i219 Software Design Methodology
4. Object-oriented programming language 1

Kazuhiro Ogata (JAIST)

Outline of lecture

- Hello world!
- Class
- Inheritance
- Interface
- Exception
- Exception handling
- Type cast

Hello world! (1)

modifier

class

public & static method that returns nothing (void)

array of strings

a String

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");  } }
```

class System

standard output (System's static field (attribute) whose type (class) is PrintStream)

prints a String and terminates the line (PrintStream's method that returns nothing)

✓ A Java application starts with main(…). Each application needs to have a class in which public static void main(String[] args) is declared.

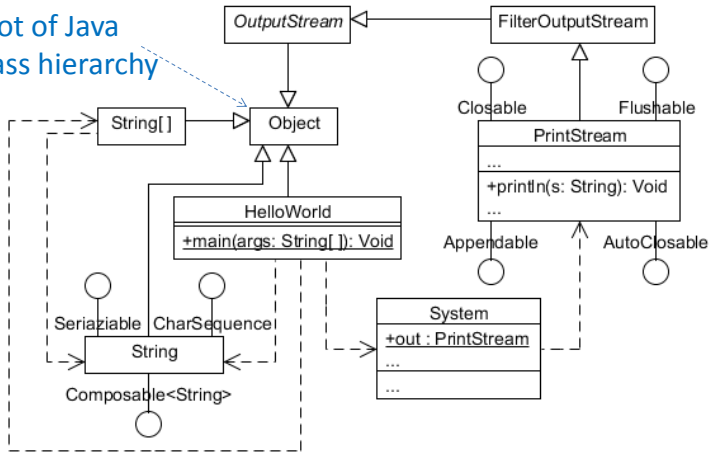
✓ The naming convention for files: each class *C* is written in one class whose name is *C*.java.

```
% javac HelloWorld.java
% java HelloWorld
Hello world!
%
```


Hello world! (2)

- A class diagram of HelloWorld

root of Java class hierarchy



Class (1)

- A class *ClassName* is declared as follows:

fields (attributes), methods, constructors, etc. are declared

cModifiers class *ClassName* { … }

can be accessed from everywhere

the class cannot be extended (cannot have any subclases)

✓ *cModifiers* ::= [public] [abstract] [final] …

can be accessed in the package of the class if public is not given

abstract class from which no object is made

Example: public class Point { … }

Class (2)

- A field (attribute) *fieldName* is declared as follows:

explained later

a reference type (class and interface) & a primitive type, such as int

the result of evaluating this expression is used to initialize the field

fModifiers type fieldName [= *initializer*] ;

all classes can access the field if the class is public

only the class in which the field is declared

explained later

✓ *fModifiers* ::= [public|protected|private] [static] [final] …

some classes such as subclasses

if none of the three access modifiers is given, classes in the same package in which the class is

If *initializer* is omitted, a non-primitive type (a reference type), integer and double fields are initialized as null, 0 and +0.0, respectively.

Example:
```
private double x = –0.5 ;
private double y;
```

Class (3)

- A method *methodName* is declared (defined) as follows

mModifiers type methodName (*parameters*) *methodBody*

method signature

The same access control effects as those to fields

the method cannot be overridden in its subclasses

✓ *mModifiers* ::= [public|protected|private] [abstract] [static] [final] …

explained later

parameter type

parameter name

no implementation

✓ *parameters* ::= $t_1\ p_1, \ldots, t_n\ p_n$

a sequence of statements such as assignments & return statements

✓ *methodBody* ::= ; (if abstract is used) | { … } (otherwise)

Example:

```
public final double getX( ) { return x; }
public final void setX(double x) { this.x = x; }
```

attribute x of the object executing setX is set to parameter x

Class (4)

instance variable in Smalltalk

- Static fields (attributes):

Each object of a class has (a copy of) each non-static field, but a static field is shared by all objects of the class (and its subclasses).

class variable in Smalltalk

n is used to count how many points have been made; every time an object of Point is made, n is incremented.

Example:

```
private static int n = 0;
```

- Static methods :

class methods in Smalltalk

instance methods in Smalltalk

In a static method, non-static fields & non-static methods cannot be used.

Example:

```
public static int howManyPoints() { return n; }
```

A static method can be invoked through an object of the class, but should be invoked through the class such as Point.howManyPoints().

Class (5)

- Objects of a class are made with constructors.
- Constructors are declared in a class whose name is *ClassName* as follows:

the same as those in methods

ctrModifiers ClassName (*parameters*) { … }

a sequence of statements, such as assignments & return statements

✓ *ctrModifiers* ::= [public|protected|private] …

The same access control effects as those to fields

a class may have multiple constructors provided that they have different parameters

Example:

```
public Point( ) { n++; }
```

If x = x is used instead of this.x = x, argument x is set to argument x and field x is initialized as the default value (+0.0).

```
public Point(double x,double y) {
    this.x = x;  this.y = y; n++; }
```


Class (6)

- If any constructors are not explicitly declared in a class *ClassName*, the default constructor is automatically declared:

Constructors like this are called non-arg constructors

ctrModifiers ClassName ( ) { }

The same access control modifier as that of the class ClassName

- If at least one constructor is explicitly declared in a class, the default constructor is not declared.

Example:

If no constructor is explicitly declared in Point, the following default constructor is automatically declared:

```
public Point( ) { }
```

Class (7)

Two method signatures $m_1(P_1)$ and $m_2(P_2)$ are equal if m_1 is the same as m_2 and P_1 is equal to P_2 up to parameter names.

- $p_1{:}t_1, \ldots, p_n{:}t_n$ is equal to $q_1{:}t'_1, \ldots, q_m{:}t'_m$ up to parameter names if and only if $n = m$ and $t_i = t'_i$ for each i.
- Two methods whose name are the same and parameters are different can be declared in a class; one is said to overload the other with each other.

method overloading

```
public double distance() {
   return Math.sqrt(x*x+y*y);
}
```

```
public double distance(Point pt) {
  double tmpx = pt.getX() - x;
  double tmpy = pt.getY() - y;
  return Math.sqrt(tmpx*tmpx+tmpy*tmpy);  }
```

- Two methods whose name are the same and parameters are equal up to parameter names cannot be declared in a class even though the return types of the two methods are different.

Class (9)

- The rest of the class Point:

```
public class Point {  ...
   public final double getY() {  return y;  }
   public final void setY(double y) {  this.y = y;  }
```

The static method sqrt(…) in the class Math is invoked.

```
   public double distance() {  return Math.sqrt(x*x+y*y);  }
```

a concatenation operation of strings

```
   public String toString() {  return "(" + x + "," + y + ")";  }
}
```

aString + *anObject* (or *anObject* + *aString*) is the same as (or converted into) *aString* + *anObject*.toSting() (or *anObject*.toString() + *aString*)

Class (10)

prefix unary operator

argument of new

- An object of a class is made with new plus a constructor with parameters if any.

Example:

(–0.5,0.0) is made

(1.4142135623730951, 1.4142135623730951) is made

```
Point p0 = new Point();
Point p1 =  new Point(1.4142135623730951,1.4142135623730951);
p0.setX(1.0);
p0.setY(1.0);
System.out.println(p1.getX());
System.out.println(p1.getY());
System.out.println(p0.distance());
System.out.println(p1.distance());
System.out.println(Point.howManyPoints());
```

1.0 is set to x in p0 by sending setX(1.0) to p0

1.0 is set to y in p0 by sending setY(1.0) to p0

x in p1 is observed by sending getX() to p1

y in p1 is observed by sending getY() to p1

The distances of p0 & p1 are observed by sending distance() to p0 & p1.

n (#points made) is observed by sending howManyPoint() to Point

Class (12)

- Let us consider a game such that given two points goal & walker and one integer maxSteps, you succeed if walker gets to goal by randomly moving to a next point in maxSteps moves.

this class cannot be extended

```
public final class RandomWalking {
  private final Point goal;
  private final int maxSteps;
  private Point walker;
  public RandomWalking(double gx, double gy,
                       double wx, double wy, int max) {
     goal = new Point (gx,gy);
     walker = new Point(wx,wy);
     maxSteps = max;  }
   public void startWalking() { … }  }
```

assignments to goal & maxSteps are not allowed once they are initialized because of final

goal & maxSteps are initialized when an object of RandomWalking is made

assignments to goal & maxSteps are not allowed here

Class (13)

- startWalking is as follows

```
public void startWalking( ) {
  // goal.setX(10.0);
  // goal = new Point(0.0,0.0);
  // maxSteps = 10;
  int steps = 0;
  while (true) {
     System.out.println("walker: " + walker);
     if (goal.distance(walker) < 1.0) { … break;  }
     if (steps >= maxSteps) { … break;  }
     double dx = Math.random();  double dy = Math.random();
     dx = Math.random() > 0.2 ? -dx : dx;
     dy = Math.random() > 0.2 ? -dy : dy;
     walker.move(dx,dy);
     steps++;  }
```

modification of the contents is allowed; even if uncommented, a compiler does not complain

assignments are not allowed; if uncommented, a compiler complains

got to goal in maxSteps

did not get to goal in maxSteps

the next point to which walker moves is randomly made

Inheritance (1)

- A class can be extended to make a new class.
- Let us make a class of points in 3D space by extending the class Point.

PointIn3D cannot be extended

fields, constructors, methods

```
public final class PointIn3D extends Point { ... }
```

superclass

Point is extended

subclass

- PointIn3D inherits all fields (attributes) & methods from Point; some of them cannot be directly accessed, such as x and y in Point.

A field added: `private double z;`

cannot be directly accessed but can be with getX() and getY()

cannot be directly accessed but can be with howManyPoints()

Each object of PointIn3D has three (copies of) fields x, y, z, and share n with all other objects of Point & PointIn3D.

not only Point

Inheritance (2)

A constructor added:

```
public PointIn3D(double x,double y,double z) {
    super(x,y);    this.z = z;  }
```

Point(x,y) in Point is invoked

field z in the object being created is set to argument z

- In an constructor, at most one constructor in either the current class (this(…)) or the super class (super(…)) may be invoked; this(…) or super(…) should appear at the very beginning place in the constructor; if this(…) is used, this(…) should be different from the constructor.
- If neither super(…) nor this(…) is invoked, a non-arg constructor such as the default one in the superclass is invoked; if a non-arg one is not declared, a compiler complains.

Inheritance (3)

- A type t' is said to be a subtype of a type t if and only if one of the following cases is fulfilled:
  - If t is a class (including an abstract class), then t' extends t (namely that t' is a subclass of t).
  - If t is an interface, then t' extends t (namely that t' is a subinterface of t) or t' implements t (namely that t' is a class (including an abstract class) that implements t).
- The subtype relation is transitive; if t' is a subtype of t and t'' is a subtype of t', then t'' is a subtype of t.
- A type t is a supertype of a type t' if and only if t' is a subtype of t.
- A subtype t' of a type t can be used at the place where t can be used, but not vice versa.

An interface will be explained later; it is like a class in which no fields are declared and all methods are abstract.

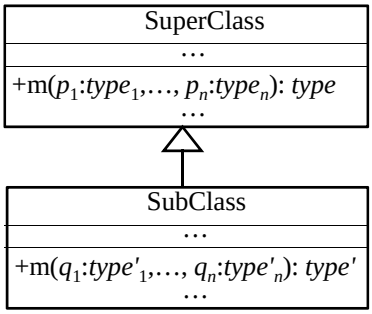
Inheritance (4)

A method added:

this overrides distance() in Point

distance() in Point is invoked

```
public double distance() {
    double disIn2D = super.distance( );
    return Math.sqrt(disIn2D*disIn2D+z*z);  }
```



m in SubClass is said to override m in SuperClass if and only if $type_i$ is the same as $type'_i$ for each i, provided that *type'* should be *type* or a subtype of *type* if *type* is a reference type, and be *type* if *type* is a primitive type; otherwise a compiler complains.

Inheritance (5)

A method added:

pt, an object of PointIn3D, can be used as an argument of distance( ) in Point because PointIn3D is a subtype of Point.

```
public double distance(PointIn3D pt) {
  double disIn2D = super.distance(pt);
  double tmpz = pt.getZ( ) - z;
  return Math.sqrt(disIn2D*disIn2D+tmpz*tmpz);  }
```

This method overloads distance() & distance(Point pt) in Point and distance() in PointIn3D.

Interface (1)

- An interface *InterfaceName* is declared as follows:

methods, etc. are declared

iModifiers interface *InterfaceName* { … }

can be accessed from everywhere

always abstract; can be omitted and usually omitted

✓ *iModifiers* ::= [public] [abstract] …

can be accessed in the package of the class if public is not given

Example:

```
public interface PointInterface {
  double getX();
  double getY();  }
```

all methods in interfaces are abstract & public, which can be omitted and usually omitted

Interface (2)

- An interface is implemented by a class (partially by an abstract class)

an abstract class that implements PointInterface

Example:

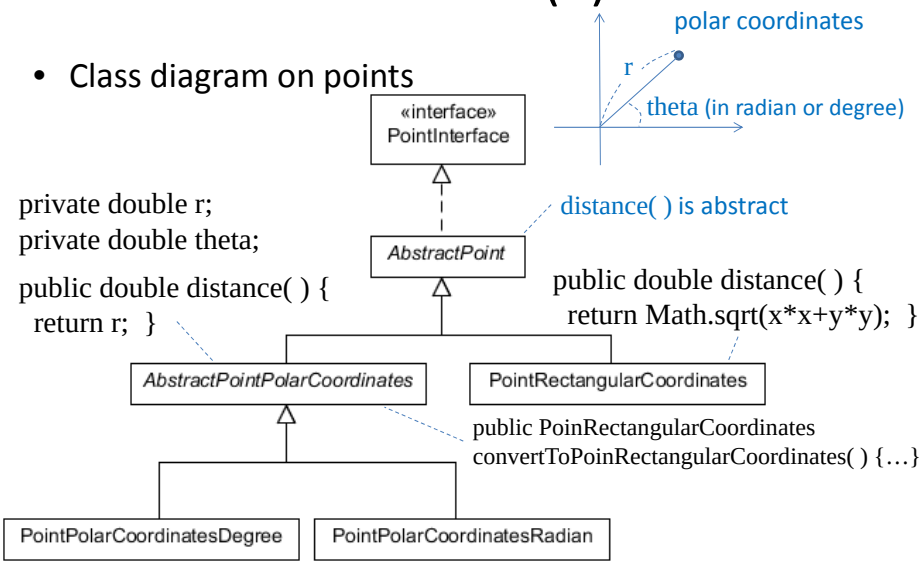
```
public abstract class AbstractPoint
                      implements PointInterface {
  public abstract double distance( );
  public double distance(PointInterface pt) {
    double tmpx = this.getX( ) - pt.getX( );
    double tmpy = this.getY( ) - pt.getY( );
    return Math.sqrt(tmpx*tmpx+tmpy*tmpy);  }  }
```

AbstractPoint has three abstract methods; one is distance( ) and the others are getX( ) and getY( ) that come from PointInterface

- Note that if a class has abstract methods, it should be abstract.

Interface (3)

- Class diagram on points



Exception (1)

- One possible constructor for PointPolarCoordinatesRadian:

```
public PointPolarCoordinatesRadian(double r, double theta) {
  this.setR(r);  this.setTheta(theta);  }
```

What if r is nagative?
One way to deal with such a case is to throw (raise) an exception.
First a class for exceptions warning such a case is declared.

Exception(String msg) is invoked

given by Java library

```
public class PointException extends Exception {
  public PointException(String msg) {
    super(msg);  }  }
```

Note that Exception can be used for this case, but a specific class that extends Exception should be made to let users (programmers) know what exception has occurred.

Exception (2)

- The constructor becomes

```
public PointPolarCoordinatesRadian(double r, double theta)
        throws PointException  {
  if (r < 0.0) {
    throw new PointException("r should not be negative!"); }
  if (theta < 0.0 || theta > 2.0 * Math.PI) {
    throw new PointException(…);  }
  this.setR(r);  this.setTheta(theta);  }
```

declaration that a PointException may be thrown

a PointException is thrown if r is negative

If an exception is thrown, the control moves back along the sequence of invoking constructors and methods until the exception is caught; if the exception is not caught, the application terminates, letting users know that the exception has occurred.

Exception handling (1)

- Let us make an application that asks a user to input two points in Polar Coordinates (where radian is used) and calculate the distance of the two points.

making classes in package java.io for input & output available

array of AbstractPoint whose length is 2 is made; but no object of AbstractPoint is made

```
import java.io.*;
public class DistanceBetweenTwoPoints {
  public static void main(String[] args) throws IOException  {
    int i = 0;  String line;
    double[] r  = new double[2];
    double[] theta = new double[2];
    AbstractPoint[] p = new AbstractPoint[2];
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    System.out.println("Input two points in polar coordinate system");
```

it decodes bytes read from the standard input into characters

standard input

it reads character from the standard input in a buffered way

Exception handling (2)

```
while (i < 2) {
  try {
    System.out.print("r" + i + ": ");
    line = br.readLine().trim();
    r[i] = Double.parseDouble(line);
    System.out.print("theta" + i + ": ");
    line = br.readLine().trim();
    theta[i] = Double.parseDouble(line);
    p[i] = new PointPolarCoordinatesRadian(r[i],theta[i]);
    i++;
  } catch(PointException e) {
    System.err.println(e);
  } catch(NumberFormatException e) {
    System.err.println(e);  }  }
```

it reads a line of text and returns the line excluding a line break as a String

it removes white spaces at both sides of the String

it converts a String line into a double; if line does not express a double, such as "abc", a NumberFormatException is thrown

a PointException may be thrown

standard error

a PointException is caught

a NumberFormatException is caught

Type cast

since the type of p[j] is AbstractPoint, it is necessary to cast the type to AbstractPointPolarCoordinate so that convertToRectangularCoordinates can be used

```
double dis = p[0].distance(p[1]);
AbstractPoint[] q = new AbstractPoint[2];
for (int j = 0; j < 2; j++) {
  AbstractPointPolarCoordinates tmp
     = (AbstractPointPolarCoordinates) p[j];
  q[j] = tmp.convertToRectangularCoordinates();
}
System.out.println(p[0] + "-->" + p[1] + ": " + dis);
System.out.println(p[0] + ": " + q[0]);
System.out.println(p[1] + ": " + q[1]);  }  }
```

if convertToRectangularCoordinates( ) is sent to p[j], a compiler complains that AbstractPoint does not have the corresponding method.

Summary

- Hello world!
- Class
- Inheritance
- Interface
- Exception
- Exception handling
- Type cast