ITI 1121. Introduction to Computing II

Inheritance: polymorphism

by

Marcel Turcotte

Version February 8, 2020

Preamble

Preamble

Overview

Overview

Inheritance: polymorphism

The concept of inheritance in Java promotes code reuse and supports the notion of polymorphism.

General objective:

- This week you will be able to create polymorphic methods.

Preamble

Learning objectives

Learning objectives

- **Describe** the concept of polymorphism.
- **Create** polymorphic methods.
- **Compare** the interface and the abstract class.

Lectures:

- Pages 7–31, 39–45 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

1 Preamble

2 Polymorphism

3 Inheritance and Java

4 Prologue

Polymorphism

Polymorphism

- From the Greek ***polus*** = several and ***morphê*** = forms, so it means **which has several forms**.

Definitions

In computer science, **polymorphism** consists in allowing the use of an identifier for different entities (see different types).

1. **Polymorphism *ad hoc* (name overloading)**: the same method name is associated with different blocks of code. These methods have the same name, but they differ by their list of parameters.
2. **Subtype polymorphism (by inheritance)**: an identifier is linked to data of different types by a subtype relationship.
3. **Parametric polymorphism (generic)**: the class has one or more formal type parameters.

Overloading

- The **PrintStream** class uses *ad hoc* polymorphism to implement the **println** method.

```
println()
println(boolean value)
println(char value)
println(char[] value)
println(double value)
println(float value)
println(int value)
println(long value)
```

Name overloading (continued)

- Three methods having different **signatures** *.

```
public static int sum(int a, int b, int c) {
    return a + b + c;
}
public static int sum(int a, int b) {
    return a + b;
}
public static double sum(double a, double b) {
    return a + b;
}
```

*In Java, the signature of a method includes the method name and the parameter list, but not the return value.

Polymorphism by subtype

Problem : implement a method **isLeftOf** which returns **true** if **this** shape is located to the left of its argument (another geometric shape) and **false** otherwise.

isLeftOf

```java
Circle c1, c2;
c1 = new Circle(10.0, 20.0, 5.0);
c2 = new Circle(20.0, 10.0, 5.0);

if (c1.isLeftOf(c2)) {
    System.out.println("c1 isLeftOf c2");
} else {
    System.out.println("c2 isLeftOf c1");
}
```

isLeftOf

```java
Rectangle r1, r2;
r1 = new Rectangle(0.0, 0.0, 1.0, 1.0);
r2 = new Rectangle(100.0, 100.0, 200.0, 400.0);

if (r1.isLeftOf(r2)) {
    System.out.println("r1 isLeftOf r2");
} else {
    System.out.println("r2 isLeftOf r1");
}
```

isLeftOf

```
if (r1.isLeftOf(c1)) {
    System.out.println("r1 isLeftOf c1");
} else {
    System.out.println("c1 isLeftOf r1");
}

if (c2.isLeftOf(r2)) {
    System.out.println("c2 isLeftOf r2");
} else {
    System.out.println("r2 isLeftOf c2");
}
```

An outrageous solution!

```
public boolean isLeftOf(Circle c) {
    return getX() < c.getX();
}
public boolean isLeftOf(Rectangle r) {
    return getX() < r.getX();
}
```

- **Why?**

An outrageous solution!

```java
public boolean isLeftOf(Circle c) {
    return getX() < c.getX();
}
public boolean isLeftOf(Rectangle r) {
    return getX() < r.getX();
}
```

- **As many implementations** as there are varieties of shapes!
- Yet, all the implementations are **identical**!
- Whenever a new category of shape is defined (say **Triangle**), a new method **isLeftOf** must be created!

Solution

- Suggestions?

```java
public boolean isLeftOf("Any Shape" s) {
    return getX() < s.getX();
}
```

- How to write any “**Any Shape**” in Java?

Solution

- Let’s implement the **isLeftOf** method in the **Shape** class as follows.

```java
public boolean isLeftOf(Shape s) {
    return getX() < s.getX();
}
```

isLeftOf

```java
Circle c;
c = new Circle(10.0, 20.0, 5.0);

Rectangle r;
r = new Rectangle(0.0, 0.0, 1.0, 1.0);

if (c.isLeftOf(r)) {
    System.out.println("c isLeftOf r");
} else {
    System.out.println("r isLeftOf c");
}
```

isLeftOf

```
if (c.isLeftOf(r)) {
    // ...
```

- The method **isLeftOf** of the object designated by the reference **c** is called.
- Perfect, **c** designates an object of the class **Circle**, the latter inherits the method **isLeftOf**.

isLeftOf

```
if (c.isLeftOf(r)) {
    // ...
```

- Um, during the call, the value of the actual parameter, **r**, is copied to the formal parameter, **s**.
- Should we conclude that the following statements are also valid?

```
Shape s;
Rectangle r;
r = new Rectangle(0.0, 0.0, 1.0, 1.0);
s = r;
```

Types

- “A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a **primitive** type (§4.2) or a **reference type** (§4.3). A variable always contains a value that is assignment **compatible** (§5.2) with its type.”
- “Assignment of a value of compile-time reference type **S** (source) to a variable of compile-time reference type **T** (target) is checked as follows:
  - If **S** is a class type:
    - If **T** is a class type, then **S** must either be the **same class** as **T**, or **S** must be a subclass of **T**, or a compile-time error occurs.”

⇒ Gosling et al. (2000) *The Java Language Specification.*

isLeftOf

Indeed, this definition confirms that the following statements are valid.

```
Shape s;
Rectangle r;
r = new Rectangle(0.0, 0.0, 1.0, 1.0);
s = r;
```

but not “**r = s**”!

Polymorphism

A variable **s** designates an object of the class **Shape** or one of its subclasses.

```
Shape s;
```

Utilisation:

```
s = new Circle(0.0, 0.0, 1.0);
s = new Rectangle(10.0, 100.0, 10.0, 100.0);
```

Polymorphism

```
public boolean isLeftOf(Shape other) {
    boolean result;
    if (getX() < other.getX()) {
        result = true;
    } else {
        result = false;
    }
    return result;
}
```

Usage:

```
Circle c = new Circle(10.0, 10.0, 5.0);
Rectangle d = new Rectangle(0.0, 10.0, 12.0, 24.0);
if (c.isLeftOf(d)) { ... }
```

Exercises

```java
Shape s;
Circle c;
c = new Circle(0.0, 0.0, 1.0);
s = c;

if (c.getX()) { ... } // valid?
if (s.getX()) { ... } // valid?

if (c.getRadius()) { ... } // valid?
if (s.getRadius()) { ... } // valid?
```

Remarks

```
Shape s;
Circle c;
c = new Circle(0.0, 0.0, 1.0);
s = c;
```

- The object designated by **s** remains a circle (**Circle**). The class of an object remains the same throughout the execution of the program.

Remarks

```
Shape s;
Circle c;
c = new Circle(0.0, 0.0, 1.0);
s = c;

if (s.getX()) { ... }
```

- When we use **s** to designate a circle (**Circle**), the object “is seen as” a geometrical shape (**Shape**), in the sense that we only see the characteristics (methods and variables) defined in the class **Shape**.

Remarks

- Polymorphism is a powerful concept. The method **isLeftOf** that we have defined can be used not only to handle circles and rectangles, but also any object of a future subclass of the class **Shape**.

```java
public class Triangle extends Shape {
    // ...
}
```

Calculating the area

Problem : We want **all** geometric shapes (objects in the subclasses of **Shape**) to have a method for calculating the **area**.

What do you mean, Marcel?

```java
public class Shape {

  // ...

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return 0;
      } else {
          return 1;
      }
  }
}
```

What do you think?

```java
public class Shape {

  // ...

  // Must be redefined by the subclasses or else ...

  public double area() {
      return -1.0;
  }

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return 0;
      } else {
          return 1;
      }
  }
}
```

Abstract

```
public class Shape {

  // ...

  public abstract double area();

  public int compareTo(Shape other) {
      if (area() < other.area()) {
        return -1;
      } else if (area() == other.area()) {
        return 0;
      } else {
        return 1;
      }
  }
}
```

Abstract

```
public abstract class Shape {

  // ...

  public abstract double area();

  public int compareTo(Shape other) {
      if (area() < other.area()) {
          return -1;
      } else if (area() == other.area()) {
          return 0;
      } else {
          return 1;
      }
  }
}
```

Abstract classes

- A class declaring an **abstract method** must be **abstract**.
- You **can't create objects** of an abstract class.
- A class **can** be declared **abstract**, even if it **does not** contain **abstract** methods.

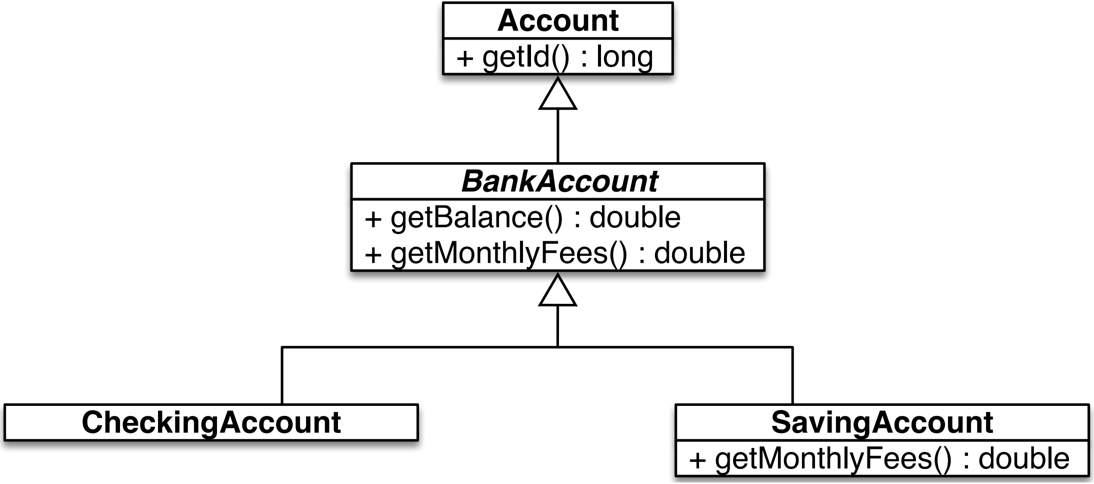
What have we achieved?

```java
public class Circle extends Shape {

}
```

```
Circle.java:1: Circle is not abstract and
does not override abstract method area() in Shape
public class Circle extends Shape {
       ^
1 error
```

```java
public class Circle extends Shape {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public void scale(double factor) {
        radius *= factor;
    }
}
```

Name lookup



- **BankAccount** and **SavingAccount** both have a method named **getMonthlyFees**.

- **BankAccount:**

```java
public double getMonthlyFees() {
    return 25.0;
}
```

- **SavingAccount:**

```java
public double getMonthlyFees() {
    double result;
    if (getBalance() > 5000.0) {
        result = 0.0;
    } else {
        result = super.getMonthlyFees();
    }
    return result;
}
```

- **Consider** the following statements:

```
Account a;

BankAccount b;

SavingAccount s;

s = new SavingAccount();
s.getMonthlyFees();

b = s;
b.getMonthlyFees();

a = b;
a.getMonthlyFees();
```

Dynamic binding

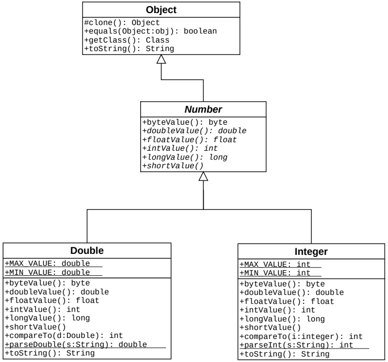
- Let **S** (*source*) be the type of the object currently designated by a reference variable of type **T** (*target*).
- Unless the method is **static** or **final**, the lookup
  1. occurs at **runtime**, and
  2. starts at the class **S**:
     - if the method is **found**, this is the method that will be **executed**,
     - otherwise the immediate **superclass** is considered,
     - this process **continues** until the first occurrence of the method is found.

⇒ A.K.A. **late binding** or **virtual binding**

Inheritance and Java

Object

- In Java, classes are organized in a tree structure. The most general class, the one at the root of the tree, is called **Object**.



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

Object

- If the superclass is not explicitly mentioned, **Object** is the default superclass, so the following statement:

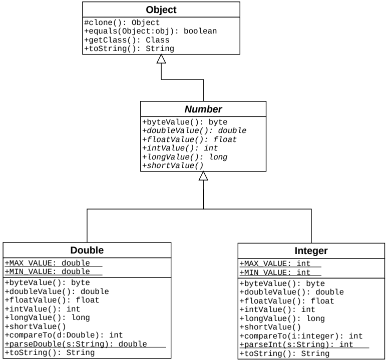
```
public class C {

}
```

is equivalent to this one:

```
public class C extends Object {

}
```

equals

- The class **Object** defines a method **equals**.
- **Every** Java object therefore has a method **equals**.
- So we can always write **a.equals(b)** if **a** and **b** are reference variables.



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

equals

- This is the **equals** method of the **Object** class.

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

Account

```java
public class Account {

    private int id;
    private String name;

    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Test

```java
public class Test {
    public static void main(String[] args) {
        Account a, b;
        a = new Account(1, new String("Marcel"));
        b = new Account(1, new String("Marcel"));
        if (a.equals(b)) {
            System.out.println("a and b are equals");
        } else {
            System.out.println("a and b are not equals");
        }
    }
}
```

- What will the **result** be?

```java
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) { // <---
            result = false;
        } ...
        return result;
    }
}
```

```java
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (this.getClass() != o.getClass()) { // <---
            result = false;
        } ...
        return result;
    }
}
```

```
public class Account {
    private int id;
    private String name;
    public Account(int id, String name) { ... }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (this.getClass() != o.getClass()) {
            result = false;
        } else {
            Account other = (Account) o; // <---
            ...
        }
        return result;
    }
}
```

```java
public class Account {
    private int id;  private String name;
    public Account(int id, String name) { ... }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (this.getClass() != o.getClass()) {
            result = false;
        } else {
            Account other = (Account) o;
            if (id != other.id) {
                result = false;
            } else if (name == null && other.name != null) {
                result = false;
            } else if (name != null && ! name.equals(other.name) ) {
                result = false;
            }
        }
        return result;
    }
}
```

Test

```java
public class Test {
    public static void main(String[] args) {
        Account a, b;
        a = new Account(1, new String("Marcel"));
        b = new Account(1, new String("Marcel"));
        if (a.equals(b)) {
            System.out.println("a and b are equals");
        } else {
            System.out.println("a and b are not equals");
        }
    }
}
```

- What will the **result** be?

toString()

- Since the class **Object** declares a method **toString()**, all objects have this method.
- Either the class inherits a method **toString()** or it redefines it.
- Thus, the statement **a.toString()** is always valid if **a** is a reference variable.

toString()

```java
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
System.out.println(a.toString());
```

System.out.println

```
public class PrintStream {

    // ...

    public void println(Object obj) {
        write(String.valueOf(obj));
    }
}
```

```
public class String {

    // ...

    public static String valueOf(Object obj) {
        return (obj == null) ? "null" : obj.toString();
    }
}
```

```
public class Account {

    private int id;
    private String name;

    public Account(int id, String name) { ... }

    // ...
}
```

toString()

```java
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
```

```
> java Test
Account@3fee733d
```

toString()

- Since the class **Object** declares a method **toString()**, all objects have this method.
- Either the class inherits a method **toString()** or it redefines it.
- Thus, the statement **a.toString()** is always valid if **a** is a reference variable.

```java
public class Object {

    // ...

    public String toString() {
        return getClass().getName()+"@"+Integer.toHexString(hashCode());
    }
}
```

```java
public class Account {

    private int id;
    private String name;

    public Account(int id, String name) { ... }

    // ...

    public String toString() {
        return "Account: id = " + id + ", name = " + name;
    }
}
```

toString()

```java
Account a;
a = new Account(101, "Marcel");
System.out.println(a);
```

```
> java Test
Account: id = 101, name = Marcel
```

Example

```
import java.awt.TextField;

public class TimeField extends TextField {
    public Time getTime() {
        return Time.parseTime(getText());
    }
}
```

```
// java.lang.Object
//   |
//   +--java.awt.Component
//         |
//         +--java.awt.TextComponent
//               |
//               +--java.awt.TextField
//                     |
//                     +--TimeField
```

instanceof

- Occasionally, one wants to determine whether a (polymorphic) variable designates an object of a given class or one of its subclasses.
  - We then use the operator **instanceof** or the instance method **isInstance**.
- If, on the other hand, one wants to know if a (polymorphic) variable designates an object of a certain class, but not one of its subclasses, then use **this.getClass() == other.getClass()**.

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape instanceof Circle) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape != null && shape.isInstanceof(s)) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

Implementation to be avoided!

- On the next page, the example uses **getClass().getName().equals("Circle")**.
- This solution offers no **type safety**.
    - If I make a typo in the class name for the parameter to the method **equals**, it is still a well-formed string, it will be compiled, but the program will not work as expected.
        - With the first two approaches, this error is detected at compile time.
    - Later, if I change the class name ("*refactor*") to **Cercle** (French for "circle"), with the first two approaches, the compiler will find all cases where I use "**ref instanceof Circle**", but not **getClass().getName().equals("Circle")**.

```java
public class Test {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        Shape s = new Circle(100.0, 200.0, 10.0);

        shapes[0] = s;
        shapes[1] = null;
        shapes[2] = new Rectangle(50.0, 50.0, 10.0, 15.0);
        shapes[3] = new Circle();
        shapes[4] = new Rectangle();

        int count = 0;

        for (Shape shape : shapes) {
            if (shape.getClass().getName.equals("Circle")) {
                count++;
            }
        }

        System.out.println("There are " + count + " circles");
    }
}
```

getClass()

- The contract of the method **equals** requires that the method be symmetrical. That is, **a.equals(b)** and **b.equals(a)** gives the same result.
- If **instanceof** were used, this property might not be verified in the context of a class hierarchy where the method **equals** is redefined in a subclass.
- It is therefore preferable to use **this.getClass() == other.getClass()**, as shown on the next page.
- `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html`

```java
public class Account {
    private int id;  private String name;
    public Account(int id, String name) { ... }
    public boolean equals(Object o) {
        boolean result = true;
        if (o == null) {
            result = false;
        } else if (this.getClass() != o.getClass()) {
            result = false;
        } else {
            Account other = (Account) o;
            if (id != other.id) {
                result = false;
            } else if (name == null && other.name != null) {
                result = false;
            } else if (name != null && ! name.equals(other.name) ) {
                result = false;
            }
        }
        return result;
    }
}
```

Prologue

Summary

- Inheritance allows for the creation of **polymorphic** methods.
- A reference variable of type **T** can be used to store the reference of objects from the class **T** or any of its **subclasses**.
- When a **superclass** declares an **abstract** method, it forces the **subclasses** to provide an implementation for the method.
- A class that declares an **abstract method** must be **abstract**.
- One cannot create an object from an **abstract** class.
- **Object** is the most general class in Java.
- All the classes inherit directly or indirectly from the class **Object**.
- **Object** declares the methods **equals**, **toString**, **getClass**, etc.
- All objects in Java have a method **equals** and **toString**.
- Subclasses can override methods.
- The **name lookup** mechanism always starts with the class of the object, not the compile time type of the reference variable (unless the method is static or final). Called **dynamic** or **late binding**.

Next module

- **Generics**

References I

E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)
University of Ottawa