Data Structures & Algorithms

Object-oriented Programming in Java

Maastricht University

OOP in Java

A. Classes and Objects

B. Interface

C. Inheritance and polymorphism

D. Generics

OOP in Java

- Classes and objects
  - Encapsulate state and functionality into modules
  - Hide implementation details

Classes and Objects

```
class BankAccount {
    private int balance = 100;
    private String owner;

    public BankAccount(String owner) {
        this.owner = owner;
    }
    public void deposit(int amount) {
        if(deposit > 0)
            balance += amount;
    }
    ...
}
```

Classes and Objects

- Suppose customers receive a € 50 bonus when they deposit more than € 100

```
// Call this method when you deposit more than 100
public void addBonus() {
      balance += 50;
}
```

- Problem?

Classes and Objects

- Hide the details from the user of the class

```
public void deposit(int amount) {
        if(deposit > 0)
            balance += amount;
        if(deposit > 100)
            addBonus();
}

private void addBonus() {
        balance += 50;
}
```

OOP in Java

- Interfaces
  - Defines types and behavior
  - Independent of functionality

```
interface Vehicle {
      void drive(int distance);
      void steer(int degrees);
}
```

Interfaces

```
class Car implements Vehicle {
    ...
    public void drive(int distance) {
        wheels.spin(distance * 100);
    }
    public void steer(int degrees) {
        steeringWheel.turn(degrees / 2.0);
    }

    public void honk() {
        soundsystem.start();
    }
    ...
}
```

Interfaces

```
class PirateShip implements Vehicle {
    ...
    public void drive(int distance) {
        shoutAtCrew(“Hoist the sails!”);
    }
    public void steer(int degrees) {
        shoutAtCrew(“Avast ye matey!”);
    }
    ...
}
```

Interfaces

- Usage of interfaces

```
PirateShip ship = new PirateShip();
Car bmw = new Car();

bmw.steer(15);
ship.steer(15);
```

Interfaces

- Usage of interfaces

```
Vehicle ship = new PirateShip();
Vehicle bmw = new Car();

bmw.steer(15);
ship.steer(15);

bmw.honk();   ????
```

Interfaces

- Usage of interfaces

```
Vehicle ship = new PirateShip();
Vehicle bmw = new Car();

bmw.steer(15);
ship.steer(15);

~~bmw.honk();~~
((Car)bmw).honk();
```

bmw is defined as vehicle!

OOP in Java

- Inheritance
  - Define both behavior and partial functionality
  - Inherited by subclasses

```
class Super{ ... }
class Sub extends Super{ ... }
```

Inheritance and polymorphism

- Abstract classes
  - Cannot be instantiated only inherited

```
abstract class Super{ ... }

~~Super super = new Super():~~
```

Inheritance and polymorphism

- Polymorphism is the ability of an object to take on *many forms*.

- A common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Inheritance and polymorphism

```
public interface Vegetarian{ ... }
public class Animal{ ... }

public class Deer extends Animal implements Vegetarian {
    ...
}

Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

OOP in Java

- Suppose you want to write a single method to print an array of:
  - Integers
  - Strings
  - Dates
  - Objects

- How can you do this?
  - Generics!

Generics

- Parametrize class with type
  - Put parameters (one or multiple) within <>
- Class can refer to it by name
  - Convention: 1-letter name
    - E.g. K for Key
- Type parameter is instantiated by the client. (e.g. E → String)

```
// a parameterized (generic) class
public class name<Type> {…}
or
public class name<Type, Type, ..., Type> {…}
```

Generics

```
public class Foo<T> {
    private T myField;                  // ok
    private T[] myArray;                // ok
    private int size;                   // ok

    public Foo(T param, int size) {
        myField = param;                // ok
        myArray = (T[]) (new Object[10]);// ok
      //  myArray = new T[size];        // error
    }
}
```

- Cannot create objects or arrays of a parameterized type
- Possible: Create variables of that type, parameters, return them, create arrays by casting Object[].
  - Casting to generic types is not type-safe

Generics

```
public class Foo<T> {
    ...
    public int indexOf(T value) {
        for (int i = 0; i < size; i++) {
        // if (myArray[i] == value) {   //error
            if (myArray[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }
}
```

- When testing objects of type `T` for equality, must use `equals`

Generics

- Instantiating the class

```
Foo<String> jan = new Foo(“jan",10);

Foo<Integer> list = new Foo(15, 10);

Foo<Double> list2 = new Foo(15.4, 10);
```

Summary

- Use Interfaces to define common behavior between classes
- Declare class variables as private
  - Use get and set methods to access data
- Use inheritance to promote modular design and code-reuse

- Use generics in all data structures