invokes the superclass's no-arg constructor.

Chapter Summary

- To *override* a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.
- If a method's parameter type is a superclass you may pass an object to this method of any of the parameter's subclasses. This is known as polymorphism.
- When invoking an instance method from a reference variable, the *actual type of the* variable decides which implementation of the method is used *at runtime. This is known* as dynamic binding.
- It is always possible to cast an instance of a subclass to a variable of a superclass, because an instance of a subclass is always an instance of its superclass.



Chapter Summary

- When casting an instance of a superclass to a variable of its subclass, explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation.
- You can use **obj instanceof AClass** to test whether an object is an instance of a class.
- You can use the **ArrayList** class to create an object to store a list of objects**.**
- You can use the **protected** modifier to prevent the data and methods from being accessed by nonsubclasses from a different package.
- You can use the **final** modifier to indicate that a class is final and cannot be extended and to indicate that a method is final and cannot be overridden.

Programming Exercises

1, 2, 3, 4, 5, 7, 8,

12, 13, 14, 18