ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 20, 2014

Abstract

- Inheritance
  - Introduction
  - Generalization/specialization

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

In particular, we have seen that OOP provides mechanisms to control the visibility of the methods and the variables.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

In particular, we have seen that OOP provides mechanisms to control the visibility of the methods and the variables.

The methods and variables that are public define the *interface* of the object.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

In particular, we have seen that OOP provides mechanisms to control the visibility of the methods and the variables.

The methods and variables that are public define the *interface* of the object.

Having the interface clearly defined allows the implementers and the users of the class to work independently; the creator can change the implementation of the class, as long as it does not affect the interface, and the programs developed by the users will continue to work.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

In particular, we have seen that OOP provides mechanisms to control the visibility of the methods and the variables.

The methods and variables that are public define the *interface* of the object.

Having the interface clearly defined allows the implementers and the users of the class to work independently; the creator can change the implementation of the class, as long as it does not affect the interface, and the programs developed by the users will continue to work.

As a general principle, in CS II, all the instance variables should be declared private.

Summary

We have seen that object-oriented programming (OOP) helps organizing and maintaining large software systems.

The data, and the methods that act upon the data, are *encapsulated* into a single entity called the object.

The instance variables define the properties or state of an object.

In particular, we have seen that OOP provides mechanisms to control the visibility of the methods and the variables.

The methods and variables that are public define the *interface* of the object.

Having the interface clearly defined allows the implementers and the users of the class to work independently; the creator can change the implementation of the class, as long as it does not affect the interface, and the programs developed by the users will continue to work.

As a general principle, in CS II, all the instance variables should be declared private.

If the value of a variable needs to be accessed (read or mutated) from outside the class, then the interface of the object will include setter and getter methods. This principle will allow us to maintain the integrity of the objects.

If the value of a variable needs to be accessed (read or mutated) from outside the class, then the interface of the object will include setter and getter methods. This principle will allow us to maintain the integrity of the objects.

The class specifies the content of the objects but it also exists during the execution of a program. Each object knows the class from which it was instantiated from (is an instance of). No matter how many instances there are, 0, 1 or n, there is only one copy of the class.

If the value of a variable needs to be accessed (read or mutated) from outside the class, then the interface of the object will include setter and getter methods. This principle will allow us to maintain the integrity of the objects.

The class specifies the content of the objects but it also exists during the execution of a program. Each object knows the class from which it was instantiated from (is an instance of). No matter how many instances there are, 0, 1 or n, there is only one copy of the class.

Class variables and methods are shared by all instances of a class.

If the value of a variable needs to be accessed (read or mutated) from outside the class, then the interface of the object will include setter and getter methods. This principle will allow us to maintain the integrity of the objects.

The class specifies the content of the objects but it also exists during the execution of a program. Each object knows the class from which it was instantiated from (is an instance of). No matter how many instances there are, 0, 1 or n, there is only one copy of the class.

Class variables and methods are shared by all instances of a class.

$\Rightarrow$ In today's lecture, we look at other important features of object-oriented programming that help organizing and maintaining large software systems: *inheritance* and *polymorphism*.

Inheritance

OO languages, in general, also offer other tools to structure large systems. **Inheritance** is one of them.

Inheritance

OO languages, in general, also offer other tools to structure large systems. **Inheritance** is one of them.

Inheritance allows to organize the classes hierarchically.

Inheritance

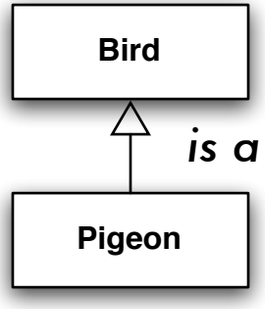
OO languages, in general, also offer other tools to structure large systems. **Inheritance** is one of them.

Inheritance allows to organize the classes hierarchically.

Inheritance favors **code reuse**!

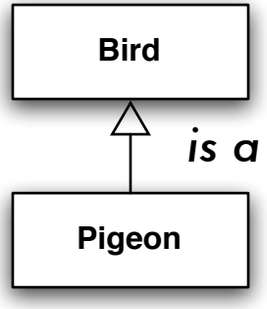
Inheritance

The class immediately above is called the **superclass** or **parent class** while the class immediately below is called the **subclass**, **child class** or **derived class**.



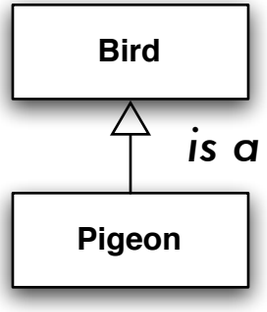
Inheritance

The class immediately above is called the **superclass** or **parent class** while the class immediately below is called the **subclass**, **child class** or **derived class**.



In this example, **Bird** is the superclass of **Pigeon**, i.e. **Pigeon** "is a" subclass of **Bird**.

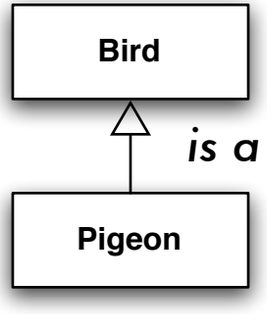
Inheritance



In Java, the "is a" relationship is expressed using the reserved keyword **extends**, as follows:

```
public class Pigeon extends Bird {
    ...
}
```

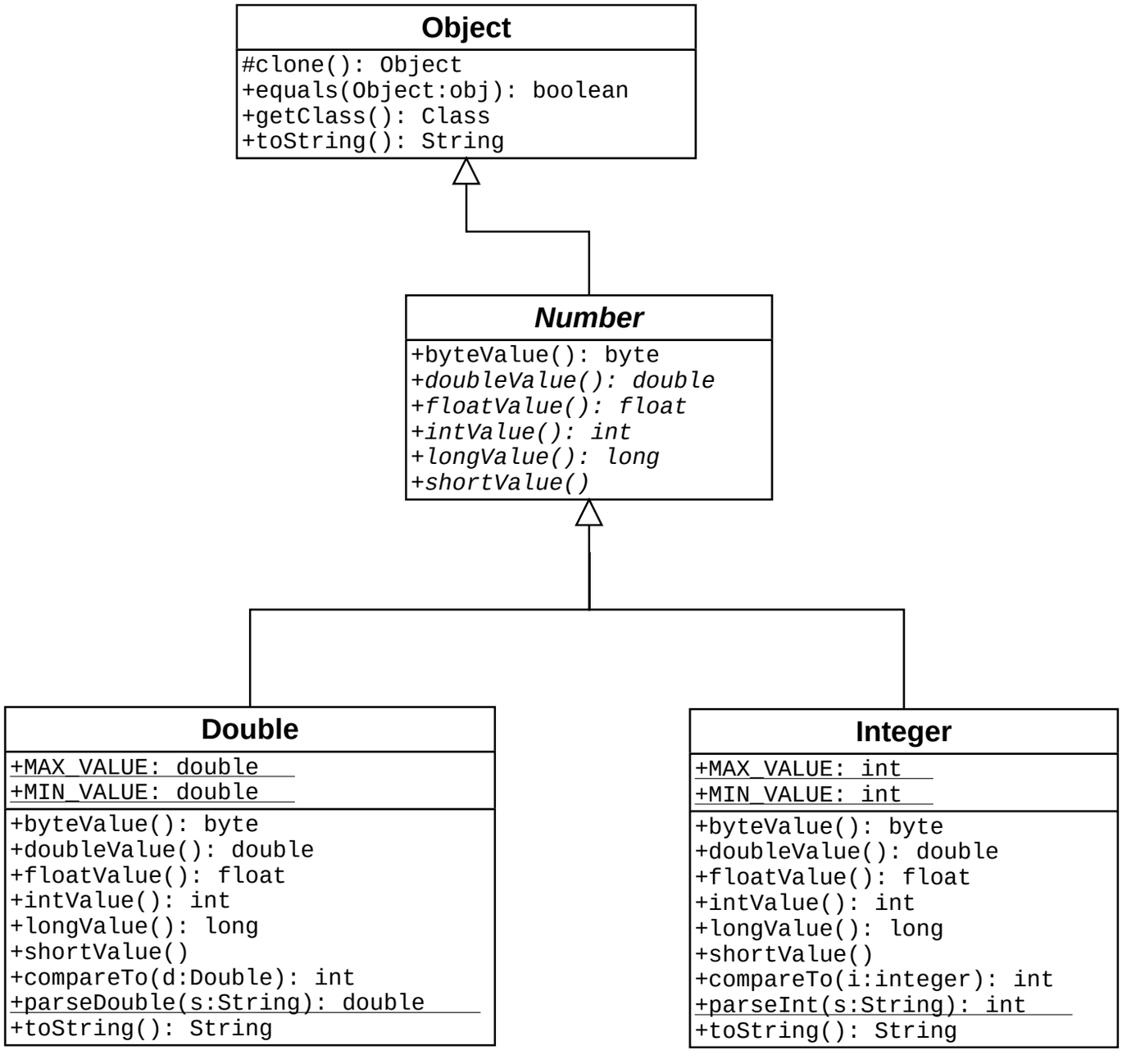
Inheritance



In UML, the "is a" relationship is expressed using a continuous line connecting the child to its parent, and an open triangle pointing towards the parent.

Inheritance

In Java, the classes are organized into a single hierarchy, with the most general class, called **Object**, being at the **top** (or **root**) of the tree.


Object
#clone(): Object
+equals(Object:obj): boolean
+getClass(): Class
+toString(): String
Number
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
Double
+MAX_VALUE: double
+MIN_VALUE: double
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(d:Double): int
+parseDouble(s:String): double
+toString(): String
Integer
+MAX_VALUE: int
+MIN_VALUE: int
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(i:integer): int
+parseInt(s:String): int
+toString(): String

Inheritance

If the **superclass** is not explicitly mentioned, **Object** is the immediate parent class, the following two declarations are therefore identical

```
public class C {
    ...
}
```

and

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

Inheritance

In Java, all the classes have exactly one parent; except **Object** that has no parent.

Inheritance

In Java, all the classes have exactly one parent; except **Object** that has no parent.

We talk about **single inheritance** as opposed to multiple inheritance.

What does it mean?

A class inherits all the characteristics (variables and methods) of its superclass(es).

What does it mean?

A class inherits all the characteristics (variables and methods) of its superclass(es).

1. a subclass inherits all the methods and variables of its superclass(es);

What does it mean?

A class inherits all the characteristics (variables and methods) of its superclass(es).

1. a subclass inherits all the methods and variables of its superclass(es);

2. a subclass can introduce/add new methods and variables;

What does it mean?

A class inherits all the characteristics (variables and methods) of its superclass(es).

1. a subclass inherits all the methods and variables of its superclass(es);
2. a subclass can introduce/add new methods and variables;
3. a subclass can override the methods of its superclass.

What does it mean?

A class inherits all the characteristics (variables and methods) of its superclass(es).

1. a subclass inherits all the methods and variables of its superclass(es);

2. a subclass can introduce/add new methods and variables;

3. a subclass can override the methods of its superclass.

Because of 2 and 3, the subclass is a **specialization** of the superclass, i.e. the superclass is **more general** than its subclasses.

Inheritance

Inheritance is one of the tools that help developing reusable components (classes).

Shape

Variants of this example can be found in most textbooks about object-oriented programming.

Shape

Variants of this example can be found in most textbooks about object-oriented programming.

Problem: A software system must be developed to represent various shapes, such as circles and rectangles.

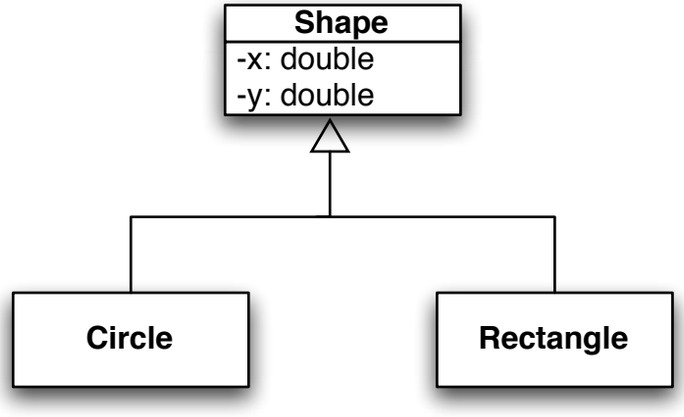
Shape

Variants of this example can be found in most textbooks about object-oriented programming.

Problem: A software system must be developed to represent various shapes, such as circles and rectangles.

All the shapes must have two instance variables, **x** and **y**, to represent the location of each object.

Shape



Shape

Furthermore, **all the shapes** should have the following methods:

```
double getX();                          // Returns the value of x

double getY();                          // Returns the value of y

void moveTo(double x, double y);        // Move the shape to a new location

double area();                          // Calculates the area of the shape

void scale(double factor);              // Scales the shape by some factor

String toString();                      // Returns a String representation
```

Shape

Furthermore, **all the shapes** should have the following methods:

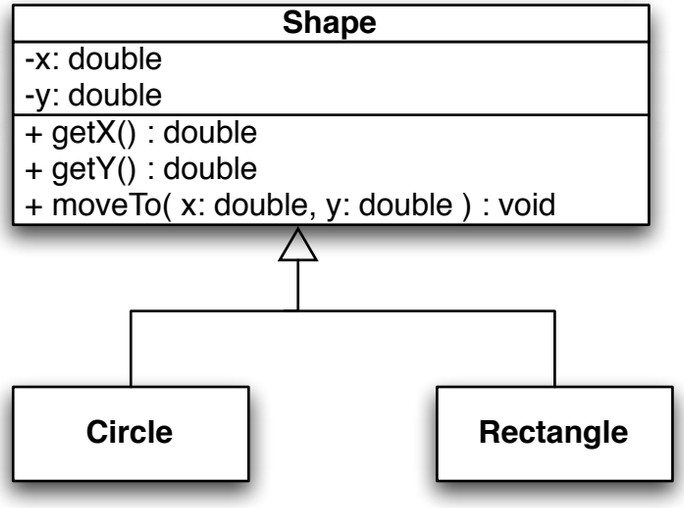
```
double getX();                          // Returns the value of x

double getY();                          // Returns the value of y

void moveTo(double x, double y);        // Move the shape to a new location

double area();                          // Calculates the area of the shape

void scale(double factor);              // Scales the shape by some factor

String toString();                      // Returns a String representation
```

Keep the specification in mind as we won't be able to implement it fully, at first.

Shape

The implementation of the first three methods would be the same for all kinds of shapes.

Shape


Shape
-x: double
-y: double
+ getX() : double
+ getY() : double
+ moveTo(x: double, y: double) : void
Circle
Rectangle

Shape

On the other hand, the calculation of the **area** and the implementation of the **scale** method would depend on the kind of shape being dealt with.

Shape

On the other hand, the calculation of the **area** and the implementation of the **scale** method would depend on the kind of shape being dealt with.

Finally, the method **toString()** requires information from both levels, general and specific, all shapes should display their location and also their specific information, such as the radius in the case of a circle.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
}
```

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Can I do this?

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Can I do this? Yes. Several methods (or constructors) with the same name can be added to a class, as long as their signature differ.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Can I do this? Yes. Several methods (or constructors) with the same name can be added to a class, as long as their signature differ. I am calling this *ad hoc* polymorphism, or overloading.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public Shape() {
        x = 0;
        y = 0;
    }
    public Shape( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

Can I do this? Yes. Several methods (or constructors) with the same name can be added to a class, as long as their signature differ. I am calling this *ad hoc* polymorphism, or overloading. Why would you want to do this?

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

}
```

Adding the getters!

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public final double getX() {
        return x;
    }
    public final double getY() {
        return y;
    }

}
```

By using the keyword **final**, we can prevent the descendants of this class overriding the method.

Shape

```
public class Shape extends Object {

    private double x;
    private double y;

    public final double getX() { return x; }
    public final double getY() { return y; }

    public final void moveTo( double x, double y ) {
        this.x = x;
        this.y = y;
    }
}
```

The method **moveTo** can be seen as a setter!

Circle

```
public class Circle extends Shape {


}
```

The above declaration defines a class **Circle** that extends Shape, which means that an instance of the class **Circle** possesses two instance variables **x** and **y**, as well as the following methods: **getX()**, **getY()** and **moveTo(double x, double)**.

Circle

```
public class Circle extends Shape {

    // Instance variable
    private double radius;

}
```

The instance variables **x** and **y** and inherited (common to all **Shapes**). The variable **radius** is specific to a **Circle**.

Private vs protected

With the current definition of the class **Shape**, it would not have been possible to define the constructor of the class **Circle** as follows:

```
public Circle( double x, double y, double radius ) {
    this.x = x;
    this.y = y;
    this.radius = radius;
}
```

The compiler would complain saying "x has private access in Shape" (and similarly for **y**).

Private vs protected

With the current definition of the class **Shape**, it would not have been possible to define the constructor of the class **Circle** as follows:

```
public Circle( double x, double y, double radius ) {
    this.x = x;
    this.y = y;
    this.radius = radius;
}
```

The compiler would complain saying "x has private access in Shape" (and similarly for **y**).

This is because an attribute declared private in the parent class cannot be accessed within the child class.

Private vs protected

With the current definition of the class **Shape**, it would not have been possible to define the constructor of the class **Circle** as follows:

```
public Circle( double x, double y, double radius ) {
    this.x = x;
    this.y = y;
    this.radius = radius;
}
```

The compiler would complain saying "x has private access in Shape" (and similarly for **y**).

This is because an attribute declared private in the parent class cannot be accessed within the child class.

Private vs protected

To circumvent this and implement the constructor as above, the definition of **Shape** should be modified so that **x** and **y** would be declared **protected**:

```
public class Shape extends Object {

    protected double x;
    protected double y;

    ...
}
```

Private vs protected

To circumvent this and implement the constructor as above, the definition of **Shape** should be modified so that **x** and **y** would be declared **protected**:

```
public class Shape extends Object {

    protected double x;
    protected double y;

    ...
}
```

Private vs protected

When possible, it is preferable to maintain the visibility **private**.

Private vs protected

When possible, it is preferable to maintain the visibility **private**.

Private instance variables and **final** instance methods go hand in hand.

Private vs protected

When possible, it is preferable to maintain the visibility **private**.

Private instance variables and **final** instance methods go hand in hand.

The declaration of an instance variable **private** prevents the subclasses from accessing the variable.

Private vs protected

When possible, it is preferable to maintain the visibility **private**.

Private instance variables and **final** instance methods go hand in hand.

The declaration of an instance variable **private** prevents the subclasses from accessing the variable.

The declaration of a method **final** prevents subclasses from overriding the method.

Private vs protected

When possible, it is preferable to maintain the visibility **private**.

Private instance variables and **final** instance methods go hand in hand.

The declaration of an instance variable **private** prevents the subclasses from accessing the variable.

The declaration of a method **final** prevents subclasses from overriding the method.

Private vs protected

By declaring the instance variables **private** and the access/mutator instance methods **final** you ensure that all the modifications to the instance variables are "concentrated" in the class where they were first declared.

Circle

```
public class Circle extends Shape {

    private double radius;

    // Constructors

    public Circle() {
        super();
        radius = 0;
    }

    public Circle( double x, double y, double radius ) {
        super( x, y );
        this.radius = radius;
    }

}
```

super()

The statement **super(. . .)** is an explicit call to the constructor of the immediate superclass.

super()

The statement **super(. . .)** is an explicit call to the constructor of the immediate superclass.

- This particular construction can only appear in a constructor;

super()

The statement **super(. . .)** is an explicit call to the constructor of the immediate superclass.

- This particular construction can only appear in a constructor;
- Can only be the first statement of the constructor;

super()

The statement **super(. . .)** is an explicit call to the constructor of the immediate superclass.

- This particular construction can only appear in a constructor;
- Can only be the first statement of the constructor;
- The super() will be automatically inserted for you unless you insert a super(...) yourself!?

super()

- The super() will be automatically inserted for you unless you insert a super(...) yourself!?

super()

- The super() will be automatically inserted for you unless you insert a super(...) yourself!?

If the first statement of a constructor is not an explicit call super(. . .), Java inserts a call super(), which means that the superclass has to have a constructor of arity 0, or else a compile time error will occur.

super()

- The super() will be automatically inserted for you unless you insert a super(...) yourself!?

If the first statement of a constructor is not an explicit call super(. . .), Java inserts a call super(), which means that the superclass has to have a constructor of arity 0, or else a compile time error will occur. Remember, the default constructor, the one with arity 0, is no longer present if a constructor has been defined.

super()

"If a constructor body does not begin with an explicit constructor invocation (. . .), then the constructor body is implicitly assumed by the compiler to begin with a superclass constructor invocation "super();", an invocation of the constructor of its direct superclass that takes no arguments."

$\Rightarrow$ Gosling et al. (2000) *The Java Language Specification.*

Circle

```
public class Circle extends Shape {

    private double radius;

    // Access method

    public double getRadius() {
        return radius;
    }

}
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    public Rectangle() {
        super();
        width = 0;
        height = 0;
    }

    public Rectangle( double x, double y, double width, double height )
        super(x, y);
        this.width = width;
        this.height = height;
    }
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }
}
```

Rectangle

```
public class Rectangle extends Shape {

    private double width;
    private double height;

    // ...

    public void flip() {
        double tmp = width;
        width = height;
        height = tmp;
    }

}
```

```
Circle d = new Circle( 100, 200, 10 );
System.out.println( d.getRadius() );

Circle c = new Circle();
System.out.println( c.getX() );

d.scale( 2 );
System.out.println ( d );

Rectangle r = new Rectangle();
System.out.println( r.getWith() );

Rectangle s = new Rectangle( 50, 50, 10, 15 );
System.out.println( s.getY() );

s.flip();
System.out.println( s.getY() );
```

Summary

Inheritance allows to reuse code. The methods **getX()**, **getY()** and **moveTo()** were only defined in the class **Shape**.

Fixing a bug or making an improvement in the superclass will fix or improve all the subclasses.