

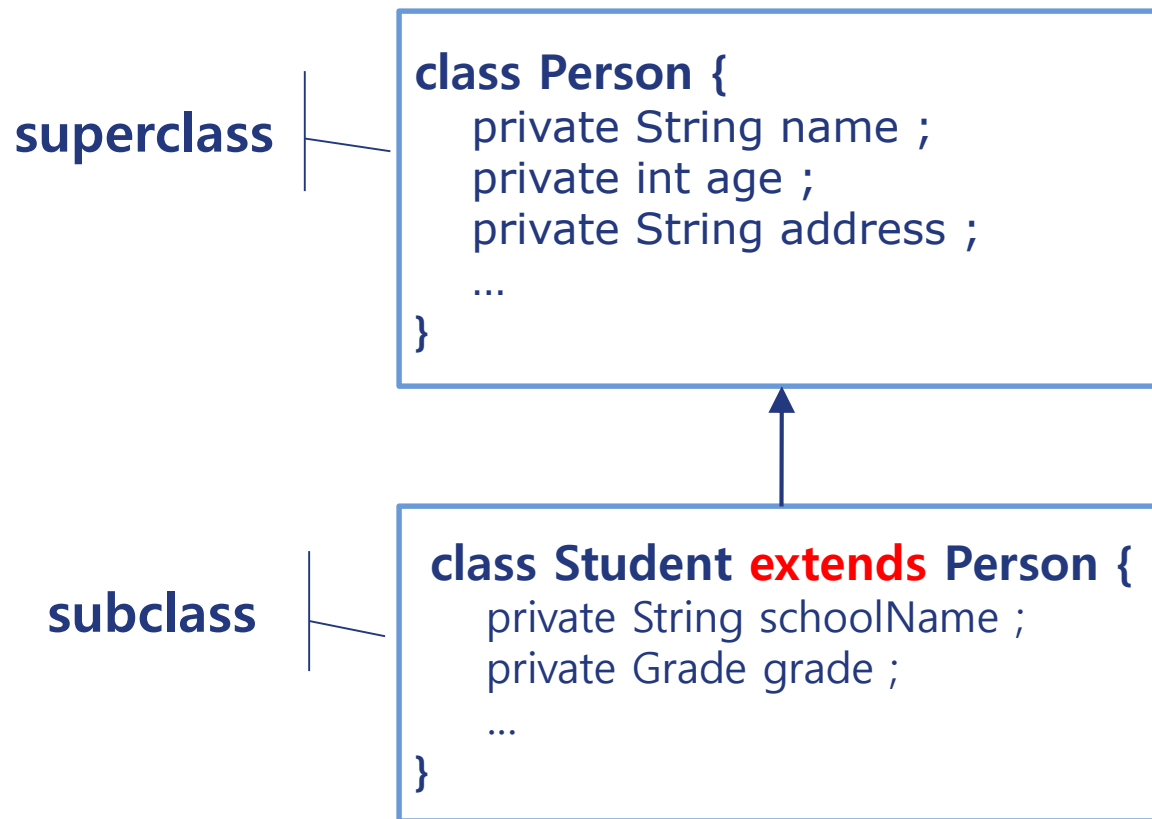
Inheritance

- ❖ Inheritance – concept
- ❖ super keyword
- ❖ Protected members
- ❖ Constructions of subclass
- ❖ Polymorphism
- ❖ Abstract classes and abstract methods
- ❖ Final classes and final methods



Inheritance

- ❖ A **subclass** inherits all the members of the **superclass**.



class Person

```
class Person {  
    private String name ;  
    private int age ;  
    private String address ;  
  
    public Person(String name, int age, String address) {  
        this.name = name ; this.age = age ; this.address = address ;  
    }  
    public String getName() { return name ; }  
    public void rename(String name) { this.name = name ; }  
    public int getAge() { return age ; }  
    public void increaseAge() { age ++ ; }  
    public String getAddress() { return address ; }  
    public void moveTo(String address) { this.address = address ; }  
    @Override  
    public String toString() {  
        return String.format("%s, %d, %s", name, age, address) ;  
    }  
    // override hashCode() and equals()  
}
```

class Student

```
public class Student extends Person {
    private String schoolName ;
    private Grade grade = new Grade();

    public Student(String name, int age, String address, String schoolName) {
        super(name, age, address) ;
        this.schoolName = schoolName ;
    }
    public String getSchoolName() { return schoolName ; }
    public void setSchoolName(String schoolName) {
        this.schoolName = schoolName ;
    }
    public Grade getGrade() { return grade ; }
    public void upGrade() { grade.upGrade() ; }
    @Override
    public String toString() {
        return String.format("%s, %d, %s, %s, %s",
            getName(), getAge(), getAddress(), schoolName, grade) ;
    }
    // override hashCode() and equals()
}
```



class Grade

```
public class Grade {  
    public static final int FRESH = 1 ;  
    public static final int SOPHOMORE = 2 ;  
    public static final int JUNIOR = 3 ;  
    public static final int SENIOR = 4 ;  
  
    private int grade ;  
    public Grade() { this.grade = FRESH ; }  
    public Grade(int grade) { this.grade = grade ; }  
  
    public int getGrade() { return grade ; }  
    public void upGrade() { if ( grade != SENIOR ) grade ++ ; }  
    @Override  
    public String toString() { return String.valueOf(grade); }  
    // override hashCode() and equals()  
}
```

Inheritance

- ❖ Clients of a class can access the **inherited public members** in addition to its public members.

```
public class StudentTest {  
    public static void main(String[] args) {  
        Student s1 = new Student("Ford", 19, "Kimhae", "PNU") ;  
        System.out.println(s1) ;  
        s1.setSchoolName("한국대학교") ;  
        s1.upGrade() ;  
        System.out.println(s1) ;  
  
        Student s2 = new Student("Porter", 20, "Ulsan", "PNU") ;  
        System.out.println(s2) ;  
        s2.rename("Harrison") ;  
        s2.increaseAge() ;  
        System.out.println(s2) ;  
    }  
}
```

```
Ford, 19, Kimhae, PNU, 1  
Ford, 19, Kimhae, 한국대학교, 2  
Porter, 20, Ulsan, PNU, 1  
Harrison, 21, Ulsan, PNU, 1
```

Access to the inherited members

- ❖ A subclass itself can access the **inherited public members**

```
class Person {  
    private String name ;  
    private int age ;  
    private String address ;  
  
    public String getName() { return name ; }  
    public int getAge() { return age ; }  
    public String getAddress() { return address ; }  
}
```

```
class Student extends Person {  
    private String schoolName ;  
    private Grade grade = new Grade();  
    ...  
    public String toString() {  
        return String.format("%s, %d, %s, %s, %s",  
            getName(), getAge(), getAddress(), schoolName, grade) ;  
        // name, age, and address can not be accessed because they are private  
    }  
}
```

protected member

- ❖ **Protected members** can be directly accessed from subclass.

```
class Person {  
    protected String name ;  
    protected int age ;  
    protected String address ;  
    ...  
}
```

```
class Student extends Person {  
    private String schoolName ;  
    private Grade grade = new Grade();  
  
    ...  
  
    public String toString() {  
        return String.format("%s, %d, %s, %s, %s",  
            name, age, address, schoolName, grade) ;  
        // name, age, address can be accessed  
        // now that they are protected!  
    }  
}
```

- ❖ **The use of protected members should be limited** because they can cause poor maintainability.

Access to the inherited members

- ❖ You can use the keyword "**super**" to indicate the members of the superclass

```
class Person {  
    protected String name ;  
    protected int age ;  
    protected String address ;  
    ...  
}
```

However, we should consider different names for different concepts

```
class Student extends Person {  
    private String name ;  
    private Grade grade = new Grade();  
  
    ...  
  
    public String toString() {  
        return String.format("%s, %d, %s, %s, %s",  
            super.name, age, address, name, grade) ;  
    }  
}
```

Add new members to subclasses

- ❖ A subclass can add new members for its own purpose.

```
class Student extends Person {  
    private String schoolName ;  
    private Grade grade = new Grade();  
  
    public Student(String name, int age, String address, String schoolName) {  
        super(name, age, address) ;  
        this.schoolName = schoolName ;  
    }  
    public String getSchoolName() { return schoolName ; }  
    public void setSchoolName(String schoolName) {  
        this.schoolName = schoolName ;  
    }  
    public Grade getGrade() { return grade ; }  
    public void upGrade() { grade.upGrade() ; }  
    @Override  
    public String toString() {  
        return String.format("%s, %d, %s, %s, %s",  
            getName(), getAge(), getAddress(), schoolName, grade) ;  
    }  
    // override hashCode() and equals()  
}
```



Inheritance and Overloading

- ❖ Overloading is applied for inherited member functions.

```
class Person {  
    ...  
    private int age ;  
    ...  
  
    public int getAge() { return age ; }  
    protected void setAge(int age) {  
        this.age = age ;  
    }  
    public void increaseAge() { age ++ ; }  
}
```

```
class Student extends Person {  
  
    ...  
    public void increaseAge(int delta) {  
        setAge(getAge()+delta) ;  
    }  
    ...  
}
```

```
Student s1 = new Student("Ford", 19,  
                        "Kimhae", "PNU") ;
```

```
s1.increaseAge() ;  
s1.increaseAge(2) ;
```



Constructors of subclasses

- ❖ A constructor of a subclass can initialize for members of a superclass through the constructor of its superclass.

```
class Person {  
    private String name ;  
    private int age ;  
    private String address ;  
  
    public Person(String name, int age, String address) {  
        this.name = name ; this.age = age ; this.address = address ;  
    }  
    ...  
}
```

```
class Student extends Person {  
    private String schoolName ;  
    private Grade grade = new Grade();  
  
    public Student(String name, int age, String address, String schoolName) {  
        super(name, age, address) ;  
        this.schoolName = schoolName ;  
    }  
    ...  
}
```

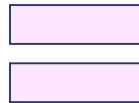
super(..) should be the **first** statement



class Object

- ❖ Object is the implicit superclass of every class when the class does not specify its superclass.

```
class Person {  
    private String name ;  
    private int age ;  
    private String address ;  
    ...  
}
```



```
class Person extends Object {  
    private String name ;  
    private int age ;  
    private String address ;  
    ...  
}
```



class Object

Method Summary	
protected Object	clone() Creates and returns a copy of this object.
boolean	equals() (Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<? extends Object>	getClass() Returns the runtime class of an object.
int	hashCode() Returns a hash code value for the object.
String	toString() Returns a string representation of the object.



Polymorphism

- ❖ Superclass variable can point to any descent objects.

...

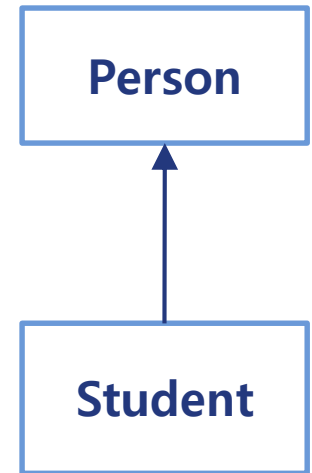
```
Person brown = new Person("Brown", 19, "Busan") ;  
Student ford = new Student("Ford", 19, "Kimhae", "PNU") ;
```

```
Person p ; // r can point to Person and Person's descents.
```

```
p = brown ; // OK  
p = ford ; // OK
```

```
Student s ;  
s = brown ; // Type mismatch: cannot convert from Person to Student
```

```
s = ford ; // OK  
s = (Student) brown ; // OK, but not recommended !
```



Polymorphism

```
public class PolymorphismTest {  
    public static void main(String[] args) {  
        Person p1 = new Person("Brown", 19, "Busan") ;  
        Person p2 = new Person("James", 20, "Masan") ;
```

```
Brown, 19, Busan  
James, 20, Masan  
Ford, 19, Kimhae  
Porter, 20, Ulsan
```

```
        Student s1 = new Student("Ford", 19, "Kimhae", "PNU") ;  
        Student s2 = new Student("Porter", 20, "Ulsan", "PNU") ;
```

```
        Person[] list = {p1, p2, s1, s2} ;
```

```
        for ( Person p: list ) {  
            System.out.printf("%s, %d\n", p.getName(), p.getAge()) ;  
            // Note: p.getSchoolName() not allowed  
            // because p is a type of Person, not Student  
            // The method getSchoolName() is undefined for the type Person  
        }  
    }  
}
```



Polymorphism

❖ **instanceof** operator can be used to determine the class of an object

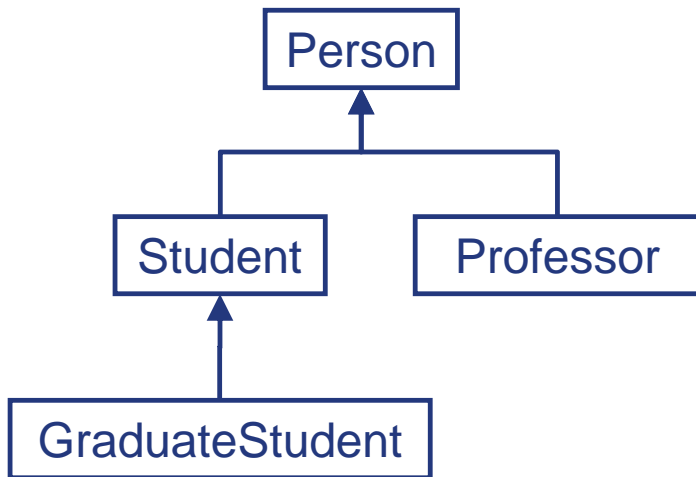
```
public class InstanceOfTest {  
    public static void main(String[] args) {  
        Person p1 = new Person("Brown", 19, "Busan") ;  
        Person p2 = new Person("James", 20, "Masan") ;  
        Student s1 = new Student("Ford", 19, "Kimhae", "PNU") ;  
        Student s2 = new Student("Porter", 20, "Ulsan", "PNU") ;  
  
        Person[] list = {p1, p2, s1, s2} ;  
  
        for ( Person p: list ) {  
            System.out.printf("%s, %d, %s%n", p.getName(), p.getAge(), p.getAddress()) ;  
            if ( p instanceof Student ) {  
                Student s = (Student) p ;  
                System.out.printf(" School: %s, %s%n", s.getSchoolName(), s.getGrade()) ;  
            }  
        }  
    }  
}
```

Brown, 19, Busan
James, 20, Masan
Ford, 19, Kimhae
School: PNU, 1
Porter, 20, Ulsan
School: PNU, 1



Inheritance Hierarchy

- ❖ A set of related classes comprises an hierarchy.



```
class Food { ... }
class Person {
    ...
    public void sleep() { ... }
    public void eat(Food food) { ... }
}
class Course { ... }
class Student extends Person {
    ...
    public void transferTo(School school) {...}
    public void takeCourse(Course course) { ... }
    public void takeExam(Course course) { ... }
}
class GraduateStudent extends Student {
    ...
    public void writeThesis() { ... }
    public void participateIn(Project project) { ... }
    public void assignAdvisor(Professor professor) { ... }
}
class Professor extends Person {
    ...
    public void teach(Course course) { ... }
    public void lead(Project project) { ... }
}
```



Abstract method

- ❖ A method is abstract if you cannot define its implementation

We know that shapes can be drawn and erased. However, we cannot implement them.

```
public class Shape {  
    private int lineColor ;  
  
    public int getLineColor() { return lineColor ; }  
    public void setLineColor(int color) { lineColor = color ; }  
  
    public abstract void draw() ;  
    public abstract void erase() ;  
    public abstract void copyToClipboard() ;  
    public abstract void pasteFromClipboard() ;  
}
```



Abstract class

- ❖ An abstract class cannot be used to create objects

```
public abstract class AbstractClass {  
    ...  
}
```

```
public class ConcreteClass extends AbstractClass {  
    ...  
}
```

```
...  
AbstractClass a1 = new AbstractClass() ; // Cannot instantiate the type Shape  
ConcreteClass c = new ConcreteClass() ; // OK  
  
AbstractClass a2 = new ConcreteClass() ; // OK
```

Abstract class

- ❖ A class should be abstract if it has abstract methods

Without abstract, Shape should be abstract to define abstract methods

```
public abstract class Shape {  
    private int lineColor ;  
  
    public int getLineColor() { return lineColor ; }  
    public void setLineColor(int color) { lineColor = color ; }  
  
    public abstract void draw() ;  
    public abstract void erase() ;  
    public abstract void copyToClipboard() ;  
    public abstract void pasteFromClipboard() ;  
}
```



Abstract class

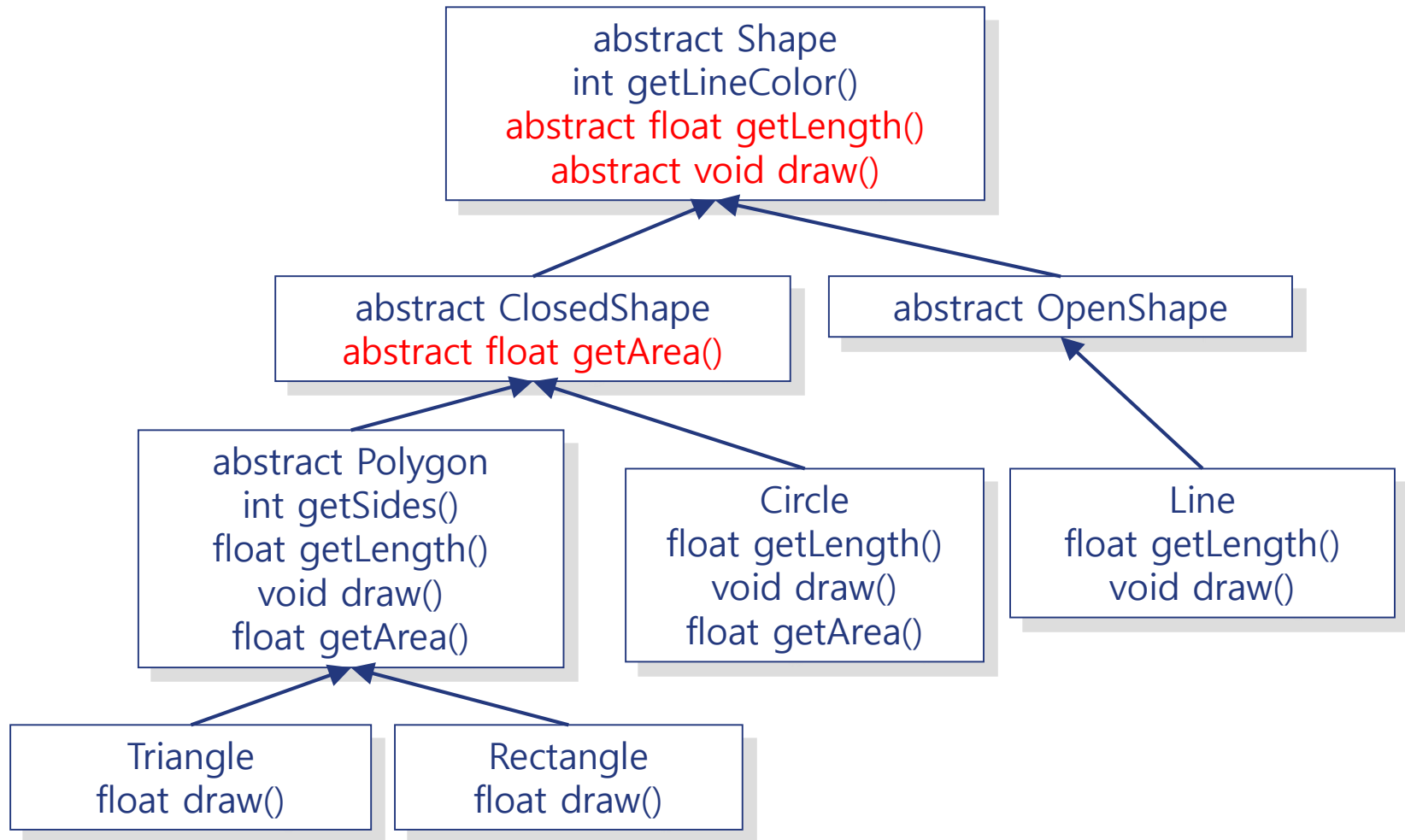
- ❖ A class with no abstract methods can be abstract

```
public abstract class Person {  
    private String name ;  
    private int age ;  
    private String address ;  
  
    public String getName() { return name ; }  
    public int getAge() { return age ; }  
    public String getAddress() { return address ; }  
}
```

```
Person p = new Person() ;  
// cannot instantiate Person, because Person is abstract
```

- ❖ This kind of classes are only used to define subclasses, not create objects.

Inheritance Hierarchy for Shape: Revised with Abstract Class



Shape, ClosedShape, OpenShape

```
public abstract class Shape {  
    private int lineColor ;  
  
    public int getLineColor() { return lineColor ; }  
    public void setLineColor(int color) { lineColor = color ; }  
  
    public abstract void draw() ;  
    public abstract float getLength();  
}
```

```
public abstract class ClosedShape extends Shape {  
    public abstract float getArea() ;  
}
```

```
public abstract class OpenShape extends Shape {  
}
```


Polygon

```
public abstract class Polygon extends ClosedShape {  
    private List<Point> points = new ArrayList<>() ;  
  
    protected void addPoint(float x, float y) {  
        Point p = new Point(x, y) ;  
        points.add(p) ;  
    }  
    @Override  
    public float getLength() { float length = 0 ; return length ; }  
    @Override  
    public float getArea() { float area = 0 ; return area ; }  
    @Override  
    public void draw() {  
        for ( final Point p : points) System.out.print(p) ;  
        System.out.println() ;  
    }  
}
```



Triangle, Rectangle

```
public class Triangle extends Polygon {  
    public Triangle(float x1, float y1, float x2, float y2, float x3, float y3) {  
        addPoint(x1, y1) ; addPoint(x2, y2) ; addPoint(x3, y3) ;  
    }  
    @Override  
    public void draw() {  
        System.out.println("Triangle") ;  
        super.draw();  
    }  
}
```

```
public class Rectangle extends Polygon {  
    public Rectangle(float x1, float y1, float x2, float y2,  
        float x3, float y3, float x4, float y4) {  
        addPoint(x1, y1) ; addPoint(x2, y2) ; addPoint(x3, y3) ; addPoint(x4, y4) ;  
    }  
    @Override  
    public void draw() {  
        System.out.println("Rectangle") ;  
        super.draw();  
    }  
}
```



Circle

```
public class Circle extends ClosedShape {  
    private Point center ;  
    private float radius ;  
  
    public Circle(float x, float y, float radius) {  
        center = new Point(x, y) ;  
        this.radius = radius ;  
    }  
    @Override  
    public float getLength() { return (float) (2 * Math.PI * radius) ; }  
    @Override  
    public float getArea() { return (float) (Math.PI * radius * radius) ; }  
    @Override  
    public void draw() {  
        System.out.println("Circle") ;  
        System.out.printf("Center: %s, Radius: %6.2f%n", center, radius) ;  
    }  
}
```



Line

```
public class Line extends OpenShape {  
    private Point start, end ;  
  
    public Line(float x1, float y1, float x2, float y2) {  
        start = new Point(x1, y1) ; end = new Point(x2, y2) ;  
    }  
    @Override  
    public float getLength() {  
        return (float) (Math.sqrt( (end.getX() - start.getX()) *  
            (end.getX() - start.getX()) + (end.getY() - start.getY()) *  
            (end.getY() - start.getY()) ) ) ;  
    }  
    @Override  
    public void draw() {  
        System.out.println("Line") ;  
        System.out.printf("\tStart: %s, End: %s%n", start, end) ;  
    }  
}
```



ShapeHierarchyTest

```
public class ShapeHierarchy {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0F, 5.5F, 10F) ;  
        c1.draw() ;  
        Triangle t1 = new Triangle(0F, 0F, 10F, 0F, 20F, 20F) ;  
        t1.draw() ;  
        Rectangle r1 = new Rectangle(0F, 0F, 10F, 0F, 10F, 10F, 0F, 10F) ;  
        r1.draw() ;  
        Line l1 = new Line(10F, 10F, 15F, 10F) ;  
        l1.draw() ;  
    }  
}
```

Circle

Center: [0.00, 5.50], Radius: 10.00

Triangle

[0.00, 0.00][10.00, 0.00][20.00, 20.00]

Rectangle

[0.00, 0.00][10.00, 0.00][10.00, 10.00][0.00, 10.00]

Line

Start: [10.00, 10.00], End: [15.00, 10.00]



Generic Shape List

❖ ArrayList of Shape can support any descents of Shape

```
public class ShapeListTest {  
    public static void main(String[] args) {  
        List<Shape> shapes = new ArrayList<>() ;  
  
        Circle c1 = new Circle(0F, 5.5F, 10F) ;  
        Triangle t1 = new Triangle(0F, 0F, 10F, 0F, 20F, 20F) ;  
        Rectangle r1 = new Rectangle(0F, 0F, 10F, 0F, 10F, 10F, 0F, 10F) ;  
  
        shapes.add(c1) ;  
        shapes.add(t1) ;  
        shapes.add(r1) ;  
  
        for ( Shape s : shapes ) s.draw() ; // polymorphic invocation of draw()  
    }  
}
```



final class

- ❖ Classes declared with **final** cannot be extended!

```
class Person {  
    ...  
}  
class Student extends Person {    // OK  
    ...  
}  
class Undergraduate extends Student {    // OK  
    ...  
}  
final class Teacher extends Person {    // OK  
    ...  
}  
  
class HomeroomTeacher extends Teacher {  
    // ERROR; final class Teacher cannot be extended!  
    ...  
}
```



final method

❖ Methods declared with **final** cannot be overridden!

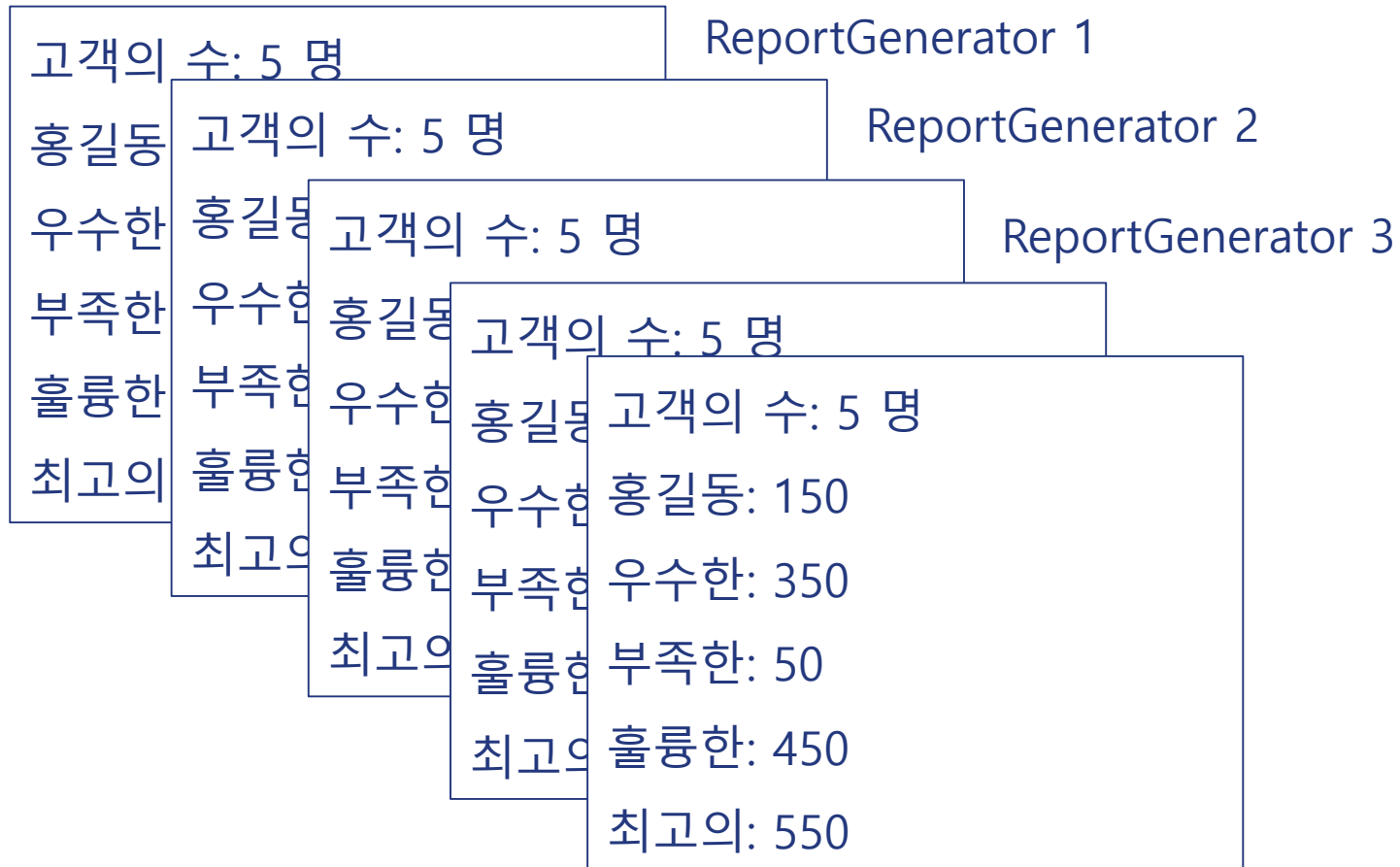
```
class Person {  
    public toString() { ... }  
    public final String getName() { ... }  
}  
  
class Student extends Person {  
    public String toString() { ... } // OK  
    public String getName() {...} // ERROR; Cannot override the final method!  
}
```



REPORT GENERATOR

Report Generators

- ❖ Classes for each report format have common codes



Customer

```
public class Customer {  
    private final String name ;  
    private int point;  
  
    public Customer(String name, int point) {  
        this.name = name;  
        this.point = point;  
    }  
    public final int getPoint() {  
        return point;  
    }  
    public void setPoint(int point) {  
        this.point = point;  
    }  
    public final String getName() {  
        return name;  
    }  
}
```



AbstractReportGenerator

```
public abstract class AbstractReportGenerator {  
    public final String generate(List<Customer> customers) {  
        List<Customer> selectedCustomers = select(customers) ;  
        String report = getReportHeader(selectedCustomers) ;  
        for ( final Customer customer : selectedCustomers )  
            report += getReportForCustomer(customer) ;  
        report += getReportFooter(selectedCustomers) ;  
        return report ;  
    }  
    protected List<Customer> select(List<Customer> customers) {  
        List<Customer> selected = new ArrayList<Customer>() ;  
        for ( final Customer customer: customers )  
            if ( customerReportCondition(customer) ) selected.add(customer) ;  
        return selected;  
    }  
    protected abstract boolean customerReportCondition(Customer customer) ;  
    protected abstract String getReportHeader(List<Customer> customers) ;  
    protected abstract String getReportForCustomer(Customer customer) ;  
    protected abstract String getReportFooter(List<Customer> customers) ;  
}
```



SimpleReportGenerator

```
public class SimpleReportGenerator extends AbstractReportGenerator {
```

```
}
```

고객의 수: 5 명
홍길동: 150
우수한: 350
부족한: 50
훌륭한: 450
최고의: 550



ComplexReportGenerator

```
public class ComplexReportGenerator extends AbstractReportGenerator {
```

```
}
```

고객의 수: 4 명입니다
150: 홍길동
350: 우수한
450: 훌륭한
550: 최고의
점수 합계: 1500



ReportGeneratorTest

```
public class ReportGeneratorTest {  
    public static void main(String[] args) {  
        List<Customer> customers = new ArrayList<Customer>() ;  
        customers.add(new Customer("홍길동", 150)) ;  
        customers.add(new Customer("우수한", 350)) ;  
        customers.add(new Customer("부족한", 50)) ;  
        customers.add(new Customer("훌륭한", 450)) ;  
        customers.add(new Customer("최고의", 550)) ;  
  
        AbstractReportGenerator simpleGenerator =  
            new SimpleReportGenerator() ;  
        System.out.println(simpleGenerator.generate(customers));  
  
        AbstractReportGenerator complexGenerator =  
            new ComplexReportGenerator() ;  
        System.out.println(complexGenerator.generate(customers));  
    }  
}
```

고객의 수: 5 명
홍길동: 150
우수한: 350
부족한: 50
훌륭한: 450
최고의: 550

고객의 수: 4 명입니다
150: 홍길동
350: 우수한
450: 훌륭한
550: 최고의
점수 합계: 1500



Q&A
