Inheritance & Polymorphism

Dr. Abdallah Karakra |Comp 2311 | Masri504

4/20/2023

BIRZEIT UNIVERSITY

Index

Chapter 11

- All Sections (11.1 – 11.15)

CHAPTER 11

Inheritance and Polymorphism

Motivations

- Suppose you will define classes to model circles, rectangles, and triangles. These geometric classes have many common features.

- *What is the best way to design these classes so to avoid redundancy?*

  The answer is to use object **inheritance**.

Example: Shared Functionality

```
public class Student {
 private String name;
 private int age;
 private int studentNumber;

  //public constructor
  //public setter and getter methods (name && age)
  public int getStudentNumber (){
        return studentNumber;
    }
  public void setStudentNumber (int studentNumber){
       this.studentNumber = studentNumber;
    }

    ….
}
```

Example: Shared Functionality Cont.

```
public class Employee {
   private String name;
   private int age;
   private double salary;
   private String departmentName;
    //public constructor
    //public setter and getter methods (name && age)
    public double getSalary (){
          return salary;
   }
   public void setSalary (double salary){
         this.salary = salary;
   }
   public String getDepartmentName() {
      return departmentName;
   }
   public void setDepartmentName(String departmentName) {
     this.departmentName = departmentName;
   }
    ….
}
```

```
public class Person {
 private String name;
 private int age;
 // constructor
 public String getName() {return name;}
 public void setName(String name){this.name=name;}

 public int getAge() {return age;}
 public void setAge(int age){this.age=age;}
   …
}
```

General class

Inherit

```
public class Student extends Person {
 private int studentNumber;
 //constructor
 public int getStudentNumber (){return studentNumber;}
 public void setStudentNumber(int studentNumber)
 {this.studentNumber = studentNumber;}

}
```

specific class

```java
public class Person {
 private String name;
 private int age;
 // constructor
 public String getName() {return name;}
 public void setName(String name){this.name=name;}
 public int getAge() {return age;}
 public void setAge(int age){this.age=age;}
   …
}
```

General class

Inherit

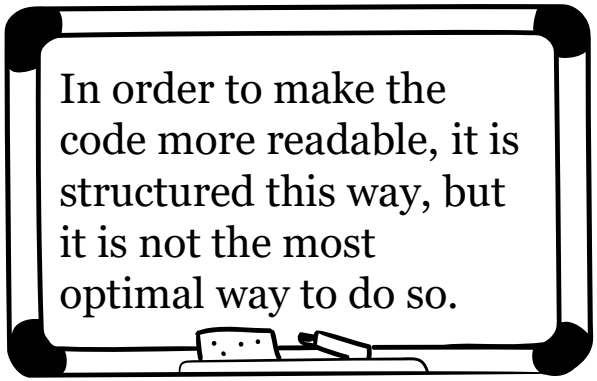
```java
public class Employee extends Person {
  private double salary;
  private String departmentName;
  public double getSalary() {return salary;}

  public void setSalary(double salary) {this.salary =salary;}

  public String getDepartmentName() {return departmentName;}

  public void setDepartmentName(String departmentName)
  {this.departmentName = departmentName;}
}
```

specific class

Next, a detailed description of each class is provided 

```java
public class Person {

   private String name;
   private int age;
   public Person() {
     name = "No name yet";
     age=0;
  }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() { return name; }

  public void setName(String name) { this.name = name;}

  public int getAge() { return age; }

  public void setAge(int age) { this.age = age; }

  public void writeOutput() {
    System.out.println("Name: " + name + "Age: "+age);
  }

}
```

this("No name yet",0)



```java
public class Student extends Person
{
    private int studentNumber;
    public Student (){
        super ();
        studentNumber = 0; //Indicating no number yet
    }

    public Student (String name,int age ,int studentNumber){
        super (name,age);
        this.studentNumber = studentNumber;
    }

    public void reset (String name, int age ,int studentNumber){
        setName (name);
        setAge(age);
        this.studentNumber = studentNumber;
    }

    public int getStudentNumber (){return studentNumber;}

    public void setStudentNumber(int studentNumber) {this.studentNumber = studentNumber;}

    public void writeOutput ()
    {
        System.out.println ("Name: " + getName());
        System.out.println ("Age: " +  getAge());
        System.out.println ("Student Number: " + studentNumber);
    }


}
```

```java
public String getName() {return name;}

public void setName(String name) {this.name = name;}

public int getAge() {    return age;}

public void setAge(int age) {    this.age = age;}

public void writeOutput() {
    System.out.println("Name: " + name + "Age: "+age);
}
```

```java
public class Employee extends Person {
    private double salary;
    private String departmentName;

    public Employee() {
        super();
        salary = 0.0; // Indicating no salary yet
        departmentName = "No name yet";}

    public Employee(String name, int age, double salary, String departmentName) {
        super(name, age);
        this.salary = salary;
        this.departmentName = departmentName;}

    public void reset(String name, int age, double salary, String departmentName) {
        setName(name);
        setAge(age);
        this.salary = salary;
        this.departmentName = departmentName;}

    public double getSalary() {return salary;}
    public void setSalary(double salary) {this.salary = salary;}
    public String getDepartmentName() {return departmentName;}
    public void setDepartmentName(String departmentName) {
        this.departmentName = departmentName;}

    public void writeOutput() {
        System.out.println("Name: " + getName());
        System.out.println("Age: " + getAge());
        System.out.println("Employee Salary: " + salary);
        System.out.println("Employee Department Name: " + departmentName);}
}
```

```java
public String getName() {return name;}

public void setName(String name) {this.name = name;}

public int getAge() {    return age;}

public void setAge(int age) {    this.age = age;}

public void writeOutput() {
    System.out.println("Name: " + name + "Age: "+age);
}
```

Inheritance (The “is-a” relationship)

The ability of a class to derive properties and behaviours from a previously defined class.

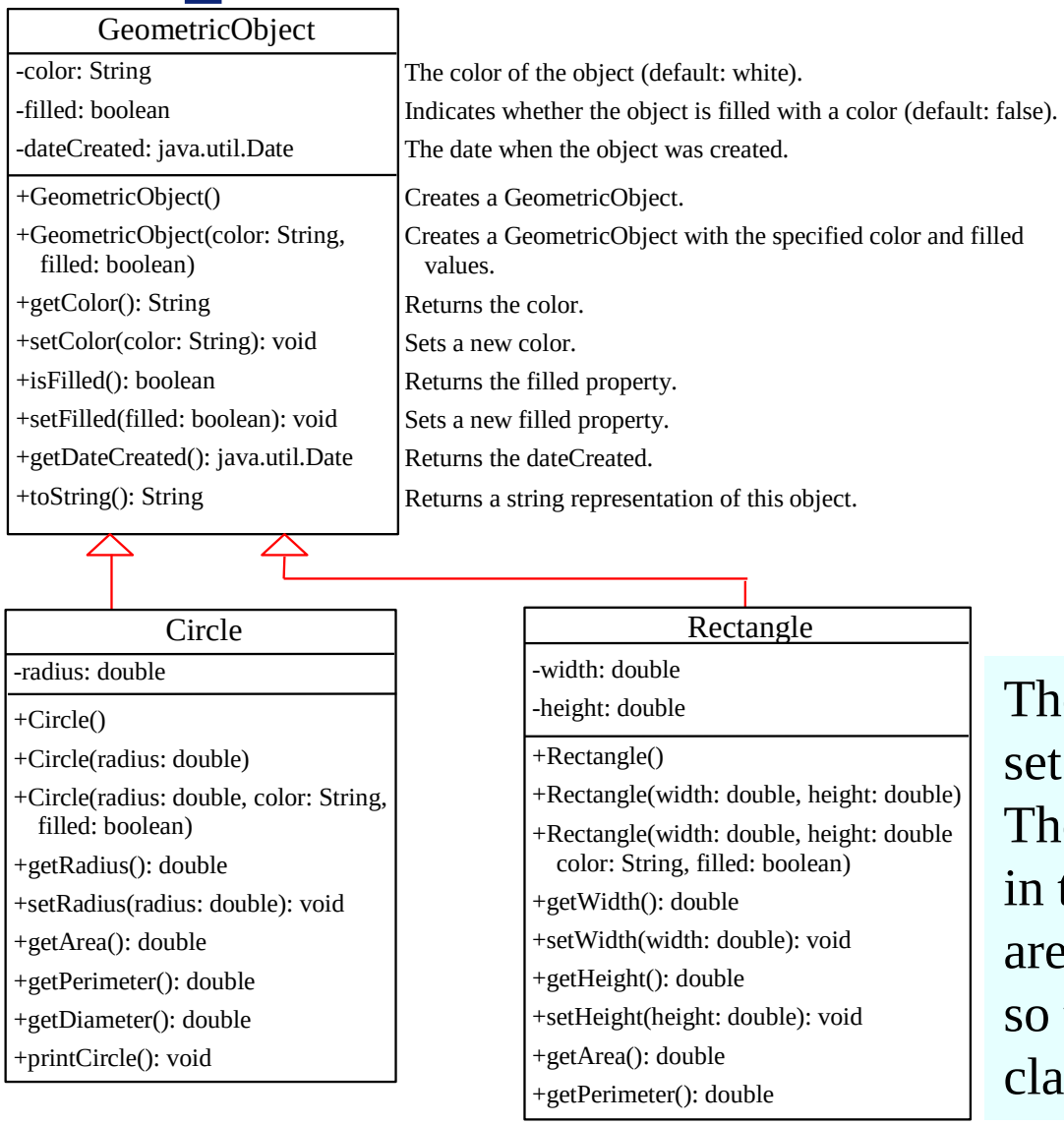
- Some classes have some common properties & behaviors, which can be generalized in a class that can be shared by other classes

- We can define a new specialized class (subclass) that extends the existing generalized class (superclass). This is called class *inheritance*

- The subclasses inherit the properties & methods (except constructors) from the superclass

```
class Fruit {
  // code
}

class Apple extends Fruit {
 // code
}
```

Superclasses and Subclasses



The **setColor** and **setFilled** methods to set the **color** and **filled** properties. These two public methods are defined in the base class **GeometricObject** and are inherited in **Circle nad Rectangle**, so they can be used in the derived class.

Are Superclass's Constructor Inherited?

No. They are invoked explicitly or implicitly.

- They can be invoked from the subclasses' constructors, using the keyword `super`
- If the keyword `super` is not explicitly used, the superclass's *accessible no-arg constructor* is automatically invoked

A constructor is used to construct an instance of a class.

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
     - **super(The parameters list, if any);**
  2. To call a superclass method
     - **super.method(The parameters list, if any);**
- 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 A() {
}
```

is equivalent to

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

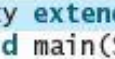
```
public A(double d) {
  // some statements
}
```

is equivalent to

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

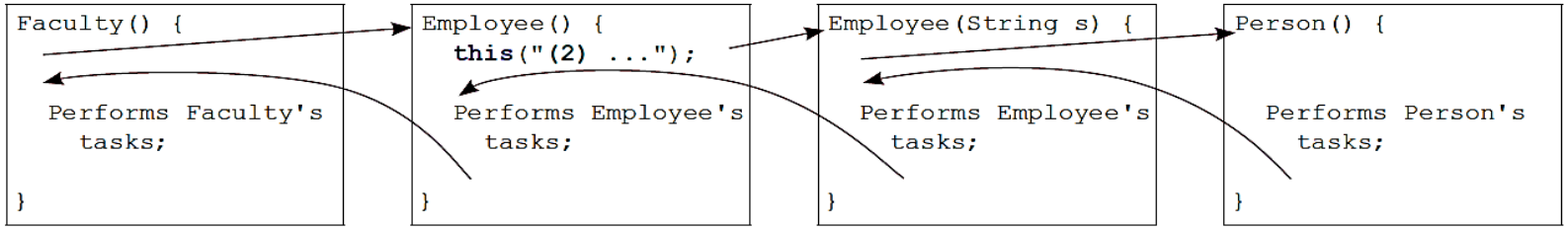
Constructor Chaining

```
1  public class Faculty extends Employee {
2    public static void main(String[] args) {
3      new Faculty();
4    }
5
6    public Faculty() {
7      System.out.println("(4) Performs Faculty's tasks");
8    }
9  }
10
11 class Employee extends Person {
12   public Employee() {
13     this("(2) Invoke Employee's overloaded constructor");
14     System.out.println("(3) Performs Employee's tasks ");
15   }
16
17   public Employee(String s) {
18     System.out.println(s);
19   }
20 }
21
22 class Person {
23   public Person() {
24     System.out.println("(1) Performs Person's tasks");
25   }
26 }
```



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

This is called *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");
  }
}
```

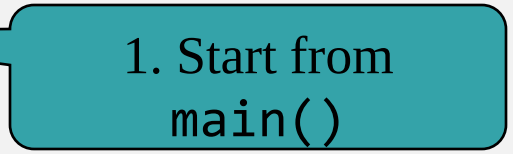
```
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee’s overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```

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

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");
  }
}
```

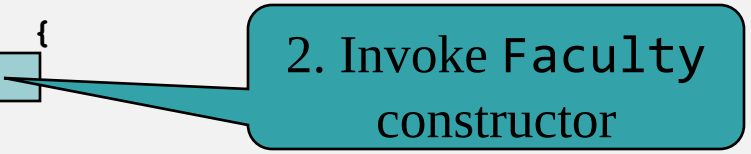



```
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

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

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");
  }
}
```

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");
  }
}
```

3. Invoke `Employee's` *no-arg constructor*

```
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");
  }
}
```

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

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

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`

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`

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`

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");
  }
}
```

9. Execute `println`

```
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");
  }
}
```

Example of the Impact of a Superclass without a no-arg constructor

Find out the error in the following program:

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

public class Apple extends Fruit {
}
```

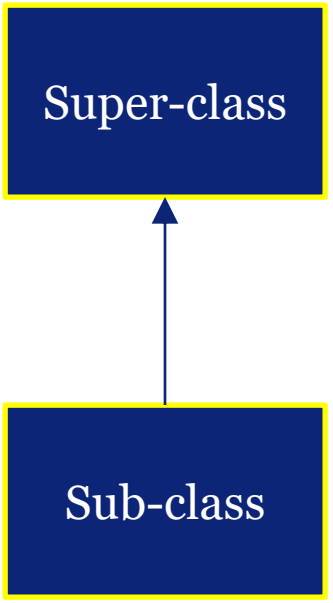
Design Guide
If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors

A compile-time error will be generated indicating the need for a *no-arg constructor* in the superclass

Properties & Methods of Subclasses

- A subclass inherits properties and methods (except constructors) from a superclass

- You can also:
  - Add new properties to the subclass
  - Add new methods to the subclass
  - Override (i.e. redefine) the implementation of the methods defined in the superclass.

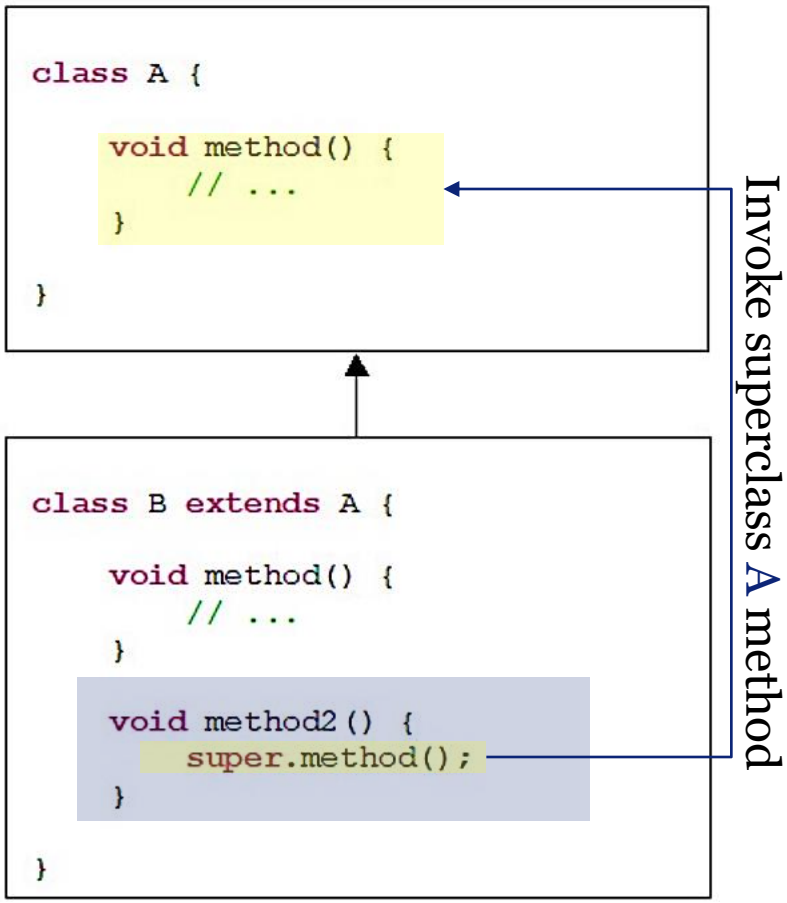


Example: Calling Superclass Methods

- The figure shows access to the superclass A method from subclass B.

- In subclass B there is a method with the same name and a list of parameters, therefore, this method overrides the method of the superclass. To access the superclass A method in the subclass B method, the keyword super is used.

- Suppose super is omitted from methode2 in subclass B. What is the consequence?



Overriding Methods in the Superclass

- 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**.

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.

Overriding vs. Overloading

Same name/arguments, different body

```
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);
  }
}
```

Output
10.0
10.0

Same name, different arguments/body

```
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);
  }
}
```

Output
10
20.0

Check Point

```
public class Test {
  public static void main(String[] args) {
    A x= new A();
    x.printOutput();
    new B().printOutput();
  }
}

class B extends A {

  public int getNum() {
    return 4;
  }
}

class A {
  public int getNum() {
    return 5;
  }

  public void printOutput() {
    System.out.println(getNum());
  }
}
```

```
5
4
```

Check Point

```
public class Test {
  public static void main(String[] args) {
      new A().printOutput();
    new B().printOutput();
  }
}

class B extends A {

  private int getNum() {
    return 4;
  }
}

class A {
  private int getNum() {
    return 5;
  }

  public void printOutput() {
    System.out.println(getNum());
  }
}
```

```
5
5
```

Check Point

True or false? A subclass is a subset of a superclass.

False.
A subclass is an extension of a superclass and normally contains more details information than its superclass.

What keyword do you use to define a subclass?

The extends keyword is used to define a subclass that extends a superclass.

What is single inheritance? What is multiple inheritance? Does Java support multiple inheritance?

Single inheritance allows a subclass to extend only one superclass. Multiple inheritance allows a subclass to extend multiple classes. Java does not allow multiple inheritance.

The `Object` Class and its Methods

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

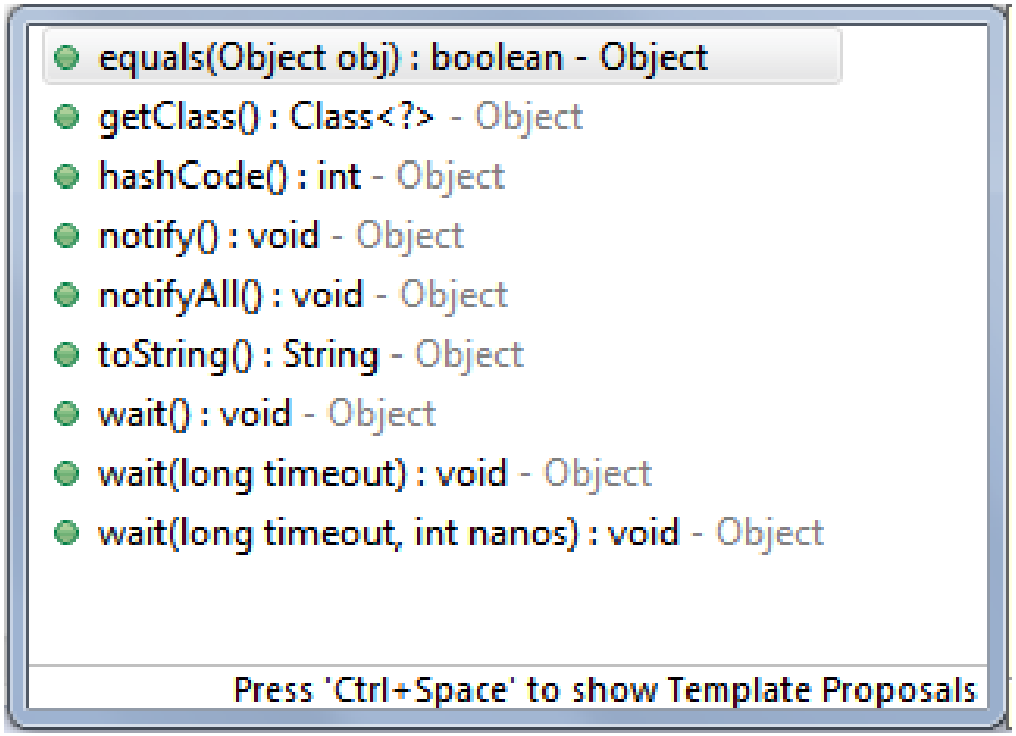
- If no inheritance is specified when a class is defined, the superclass of the class is `Object`

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

Equivalent

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

The Object Class and Its Methods



The toString() method in Object

- `toString()` method returns a string representation of the object.

- The default implementation returns a string holding:

1. a class name of which the object is an instance,
2. the at sign @, and
3. a number (hashcode) representing this object.

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

For this example we get something like: `Loan@15037e5`

You should override the `toString` *method so that it returns a more meaningful string representation of the object.*

toString

```
public class Test{

    public static void main (String [] args){
     Test testObject= new Test();
     System.out.println(testObject.toString());//Test@1db9742
     System.out.println(testObject);//Test@1db9742
   }

}
```

toString

```
public class Test extends Test2{

    public static void main (String [] args){
     Test testObject= new Test();
     System.out.println(testObject.toString());//Hello from Test2
     System.out.println(testObject);//Hello from Test2
   }

}


class Test2{

    public String toString(){
      return "Hello from Test2";
     }

}
```

Polymorphism

Key Point 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**.
- Every Circle is a geometric object, but not every geometric object is a circle.

Key Point **The point is:** *you can always pass an instance of a subclass to a parameter of its superclass type*

```
public class Test{

     public static void main (String []args){
        D d = new D();
        C c= new C();
        C a=new A();
        C b =new B();
        d.poly(c);
        d.poly(a);
        d.poly(b);
     }

}
```

```
class A extends C{

  public void print(){
     System.out.println("Hello, From A");
   }
}
```

```
class B extends C{

  public void print(){
     System.out.println("Hello, From B");
   }

}
```

```
class C {

    public void print(){
     System.out.println("Hello, From C");
   }
}
```

```
class D {

     public void poly (C obj){
       obj.print();
    }
}
```

Hello, From C

Hello, From A

Hello, From B

Example – Polymorphism and Dynamic Binding

```
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 {
  @Override
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  @Override
  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*.

- 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***.

Output will be:
Student
Student
Person
java.lang.Object@15db9742

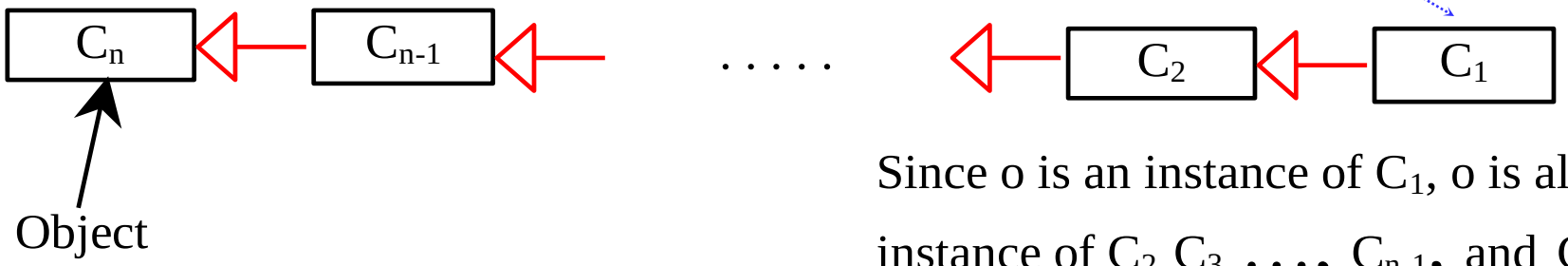
Dynamic Binding

- A method can be implemented in several classes along the inheritance chain. The Java Virtual Machine (JVM) decides which method is invoked at runtime.

- 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 `p()`, the JVM searches the implementation for the `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

The compiler finds a *matching* method according to:

1. parameter type,
2. number of parameters, and
3. order of the parameters at compilation time.

Example of different invocation of the pay method

```
bill.payWith ( 19.99 )

bill.payWith ("Visa", "1234 1234 1234 1234" );

bill.payWith ("two cows" );
```

Method Matching vs. Binding

- **Matching a method signature** and **binding a method implementation** are two issues.
- Per the ***declared type*** of the reference variable, the compiler finds a matching method according to (1) parameter type, (2) number of parameters, and (3) order of the parameters at **compilation time**.
- A method may be implemented in several subclasses along the inheritance chain.
- The Java Virtual Machine dynamically binds the implementation of the method at **runtime** per the ***actual type*** of the variable.

Given: Object o = new GeometricObject( );

In short: *The declared type of the reference variable decides which method to match at compile time and the actual type decides which specific method to implement during 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.
- When an object (e.g., a Student 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 used the casting operator to convert variables of one primitive type to another.

Example: (int)(x / y);

- *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In previous slide, 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?

1. To tell the compiler that an object should be treated as of a particular type you use an *explicit casting*.

2. In the example below, *msg* is a block of binary data that could be interpreted in different ways depending on the casting applied on it.

```
Object msg = ChunkOfBinaryData();

Voice v = (Voice) msg;
 ...
Sms   s = (Sms)   msg;
 ...
Email e = (Email) msg;
 ...
Morse m = (Morse) msg;
...
```

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 compile error would occur. 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. 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 y = (Orange)fruit;

float f = 123.45;

int n = (int) f;
```

Observation:
Assume fruit is a "banana". Both castings will fail.
The numeric example works.

The `instanceof` Operator

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

```
Object myObject = new Circle();

// Some lines of code here . . .

// Perform casting if myObject is an instance of Circle
if (myObject instanceof Circle) {
   System.out.println("The circle diameter is " +
                      ( (Circle)myObject ).getDiameter() );
  ...

}
```

```
public class Test{

    public static void main (String [] args){
        Object obj= new Circle();
        Circle cl = new Circle();
        Rectangle  rect = new Rectangle ();
        Shape sh =new Rectangle();
        boolean res=obj instanceof Circle;
        boolean res2=rect instanceof Rectangle;
        boolean res3=obj instanceof Rectangle;
        boolean res4=cl instanceof Object;
        boolean res5=cl instanceof Shape;
        System.out.println(res+ " "+ res2+ " "+res3+ " "+res4+ " "+res5);
    }


}
class Shape {

}

class Circle extends Shape {

}

class Rectangle extends Shape{

}
```

true true false true true

Example: Demonstrating Polymorphism and Casting

- displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

LISTING 11.7 CastingDemo.java

```
1  public class CastingDemo {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create and initialize two objects
5      Object object1 = new Circle4(1);
6      Object object2 = new Rectangle1(1, 1);
7
8      // Display circle and rectangle
9      displayObject(object1);
10     displayObject(object2);
11   }
12
13   /** A method for displaying an object */
14   public static void displayObject(Object object) {
15     if (object instanceof Circle4) {
16       System.out.println("The circle area is " +
17         ((Circle4)object).getArea());
18       System.out.println("The circle diameter is " +
19         ((Circle4)object).getDiameter());
20     }
21     else if (object instanceof Rectangle1) {
22       System.out.println("The rectangle area is " +
23         ((Rectangle1)object).getArea());
24     }
25   }
26 }
```

polymorphic call

polymorphic call

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;
}
```

This implementation checks whether two reference variables point to the same object using the == operator

For example, the equals method is overridden in the Circle class.

```
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.

Example: equals

```
public class TestEquals{

  public static void main (String [] args){

     Circle c1= new Circle (3);
     Circle c2= new Circle (3);
     Rectangle r1= new Rectangle (1,2);
     Rectangle r2= new Rectangle (1,2);
     System.out.println(c1.equals(c2));
     System.out.println(r1.equals(r2));
  }
}
```

```
class Circle{

  private double radius;

  //Setter && getter Method

  public Circle (double radius){

      this.radius=radius;
  }
}
```

```
class Rectangle {

     private double length,width;
     public Rectangle (double length, double width){
        this.length=length;
        this.width=width;
     }
     public boolean equals (Object o){
        if (o instanceof Rectangle){
            if (this.length== ((Rectangle)o).length && this.width== ((Rectangle)o).width )
               return true;
        }

        return false;
     }
}
```

false
true

Useful Predefined Classes: The ArrayList

- 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.

An ArrayList object can be used to store a list 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 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>();
```

Either syntax will work

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

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();`

Example

```
import java.util.ArrayList;
public class TestArrayList{

    public static void main(String []args){
       ArrayList<String> list = new ArrayList<String>();  // [ ]
       System.out.println(list.size()); //0
       list.add("Hello"); // [Hello]
       list.add("Hi"); // [Hello,Hi]
       System.out.println("Size is "+list.size() +list);//Size is 2[Hello, Hi]
       list.add(1,"welcome");//[Hello,welcome,Hi]
       System.out.println("Size is "+list.size() +list);//Size is 3[Hello, welcome, Hi]
       list.set(1,"Salam");//{Hello, Salam, Hi}
       System.out.println("Size is "+list.size() +list);//Size is 3[Hello, Salam, Hi]
       String s = list.get(0);//Hello
       System.out.println(s);//Hello
       list.remove("Hi");//[Hello, Salam]
       System.out.println("Size is "+list.size() +list);
       list.clear();//[]
       System.out.println("Size is "+list.size() +list);//Size is 0[]

    }
}
```

Check Point

Suppose you want to create an **ArrayList** for storing integers. Can you use the following code to create a list?

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

No. This will not work because the elements stored in an ArrayList must be of an object type. You cannot use a primitive data type such as int to replace a generic type. However, you can create an ArrayList for storing Integer objects as follows:

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

LISTING 11.9 DistinctNumbers.java

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class DistinctNumbers {
5    public static void main(String[] args) {
6      ArrayList<Integer> list = new ArrayList<Integer>();
7
8      Scanner input = new Scanner(System.in);
9      System.out.print("Enter integers (input ends with 0): ");
10     int value;
11
12     do {
13       value = input.nextInt(); // Read a value from the input
14
15       if (!list.contains(value) && value != 0)
16         list.add(value); // Add the value if it is not in the list
17     } while (value != 0);
18
19     // Display the distinct numbers
20     for (int i = 0; i < list.size(); i++)
21       System.out.print(list.get(i) + " ");
22   }
23 }
```

```
Enter numbers (input ends with 0): 1 2 3 2 1 6 3 4 5 4 5 1 2 3 0 Enter
The distinct numbers are: 1 2 3 6 4 5
```

```
import java.util.ArrayList;
public class HelloWorld{
     public static void main(String []args){
       ArrayList<Student> students =new ArrayList <>();
       students.add (new Student("Sandy ",123));
       students.add (new Student("Sam ",567));
       System.out.println(students);
       /*  If toString does not override then output will be
           [Student@6d06d69c, Student@7852e922]
           else
           [Name is Sandy Id= 123, Name is Sam Id= 567]
       */
     }
}

class Student {

    private String name;
    private int id;

    Student (String name, int id){
        this.name=name;
        this.id=id;
    }
    //setter && getter Method
    public String toString(){
        return "Name is "+ name + "Id= "+id;
    }
}
```

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:

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

```
import java.util.ArrayList;
import java.util.Arrays;
public class HelloWorld{

    public static void main(String []args){
        String [] obj = {"Hello","Hi","Welcome"};
        ArrayList <String> list = new ArrayList <>(Arrays.asList(obj));
        System.out.println(list); //[Hello, Hi, Welcome]
    }
}
```

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)));
```

```
import java.util.ArrayList;
import java.util.Arrays;
public class HelloWorld{

    public static void main(String []args){
        String [] obj = {"Hello","Hi","Welcome"};
        ArrayList <String> list = new ArrayList <>(Arrays.asList(obj));
        System.out.println(list); //[Hello, Hi, Welcome]
        System.out.println(java.util.Collections.max(list));//Welcome
        System.out.println(java.util.Collections.min(list));//Hello
    }
}
```

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);
```

[95, 4, 5, 6, 34, 3, 5, 15, 3]

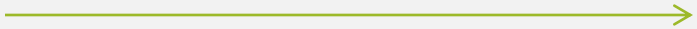
[6, 5, 5, 4, 3, 15, 3, 34, 95]

[6, 3, 4, 95, 15, 3, 5, 34, 5]

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class.
- A `protected` data or 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 (no modifier is used) ⟶ protected ⟶ public

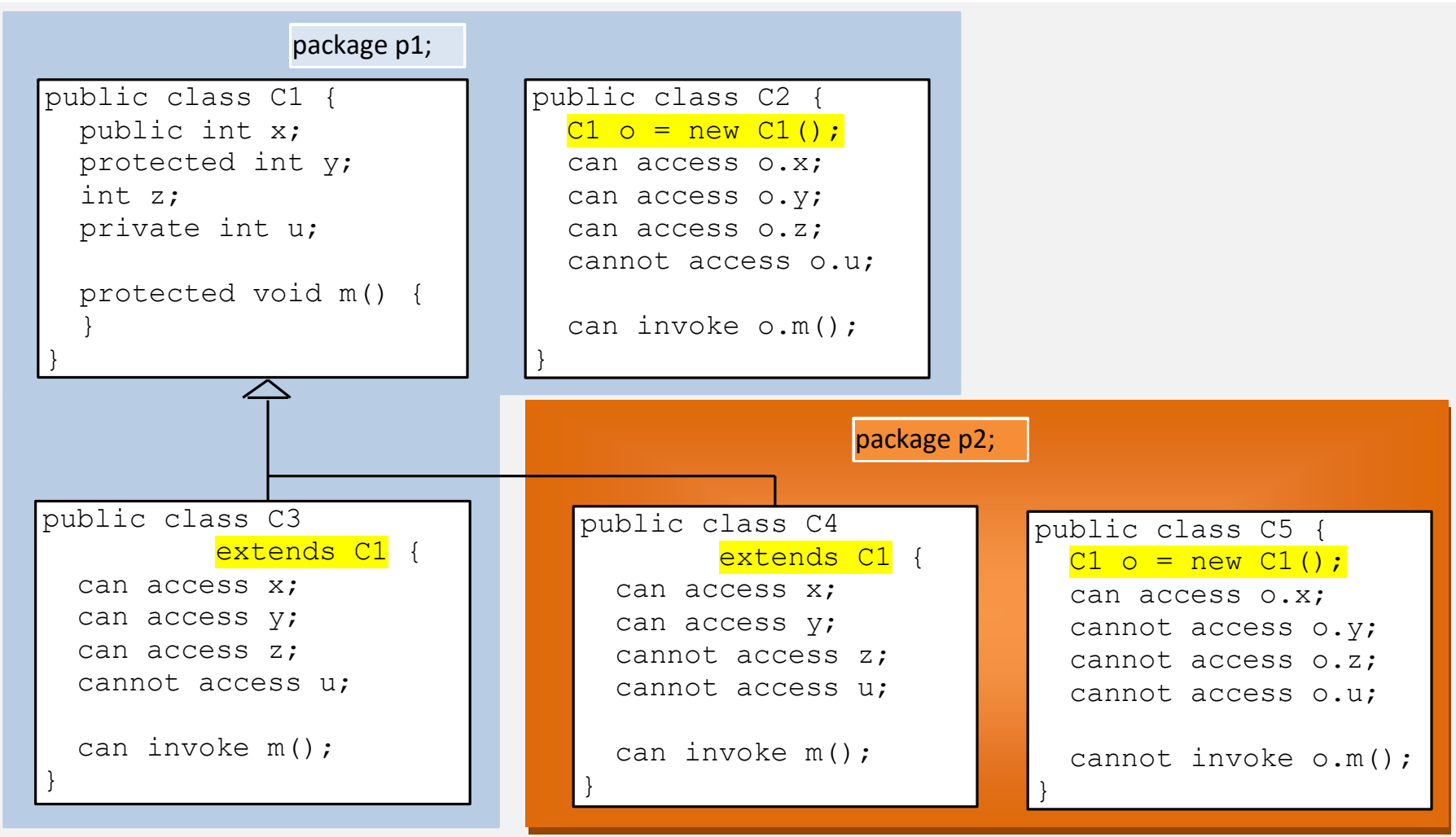


Visibility increases

Accessibility Summary

The notations used in UML	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	Yes	Yes	Yes	Yes
#	protected	Yes	Yes	Yes	No
	default	Yes	Yes	No	No
-	private	Yes	No	No	No

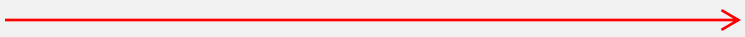
Visibility Modifiers



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.

private ⟶ default (no modifier is used) ⟶ protected ⟶ public



Changes of visibility are valid in this direction ⟶
Changes of visibility are invalid in this direction ⟵

final NOTE

- Modifiers (`private, public, protected, default`) are used on classes, methods, and class variables.
- The `final` modifier can also be used on local variables in a method.
- A `final` local variable is **a *constant* inside a method**.
- A `final` class is one that **cannot be extended by sub-classing**.
- A `final` method **cannot be overridden by its subclasses**.

The `final` Modifier

The `final` class cannot be extended:

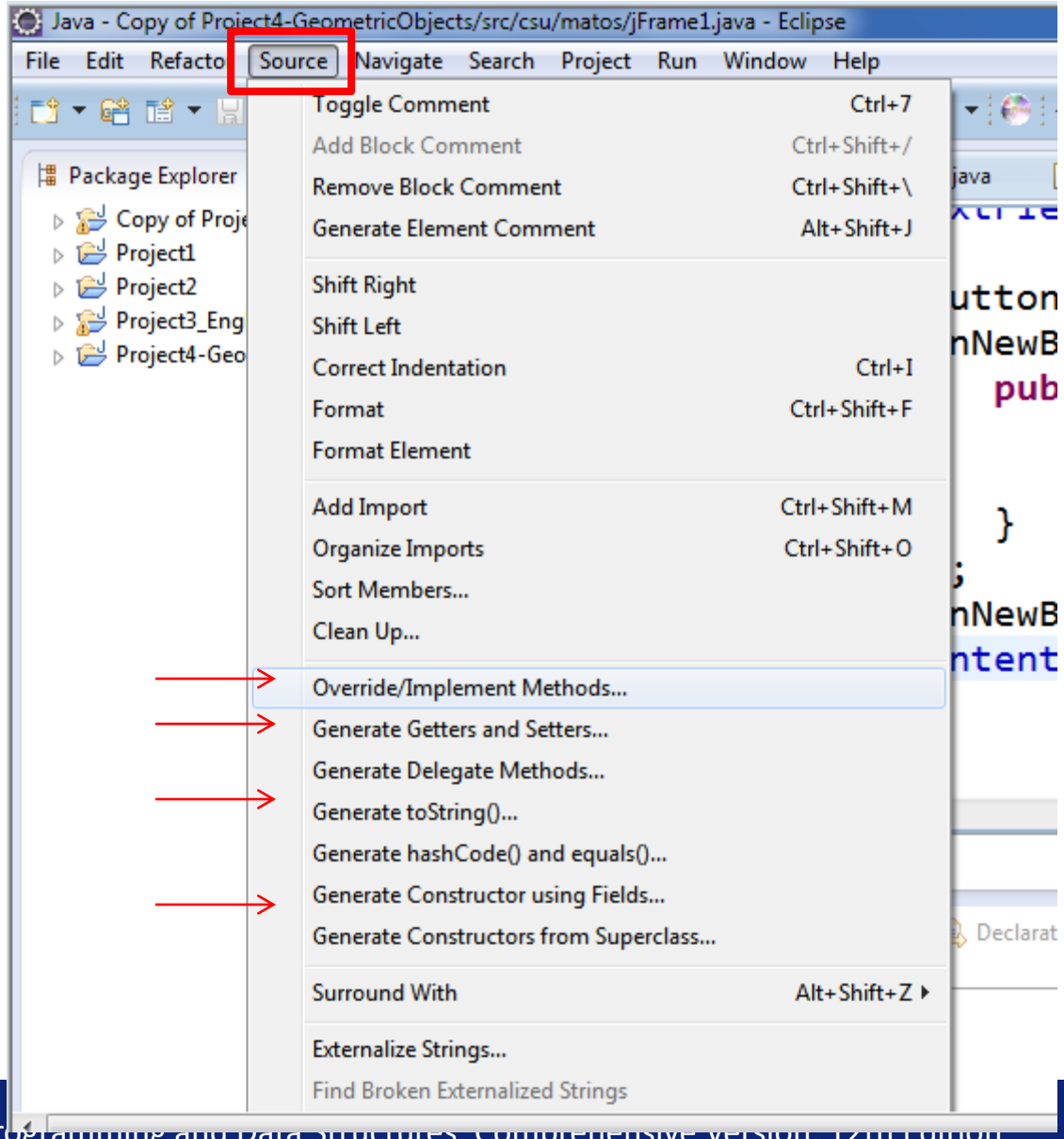
```
final class Math {
  ...
}
```

The `final` variable is a constant:

```
final static double PI = 3.14159;
```

The `final` method cannot be overridden by its subclasses.

Appendix A. Using Eclipse Tool Bar to code a POJO (Plain Old Java Object)



Check Point

How do you prevent a class from being extended? How do you prevent a method from being overridden?

Use the final keyword.

Check Point

Indicate true or false for the following statements:
a. A protected data field or method can be accessed by any class in the same package.
b. A protected data field or method can be accessed by any class in different packages.
c. A protected data field or method can be accessed by its subclasses in any package.
d. A final class can have instances.
e. A final class can be extended.
f. A final method can be overridden.

a. True.
b. False. (But yes in a subclass that extends the class where the protected data field is defined.)
c. True.
d. Answer: True
e. Answer: False
f. Answer: False

Inheritance & Polymorphism