5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Relationships between objects or classes: Using references, we can refer from one object of a class to another object of the same or a different class.

> *References define **relationships** between objects (instance or object level) and between classes (type or class level)*

Types of relationships:

- **organizational**
  e.g., a list element refers to its successor
- **application-specific**
  e.g., a student is enrolled for a field of study
- **”is-a”** (specialization, generalization, **inheritance**)
  e.g., an automobile is a vehicle (and has all properties of a vehicle, in addition to properties specific to an automobile)
- **”has-a”** (composition)
  e.g., a car has an engine, has four wheels, etc.

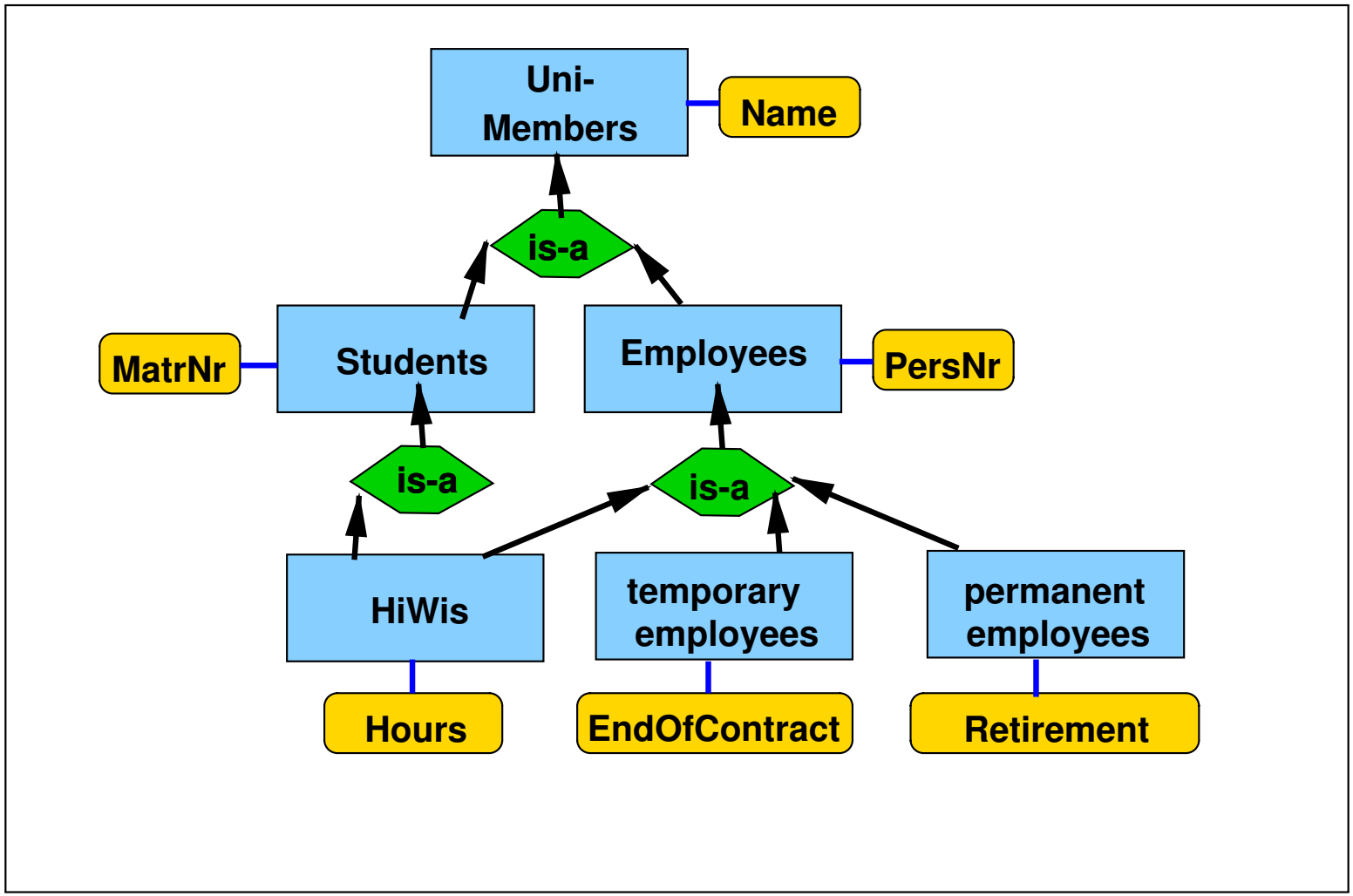
Inheritance:

- Goal: Reusability!
- Definition of new classes based on existing classes
  - Reuse data and behavior from the existing class
  - add new properties
- Superclass is **specialized** to subclass
  - objects with additional, more specific properties.
- Superclass **generalizes** its subclasses
  - generalization of the objects to their common properties.
- Keyword in Java: subclass "`extends`" superclass
- Principle:
  - Subclass inherits the variables and methods of the superclass,
  - extends them by new variables and methods and
  - may overwrite methods to adapt to the extended properties of the objects.

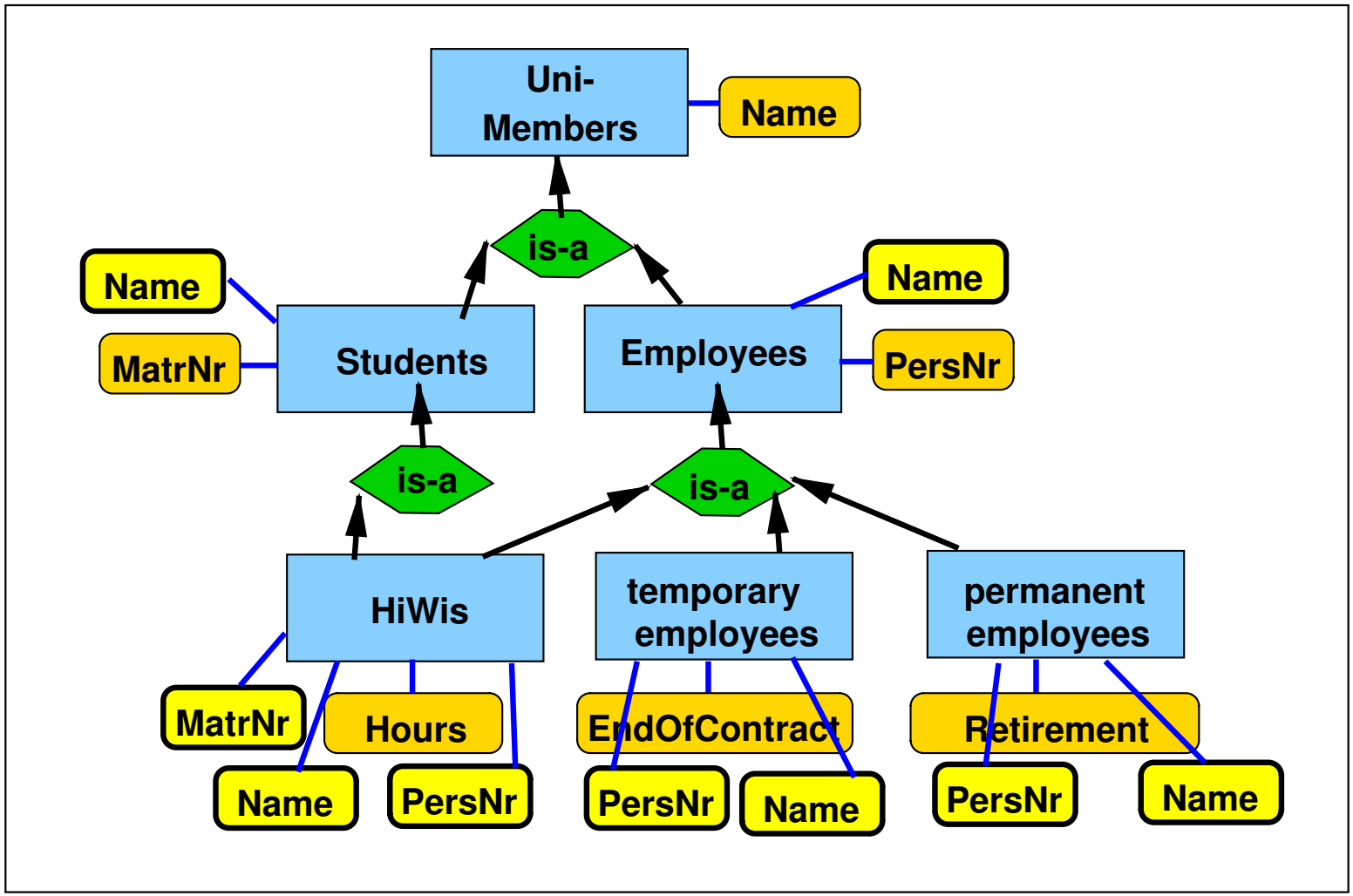
Class hierarchy

- The inheritance defines a **class hierarchy**.
- The most general class in Java is the class `Object`.
- An object of a subclass is also an object of all corresponding superclasses, e.g.:
  A sports car is a car,
  is also a land vehicle,
  is also a means of transportation,
  is also a ...
  - **subclass**: sports car
  - **direct superclass**: car
  - **indirect superclasses**: land vehicle, means of transportation, ...
- Superclasses usually represent larger sets of objects than subclasses, e.g.:
  - **superclass** land vehicle includes cars, buses, trucks, motorbikes, bicycles, ...
  - **subclass** car only contains vehicles with special properties.

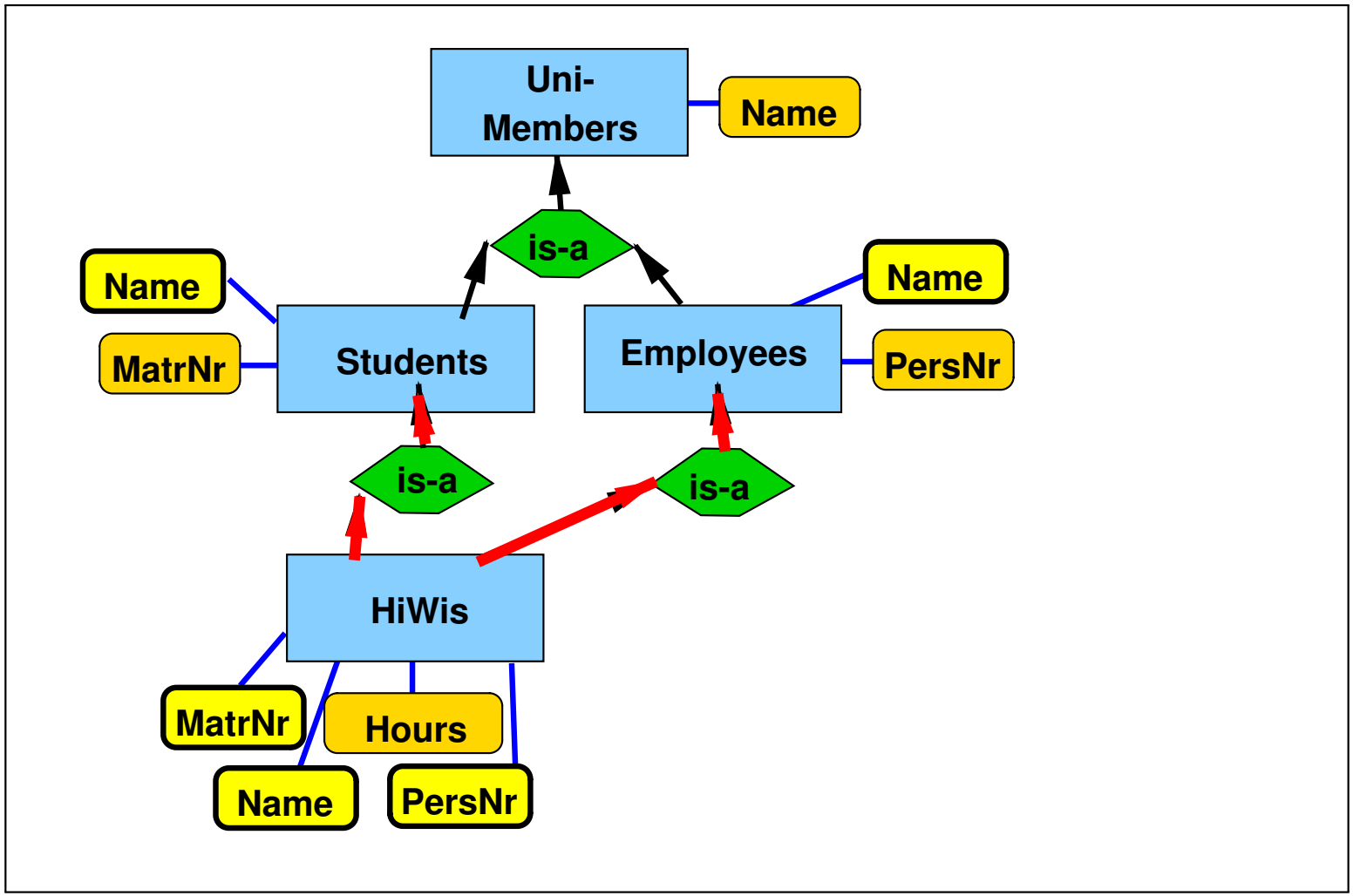
Example: **University members**,
initially without inherited properties:



Example: **University members**,
with inherited properties:



Attention: Example includes **multiple inheritance**!
(forbidden in Java, only ‘workaround’ possible)

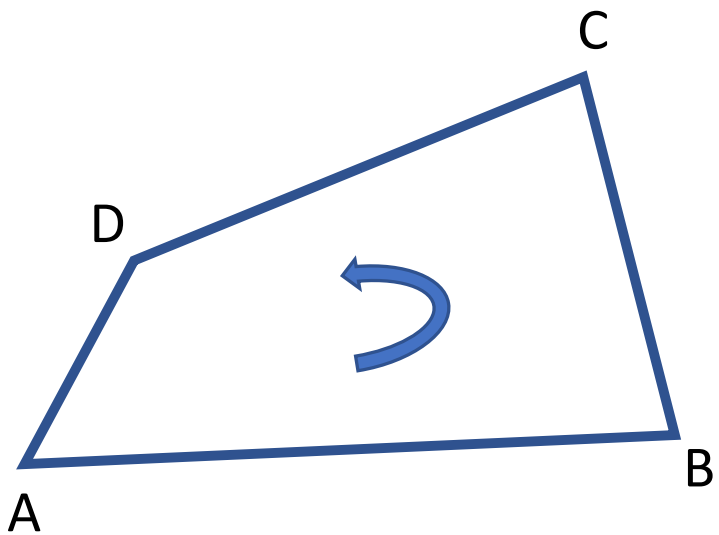


Inheritance hierarchy in Java

- Inheritance hierarchies were developed in the 60ies when trying to process natural language with the computer. Thus inheritance hierarchies often followed terminological is-a relationsips.
- Inheritance hierarchies have a tree-like structure, where every class is super- and/or subclass.
- In Java the root of the inheritance hierarchy (i.e., the most general class) is the class `Object`.
- In the following Java examples, reuse of class definitions is in the focus.

Inheritance hierarchy in Java

- We consider specific geometric objects, namely quadrilaterals (polygons with four sides and four corners):
  - Quadrilaterals
  - Parallelograms
  - Rectangles
- We represent a geometric object by its corners, i.e., for a quadrilateral, we store four corner points (in counter-clockwise order)



As a foundation for representing these objects, we use the class `Pt` which represents points in the plane.

```
1  public class Pt {
2   private int x, y;
3
4   public Pt(int xVal, int yVal){
5    x = xVal ; y = yVal ; }
6
7   public void setX( int xVal ) {x = xVal;}
8
9   public void setY( int yVal ) {y = yVal;}
10
11  public int getX() {return x;}
12  public int getY() {return y;}
13
14  public double dist(Pt p){
15    return Math.sqrt((x-p.getX())*(x-p.getX())
16                     +(y-p.getY())*(y-p.getY()));}
17
18  public String toString() {return "("+x+";"+y+")";}
19 }
```

Class for Quadrilateral objects

```
1  public class Quadrilateral {
2   private Pt a,b,c,d;
3
4   public Quadrilateral() { }
5   public Quadrilateral
6          (Pt p1, Pt p2,
7           Pt p3, Pt p4){
8    a=p1; b=p2; c=p3; d=p4;
9   }
10
11
12
13
14
15  public Pt getA() {return a;}
16  public Pt getB() {return b;}
17  public Pt getC() {return c;}
18  public Pt getD() {return d;}
19
20  ...
```

Class for Parallelogram objects

```
1  public class Parallelogram {
2   private Pt a,b,c,d;
3
4   public Parallelogram() { }
5   public Parallelogram
6          (Pt p1, Pt p2,
7           Pt p3){
8    a=p1; b=p2; c=p3;
9    int dx=a.getX()-b.getX();
10   int dy=a.getY()-b.getY();
11   d=new Pt(c.getX()+dx,
12            c.getY()+dy);
13  }
14
15  public Pt getA() {return a;}
16  public Pt getB() {return b;}
17  public Pt getC() {return c;}
18  public Pt getD() {return d;}
19
20  ...
```

```
1   ...
2
3
4
5
6
7
8
9
10  public double perimeter() {
11   return a.dist(b)+b.dist(c)
12         +c.dist(d)+d.dist(a);}
13
14  public String toString(){
15   return "["+a+","+b+","
16             +c+","+d+"]";}
17 }
```

```
1   ...
2   public double area() {
3    return
4      a.getX()*b().getY()
5     +a.getY()*c().getX()
6     +b.getX()*c().getY()
7     -b.getY()*c().getX()
8     -a.getY()*b().getX()
9     -a.getX()*c().getY();}
10
11  public double perimeter() {
12   return a.dist(b)+b.dist(c)
13         +c.dist(d)+d.dist(a);}
14
15  public String toString(){
16   return "PG["+a+","+b+","
17             +c+","+d+"]";}
18 }
```

These *independent* definitions of the classes **`Quadrilateral`** and **`Parallelogram`** show commonalities, e.g.:

- equal: variables `a`, `b`, `c`, `d`, methods `getA` etc., `perimeter`
- different are: constructor, methods `area`, `toString`

Now: **Parallelogram** as subclass of **Quadrilateral**:

```
1  public class Quadrilateral {
2
3  private Pt a,b,c,d;
4
5   public Quadrilateral() { }
6   public Quadrilateral
7           (Pt p1, Pt p2,
8            Pt p3, Pt p4){
9    a=p1; b=p2; c=p3; d=p4;
10  }
11
12
13
14
15
16
17
18  public int getA() {return a;}
19  public int getB() {return b;}
20  public int getC() {return c;}
21  public int getD() {return d;}
22  ...
```

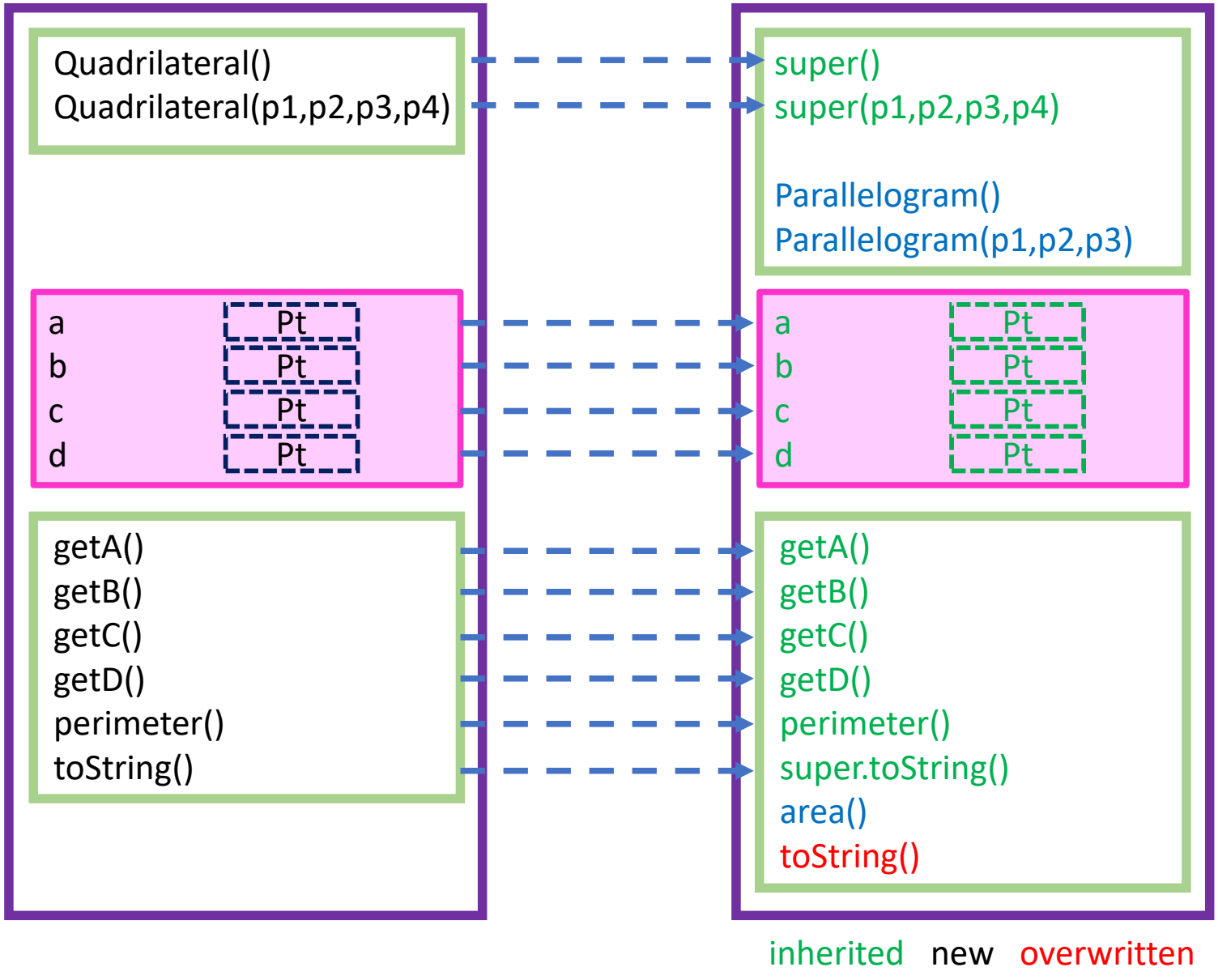
```
1  public class Parallelogram
2   extends Quadrilateral{
3   // a,b,c,d are inherited
4
5   public Parallelogram() { }
6   public Parallelogram
7        (Pt p1, Pt p2,
8         Pt p3){
9    super(p1,p2,p3,
10         new Pt(p3.getX()
11                +p1.getX()
12                -p2.getX(),
13                 p3.getY()
14                +p1.getY()
15                -p2.getY()));
16  }
17
18  // getA() is inherited
19  // getB() is inherited
20  // getC() is inherited
21  // getD() is inherited
22  ...
```

```
1   ...
2
3
4
5
6
7
8
9
10
11  public double perimeter() {
12   return a.dist(b)+b.dist(c)
13         +c.dist(d)+d.dist(a);}
14
15
16  public String toString(){
17   return "["+a+","+b+","
18             +c+","+d+"]";}
19 }
```

```
1   ...
2   public double area() {
3    return
4      getA().getX()*getB().getY()
5     +getA().getY()*getC().getX()
6     +getB().getX()*getC().getY()
7     -getB().getY()*getC().getX()
8     -getA().getY()*getB().getX()
9     -getA().getX()*getC().getY();}
10
11  // perimeter inherited
12
13
14
15  // toString, overwritten
16   public String toString(){
17    return "PG"
18           +super.toString();}
19  }
```

- **a**,**b**,**c** and **d** are declared as **private** in **Quadrilateral** and thus cannot be directly accessed in **Parallelogram**.
- Alternative: Modifier **protected**, between **public** and **private**: accessible in all subclasses and all classes of the same package (see later)
- Three methods of the superclass **Quadrilateral** are not directly accessible in the subclass **Parallelogram**, but only with the keyword **super** to access the superclass:
  - the two constructors **Quadrilateral()** and **Quadrilateral(p1,p2,p3,p4)**, only via **super()** or **super(p1,p2,p3,p4)** at the start of the constructors of **Parallelogram**.
  - the overwritten method **toString()** as **super.toString()**

  The calls **super.toString()**, **super()** and **super(p1,p2,p3,p4)** can be used in this form only in direct subclasses
- **Quadrilateral** and **Parallelogram** are also (indirect) subclasses of **Object** and thus have additional methods (later...)!


Quadrilateral()
Quadrilateral(p1,p2,p3,p4)
a
b
c
d
Pt
Pt
Pt
Pt
getA()
getB()
getC()
getD()
perimeter()
toString()
super()
super(p1,p2,p3,p4)
Parallelogram()
Parallelogram(p1,p2,p3)
a
b
c
d
Pt
Pt
Pt
Pt
getA()
getB()
getC()
getD()
perimeter()
super.toString()
area()
toString()
inherited
new
overwritten

Visibility modifiers for the instance variables **`a,b,c,d`** in **`Quadrilateral`**:

- **`private`**:
  then access to **`a,b,c,d`** from the subclass **`Parallelogram`** not directly possible, only using inherited methods, i.e.,
  `getA(),getB(),getC(),getD(),perimeter(), super.toString(), super(), super(p1,p2,p3,p4)`
- **`protected`**: direct access from subclass **`Parallelogram`** allowed
- **`public`**: direct access allowed in all classes

Recommendation: Variables should be **`private`**, without direct access outside the defining class

Thus: define **`a`**,**`b`**,**`c`** and **`d`** in **`Quadrilateral`** as **`private`** and access them only via methods!

Complete example `C5E09_Inheritance`:

- file `Pt.java`: class `Pt` as before for representing points in the plane.
- file `Quadrilateral.java`: class `Quadrilateral` as above, with **`private`** modifier for the instance variables.
- file `Parallelogram.java`: Klasse `Parallelogram` as subclass of `Quadrilateral`.
- file `Rectangle.java`: extension of the class hierarchy by the class `Rectangle` as subclass of `Parallelogram`.
- additionally `main` in the class `Rectangle` as a method for testing.

file `Rectangle.java`: extension of the class hierarchy by the class `Rectangle` as subclass of `Parallelogram`

```
1  public class Rectangle extends Parallelogram {
2
3   public Rectangle() {}
4
5   public Rectangle( Pt p1, Pt p2, Pt p3) {
6     super( p1, p2, p3 );
7   }
8
9   public double area() {
10     return getA().dist(getB())
11           *getB().dist(getC());
12  }
13
14  public double diag() {
15     return getA().dist(getC());
16  }
17
18  @Override public String toString() {
19     return "Rect:"+super.toString();
20  }
21 ...
```

in file `Rectangle.java`: method for testing class `Rectangle`

```
1    public static void main( String[] args ) {
2       System.out.println("Test method for class hierarchy.\n");
3
4       Rectangle c = new Rectangle
5                         (new Pt(0,0),new Pt(2,0),new Pt(2,2));
6
7       System.out.println(rect);
8       System.out.println("area  : " +rect.area());
9       System.out.println("perim : "+rect.perimeter());
10      System.out.println("diag  : "+rect.diag());
11
12   }
13 }
```

The generation of a subclass object generates a sequence of **constructor** calls:

- The subclass constructor first(!) implicitly or explicitly calls the superclass constructor.
- The constructor calls follow the class hierarchy up to the constructor of the class `Object`.
- The method bodies of the constructors are then executed from the superclass to the subclass (along the class hierarchy ‘top down’).
- Explicit calls of a constructor of the superclass must always be done at the beginning of the declaration of the calling constructor.

- The **garbage collector** removes objects from memory to which no references refer.
- In early versions of Java, a **finalizer** was recommended, but this has been `deprecated` since Java 9.
- Many programs never make use of finalizers!
- In practice the gargabe collector works only after some delay. i.e., even calling it directly may not immediately start garbage collection.

Example `K5B10E_Constructors`:

(a) class **`SuperClass`** for constructor example:

```
1  public class SuperClass {
2
3     private String superData;
4
5     public SuperClass() {
6        superData = "-SP-";
7        System.out.println("super default constructor: " + this);
8     }
9
10    public SuperClass(String name) {
11       superData=name ;
12       System.out.println("super special constructor: " + this);
13    }
14
15    @Override public String toString() {
16       return "superData␣" + superData;
17    }
18 }
```

(b) class **`SubClass`** for constructor example:

```
1  public class SubClass extends SuperClass  {
2
3     private String subData;
4
5     public SubClass()  {
6        subData = "-sb-";;
7        System.out.println("sub default constructor:   " + this);
8     }
9
10    public SubClass(String supername, String subname)  {
11       super( supername );
12       subData  = subname;
13       System.out.println("sub special constructor:   " + this);
14    }
15
16    @Override public String toString() {
17       return "subData " + subData + ", "+ super.toString();
18    }
19 }
```

(c) test class:

```
1  public class Test {
2     public static void main( String args[] ) {
3        SuperClass a = new SuperClass ("-AA-");
4        SubClass   b = new SubClass   ();
5        SubClass   c = new SubClass   ("-C1-","-C2-");
6        System.out.println();
7        a = null;  // marked for garbage collection
8        b = null;  //                  -  "  -
9        c = null;  //                  -  "  -
10       System.gc();  // call garbage collector
11 }  }
```

(d) example output:

```
1 super special constructor: superData -AA-
2 super default constructor: subData null, superData -SP-
3 sub default constructor:   subData -sb-, superData -SP-
4 super special constructor: subData null, superData -C1-
5 sub special constructor:   subData -C2-, superData -C1-
```

5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Important: When can we store a reference to a class in a variable of *a*nother class?

Possible combinations of variables and references with inheritance (with example **`Parallelogram`** as subclass of **`Quadrilateral`**):

$x = r$	reference r to superclass	reference r to subclass
variable x of superclass type	yes	**yes** (Parallelogram 'is-a' Quadrilateral)
variable x subclass type	**no** (Quadrilateral 'is-NOT-a' Parallelogram)	yes

Example C5E11_References (using the known classes Quadrilateral and Parallelogram):

```
1  public class References {
2   public static void main( String[] args ) {
3    Pt a=new Pt(0,0), b=new Pt(1,0), c=new Pt(1,1), d=new Pt(0,1);
4
5  // Superclass reference to Superclass object:
6     Quadrilateral ql = new Quadrilateral(a,b,c,d);
7
8  // Subclass reference to Subclass object:
9     Parallelogram pg = new Parallelogram(a,b,c);
10
11 // Superclass reference to Subclass object:
12    Quadrilateral qlRef = pg;
13
14 // Subclass reference to Superclass object:
15 // Parallelogram pgRef = ql;  // compile error
16
17    System.out.println( ql.toString() +
18       "\n (call toString() of the class Quadrilateral)\n\n");
19
20    System.out.println( pg.toString() +
21       "\n (call toString() of the class Parallelogram)\n\n");
22
23    System.out.println( qlRef.toString() +
24       "\n (call toString() of the class Parallelogram)\n\n");
25 }}
```

Casting of a superclass reference is possible if the test **`instanceof`** for the subclass property is successfull.

`instanceof` is a special boolean operator that has a reference and a class name as parameters, see Example `K5B12E_ReferenceCast`:

```
1  public class ReferenceCast {
2   public static void main( String[] args ) {
3
4     Pt a=new Pt(0,0), b=new Pt(1,0), c=new Pt(1,1), d=new Pt(0,1);
5
6     Quadrilateral  qlRef  = new Parallelogram(a,b,c,d);
7
8     Parallelogram pgRef = new Parallelogram();
9     if ( qlRef instanceof Parallelogram )
10       pgRef = (Parallelogram) qlRef;
11
12    System.out.println(  qlRef.toString()
13       + "\n␣␣call toString() of the class Parallelogram"
14       + "\n␣␣superclass reference to subclass object\n\n");
15
16    System.out.println( pgRef.toString()
17       + "\n␣␣call toString() of the class Parallelogram"
18       + "\n␣␣superclass reference to subclass object "
19       + "cast to Parallelogram\n");
20 } }
```

- Annotation: language component to include meta data into the source code, used by the compiler
- most important example: `@Override`
- purpose: annotate methods that overwrite superclass methods
- if no corresponding superclass method exists: compile error!
- helps to identify typos, e.g. `@Override public tostring()` instead of `@Override public toString()`
- other annotations: `@Deprecated`

5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Methods in Java can be defined as ‘abstract’ (without body):

```
1 public abstract double perimeter (); // no '{}'
```

A class including an abstract method must be defined as abstract as well:

```
1 public abstract class Figure {
2    public abstract double perimeter ();
3    public abstract double area ();
4 }
```

- Abstract classes cannot be instantiated, they only serve to construct subclasses.
- Classes can also defined as abstract if they include non-abstract methods. (This prevents an instantiation.)

- A subclass of an abstract class can be instantiated, if it overwrites and implements all abstract methods of the superclass.
- A subclass that does not implement all inherited abstract methods is itself abstract.

	`class`	`abstract class`	`final class`
Generation of objects	yes	no	yes
Creation of subclasses	yes	yes	no

Example: *abstract* class **Shape**

```
1  public abstract class Shape {
2
3  // concrete constructor:
4  // for (usually implicit) calls by subclass constructors
5     protected Shape() { }
6
7  // abstract methods:
8
9     public abstract double perimeter();
10
11 //  concrete methods:
12
13    public String getName() {
14       return this.getClass().toString();
15      // name of the class of an object, as String
16    }
17 }
```

Quadrilateral as concrete subclass of the abstract class **Shape**

```
1  public class Quadrilateral extends Shape {
2
3   private Pt a,b,c,d;
4
5   public Quadrilateral() { }
6
7   public Quadrilateral(Pt p1, Pt p2, Pt p3, Pt p4) {
8     a=p1; b=p2; c=p3; d=p4; }
9
10 // overwrite abstract methods with concrete methods
11
12  @Override public double perimeter() {
13   return a.dist(b)+b.dist(c)+c.dist(d)+d.dist(a);}
14
15 // additional methods:
16
17   public int getA() {return a;}
18   public int getB() {return b;}
19   public int getC() {return c;}
20   public int getD() {return d;}
21
22  @Override public String toString(){
23   return "["+a+","+b+","+c+","+d+"]";}
24 }
```

Parallelogram as indirect subclass of the abstract class **Shape**

```
1  public class Parallelogram extends Quadrilateral {
2  // new constructors:
3   public Parallelogram() { }
4   public Parallelogram (Pt p1, Pt p2, Pt p3) {
5    super(p1,p2,p3, new Pt(p3.getX()+p1.getX()-p2.getX(),
6                           p3.getY()+p1.getY()-p2.getY()));
7   }
8  // new methods:
9   public double area() {
10   return getA().getX()*getB().getY()+getA().getY()*getC().getX()
11         +getB().getX()*getC().getY()-getB().getY()*getC().getX()
12         -getA().getY()*getB().getX()-getA().getX()*getC().getY();}
13
14 // inherited (from Shape): getName
15
16 // inherited (from Quadrilateral): perimeter
17
18 // overwritten (from Quadrilateral):
19  @Override public String toString() {
20    return "PG" + super.toString();
21  }
22 }
```

Rectangle as indirect subclass of the abstract class **Shape**

```
1  public class Rectangle extends Parallelogram {
2  // new:
3   public Rectangle() { }
4   public Rectangle(Pt p1, Pt p2, Pt p3 ) {
5       super( p1, p2, p3 );
6   }
7
8   public double diag() {
9       return getA().dist(getC());
10  }
11 // overwritten (from Parallelogram):
12 @Override public double area() {
13      return getA().dist(getB())
14             *getB().dist(getC());
15  }
16  }
17  @Override public String toString() {
18      return "Rect:"+super.toString()
19
20 // inherited (from Quadrilateral): perimeter
21 // inherited (from Shape): getName
22  }
```

Example C5E13_Shape_Abstract: calling '**polymporphic**' methods

```
1  public class ShapeTest {
2   public static void main( String args[] ) {
3
4     Quadrilateral ql = new Quadrilateral(new Pt(0,0),new Pt(1,0),
5                                          new Pt(1,1), new Pt(0,1));
6     Parallelogram pg = new Parallelogram(new Pt(0,0),new Pt(2,0),
7                                          new Pt(2,2));
8     Rectangle re = new Rectangle(new Pt(0,0),new Pt(3,0),
9                                  new Pt(3,3));
10
11  // usual access to methods of each class:
12    System.out.println("\n\n" + ql.getName()  + "␣(direct)\n"
13          + "[␣" + ql.getA()  + "␣,␣" + ql.getB()
14          + "␣,␣"+ ql.getC()  + "␣,␣" + ql.getD() + "]");
15
16    System.out.println("\n\n" + pg.getName() + "␣(direct)\n"
17          + "[␣" + pg.getA()  + "␣,␣" + pg.getB()
18          + "␣,␣"+ pg.getC()  + "␣,␣" + pg.getD() + "]");
19
20    System.out.println("\n\n" + re.getName() + "␣(direct)\n"
21          + "[␣" + re.getA()  + "␣,␣" + re.getB()
22          + "␣,␣"+ re.getC()  + "␣,␣" + re.getD() + "]");
23
24    ...
```

```
1    ...
2    Shape arrayOfShapes[] = new Shape[ 3 ];
3    arrayOfShapes[ 0 ] = ql;assignment of subclass references!
4    arrayOfShapes[ 1 ] = pg;
5    arrayOfShapes[ 2 ] = re;
6  // access via concrete methods of Shape/Quadrilateral:
7    for ( int i = 0; i < arrayOfShapes.length; i++ ) {
8       System.out.println("\n\n"
9          +  arrayOfShapes[ i ].getName() + "␣(via␣Shape)\n");
10   }
11 // use of overwritten methods:
12   for ( int i = 0; i < arrayOfShapes.length; i++ ) {
13      System.out.println("\n\n"
14         +  arrayOfShapes[ i ].getName() + "␣(via␣Shape)\n"
15         + "\n" + arrayOfShapes[ i ].toString()
16         + "\nperimeter = " + arrayOfShapes[ i ].perimeter());
17       }
18   }
19 }
```

Polymorphism:

- Methods are **polymorphic** if they can be used with objects of different types.
- Implementation in Java:
  1. Definition the method in a superclass (abstract or not)
  2. Overwriting of the method in the corresponding subclasses.
  3. Method can be applied for objects of classes where the implementation of the method is defined.
- The array **`arrayOfShapes`** is defined on the abstract class **`Shape`** and can thus contain elements of arbitrary (non-abstract) subclasses of **`Shape`**.
- The methods **`perimeter`**, **`getName`** can be called for the abstract class **`Shape`**, the (overwriting) implementation of the corresponding subclass is executed.
- The methods **`getA`**, **`getB`**, **`getC`**, **`getD`** can be called only for objects of the class **`Quadrilateral`** or its subclasses.
  The method **`area`** and **`toString`** are polymorphic since the implementations in different subclasses are different.

5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

Interfaces are similar to abstract classes and are used to define interfaces.

	Interfaces	**abstract classes**
identifier	**`interface`** ifname	**`abstract class`** classname
methods	all implicitly abstract	abstract and concrete
variables	only **`static final`**	arbitrary
constructors	none	default or explicit
subclasses	**`implements ifname`**	**`extends classname`**
subinterfaces	**`extends ifname`**	-

A class can

- be a subclass of *at most one* superclass (extends), but
- implement *arbitrary many* interfaces (implements)

Thus:

> *Interfaces in Java implement a restricted form of multiple inheritance.*

Example interface **`Shape`**

- no constructor (constructor not useful since any variables must be `static final`)
- All methods are implicitly(!) defined as **`abstract`**.

```
1 public interface Shape {
2
3     // no constructor !
4     // all methods abstract
5
6     public double perimeter();
7     public String getName();
8 }
```

`Quadrilateral` as concrete subclass of interface `Shape`:

- overwrites abstract methods with concrete (implemented) methods

```
1  public class Quadrilateral implements Shape {
2
3   private Pt a,b,c,d;
4
5   public Quadrilateral() { }
6   public Quadrilateral(Pt p1, Pt p2, Pt p3, Pt p4) {
7     a=p1; b=p2; c=p3; d=p4; }
8
9  //implementation as before:
10    public int getA() {return a;}
11    public int getB() {return b;}
12    public int getC() {return c;}
13    public int getD() {return d;}
14
15 //implementation of abstract methods:
16   @Override public double perimeter() {
17    return a.dist(b)+b.dist(c)+c.dist(d)+d.dist(a);}
18   @Override public String getName() { return this.getClass().toString()
19 //overwrite the method inherited from Object :
20   @Override public String toString()
21    {return "["+a+","+b+","+c+","+d+"]";}
22 }
```

The classes `Parallelogram`, `Rectangle` and `ShapeTest` can be reused without any change(!) from the example for abstract classes with identical result, see `C5E14_Shape_Interface`!