tutorial OO 3
Interfaces, Comparable<T>, Comparator<T>

Pol Van Aubel

Radboud University

INTERFACES

interface

- what it is
  - set of method types
  - a class implementing an interface has at least the public methods of that interface
  - one class can implement multiple interfaces
  - one interface can be implemented by many classes
- what use
  - use as type in attribute or argument definition (instead of class),
    any object implementing this interface will do
- what purpose
  - reuse of code: any object implementing the interface will do
  - store objects of mixed types in an array:
    any object implementing the interface fits

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

review questions

```
interface A {
  void m1();
}
```

```
A a;
```

Is the following legal Java?

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

```
class B implements A {
  void m1() {
    System.out.println("B.m1()");
  }
}
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
class B implements A {
  void m1() {
    System.out.println("B.m1()");
  }
}
```

must be public

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

must be public

```
class B implements A {
  void m1() {
    System.out.println("B.m1()");
  }
}
```

```
A a = new B();
```

review questions

```
interface A {
  void m1();
}
```

Is the following legal Java?

```
A a;
```

```
A a = new A();
```

no instances of an interface !

```
public A m2(A a)
{
  a.m1();
  return a;
}
```

must be public

```
class B implements A {
  void m1() {
    System.out.println("B.m1()");
  }
}
```

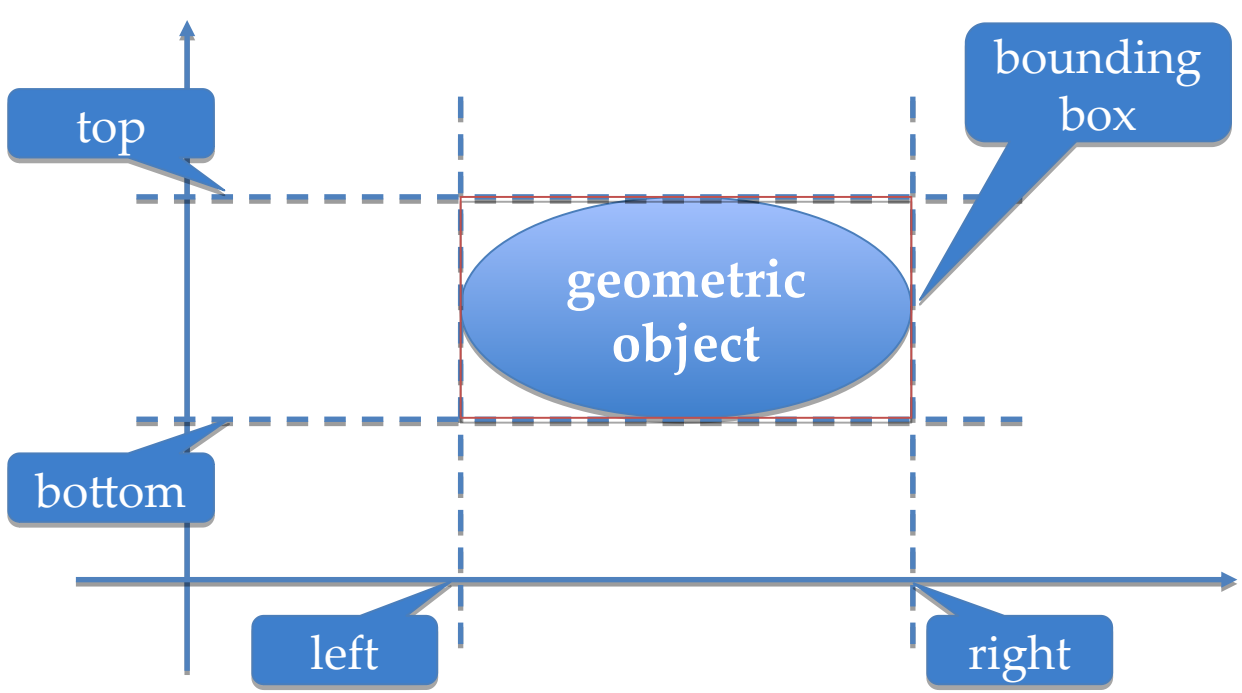
```
A a = new B();
```

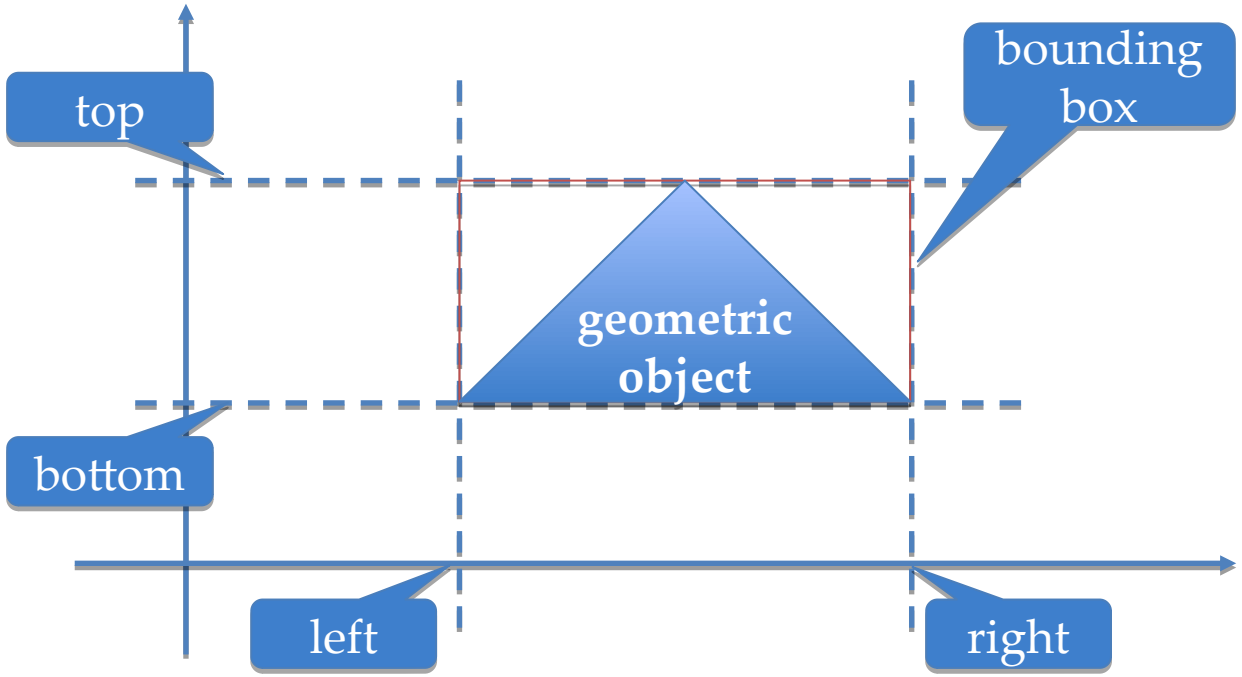
which methods in the interface

- rule of thumb: **all methods every instance should have**
  - not more
  - not fewer
- use different interfaces for unrelated sets of methods
  - unrelated methods would become coupled by grouping them in an single interface
  - each class can implement any number of interfaces
    there is no need to limit the number of interfaces
- interface design
  - just like a class definition: look at the required manipulations of objects (verbs)
  - now we look for manipulations of all objects implementing the interface
    not restricted to a single class

geometric objects

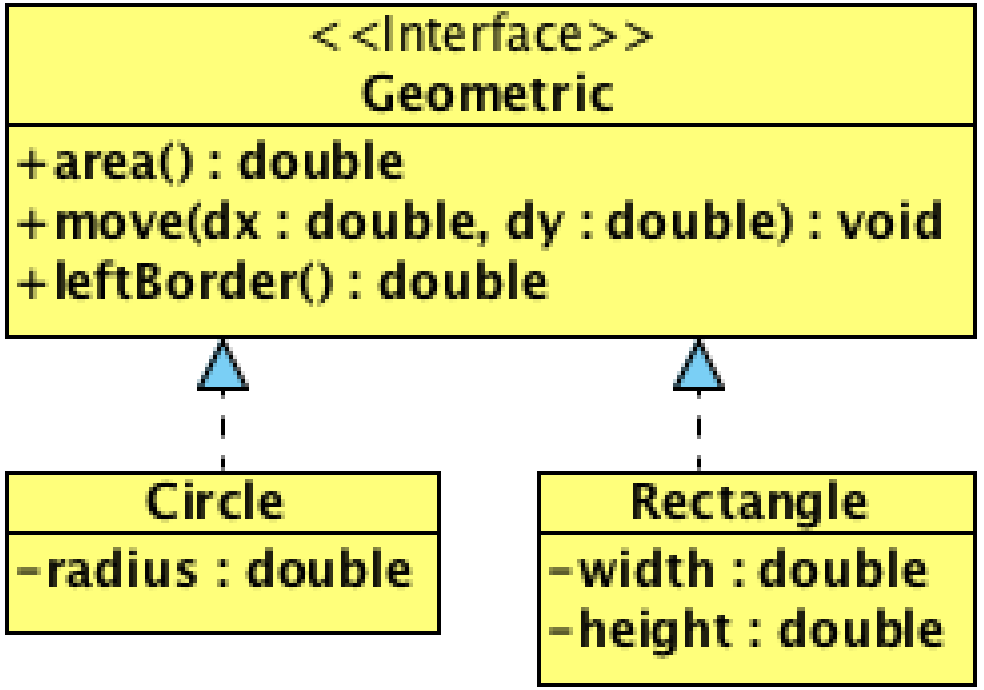
- circle, rectangle, ellipse, triangle, ..





geometric objects in Java

- only consider circles and rectangles in this assignment
- KISS (Keep It Safe & Stupid) [US Navy, 1960]



interface definitions

- list the type, name, and arguments of the methods to be implemented

```
public interface Geometric {
  public double area();
  public void move(double x, double y);
  ..
```

in file Geometric.java

- here **public** is the default visibility and can be omitted
  - but I don't think you should: "explicit is better than implicit" – Tim Peters, the Zen of Python

```
public interface Geometric {
  double area();
  void move(double x, double y);
```

allowed because of default methods, but discouraged!

  - is there any use for private methods in an interface?
  - is there any use for a constructor in an interface?

not allowed in Java 13

static and default methods

- since Java 8 interfaces can have method implementations
- there are no attributes: methods cannot use them

```
static double minimumArea() {
  return 0.0;
}

default boolean smallerEq(Geometric g) {
  return getArea() <= g.getArea();
}

default double sizeX() {
  return rightBorder() - leftBorder();
}
```

use keyword `default`

all necessary information is in the arguments

only use other methods to access information

redefining methods

- default methods can be redefined (overridden) in an implementing class whenever the default does not apply
- use this only for exceptional cases
  - reconsider your decision to provide a default implementation of the method when this happens (often)

```
public class Point implements Geometric {

  @Override
  public boolean smallerEq(Geometric g) {
    return true;
  }
```

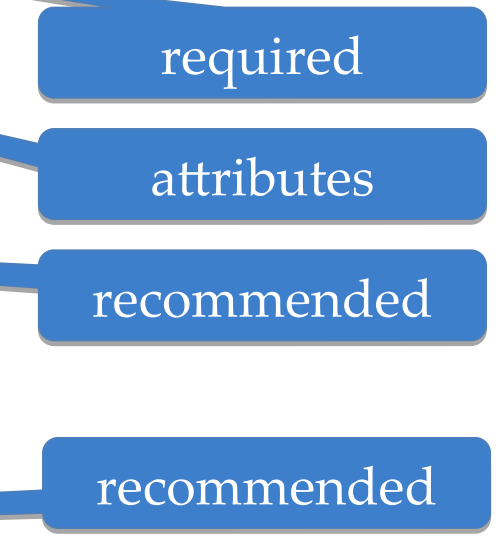
implementing interfaces

- indicate that you will define the methods and do it

```
public class Circle implements Geometric {
  private final double r;
  ..

  @Override
  public double getArea() {
    return ...
  }

  @Override
  public void move(double dx, double dy) {
    ...
  }
```



implementing multiple interfaces

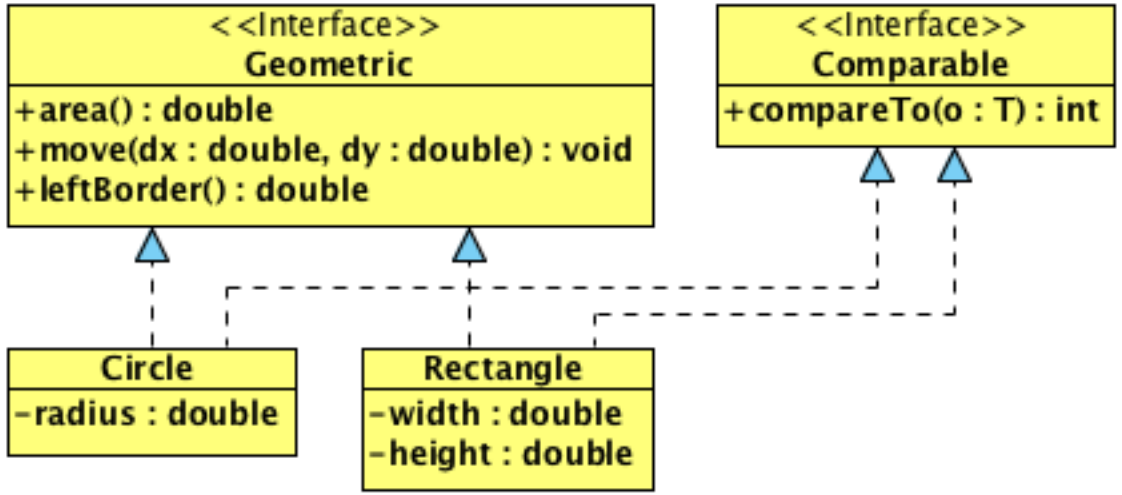
- a class can implement any number of interfaces

do **not** *define* this interface, use it

```
public class Circle implements Geometric, Comparable<Circle> {
  private final double r;
  ..
```

more general `Comparable<Geometric>`

- **all** methods of **all** interfaces must be implemented
  - except default and static methods



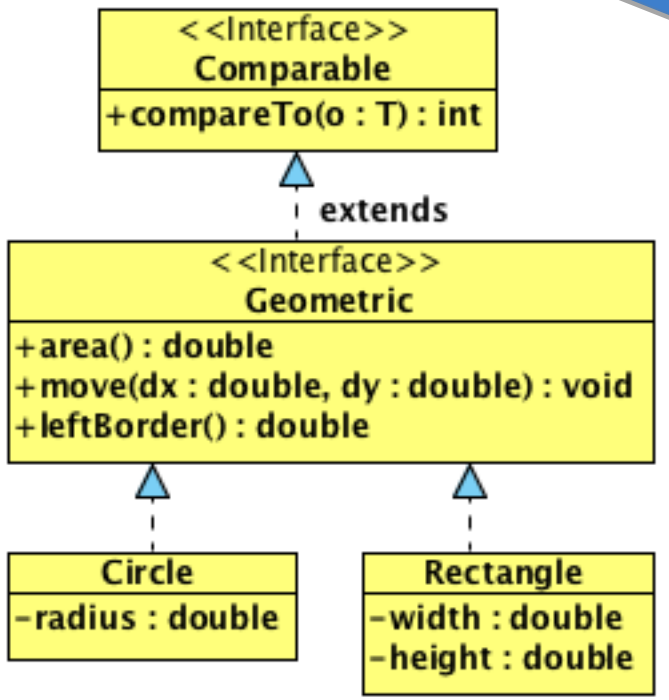
extending interfaces

- interface contains everything from the parent (extended) interface as well as the listed new methods

```
public interface Geometric extends Comparable<Geometric> {
  double area();
  void move(double x, double y);
  double leftBorder();
}
```

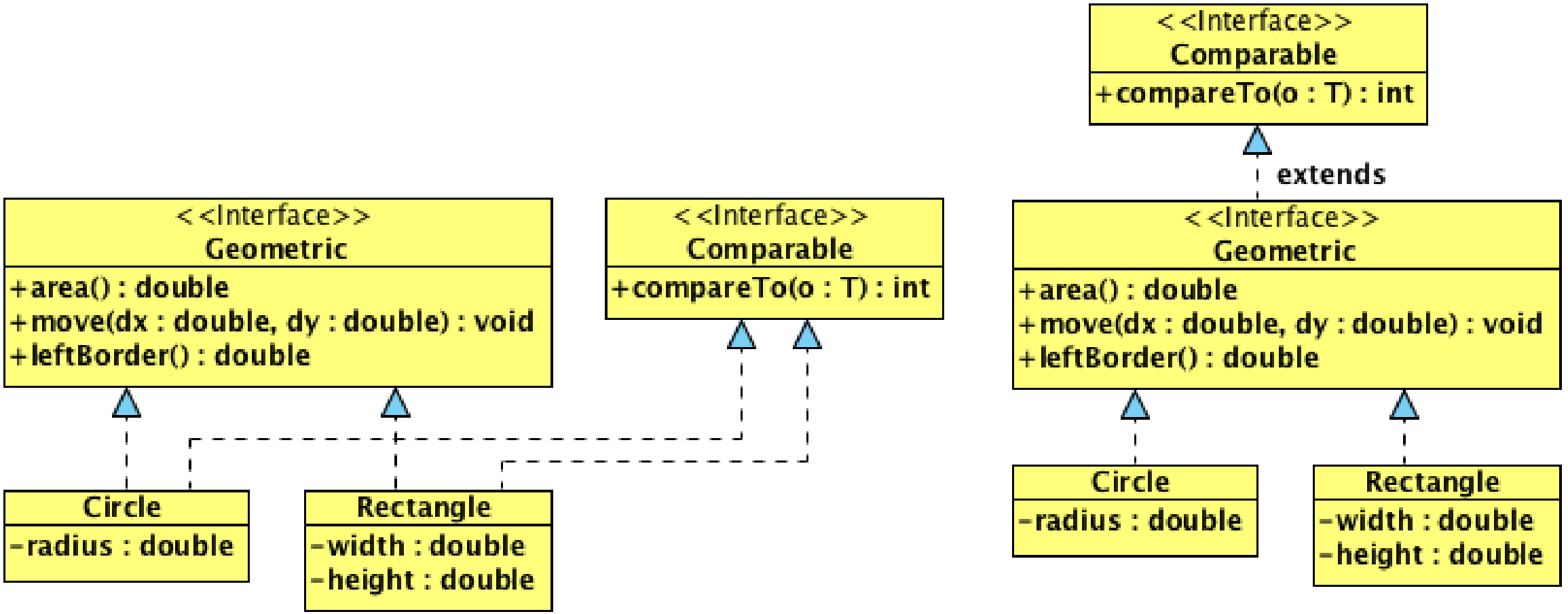
do **not** define this interface, use it

must have methods
compareTo
area
move
leftBorder



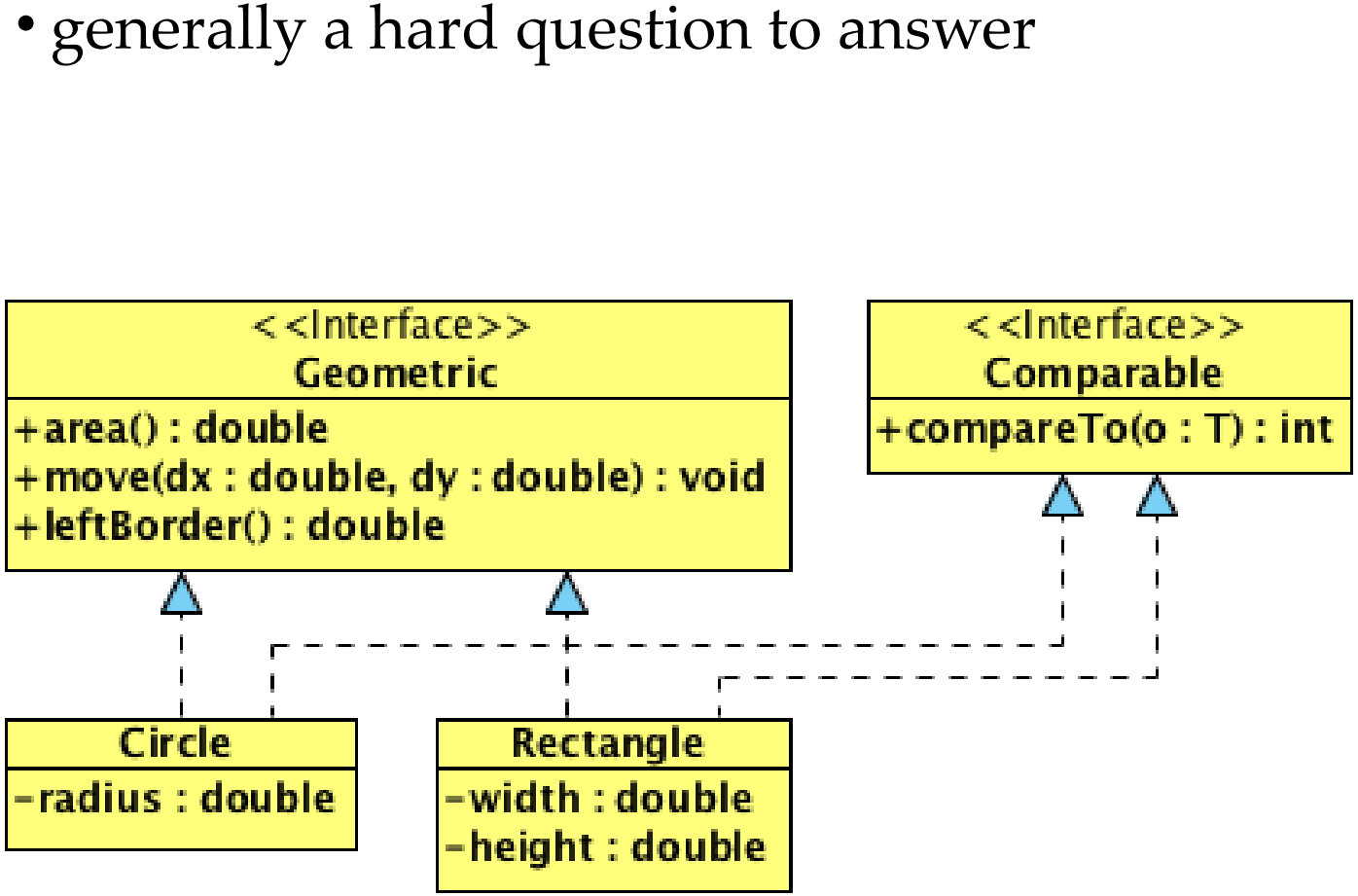
design decision

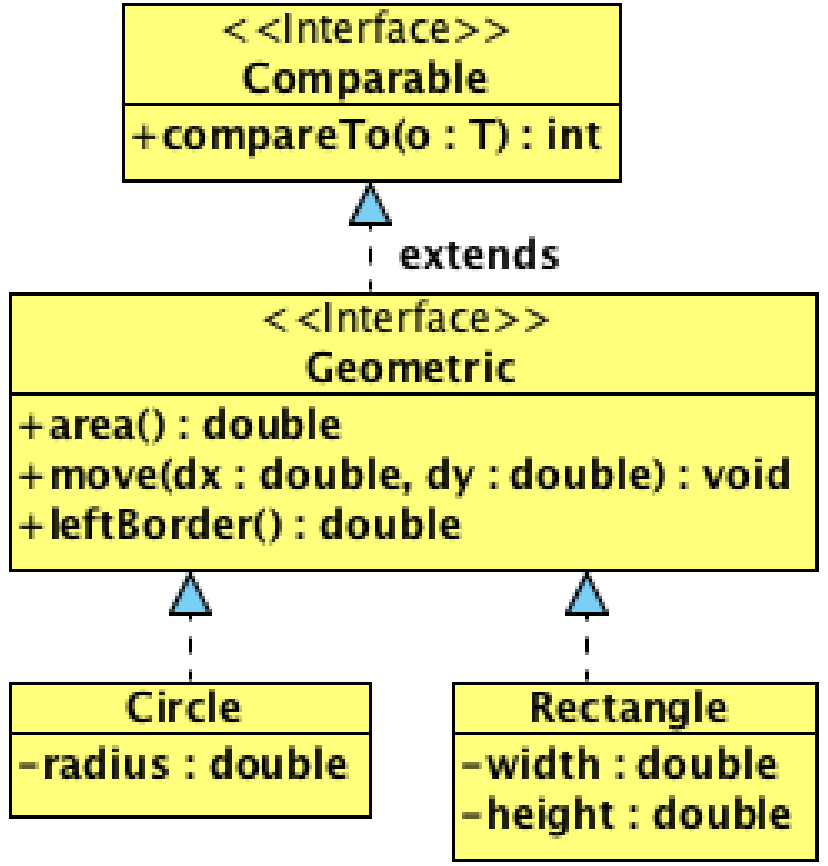
- which design is better?



design decision

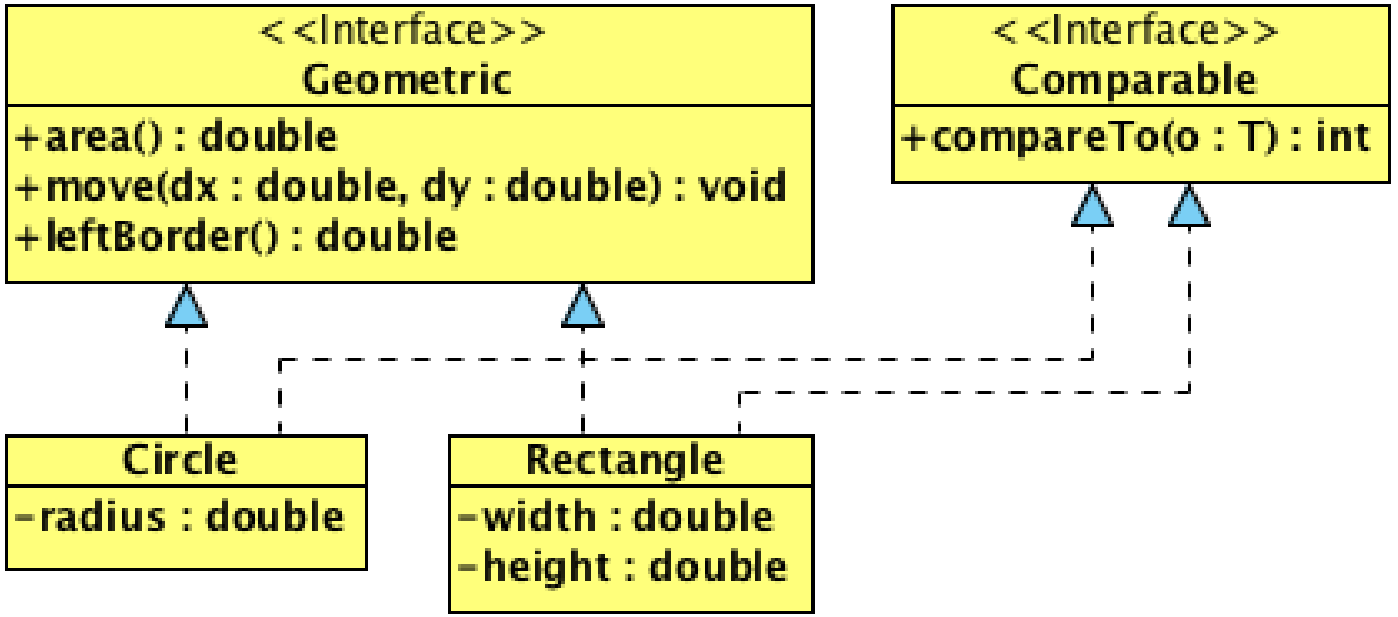
- which design is better?
- generally a hard question to answer

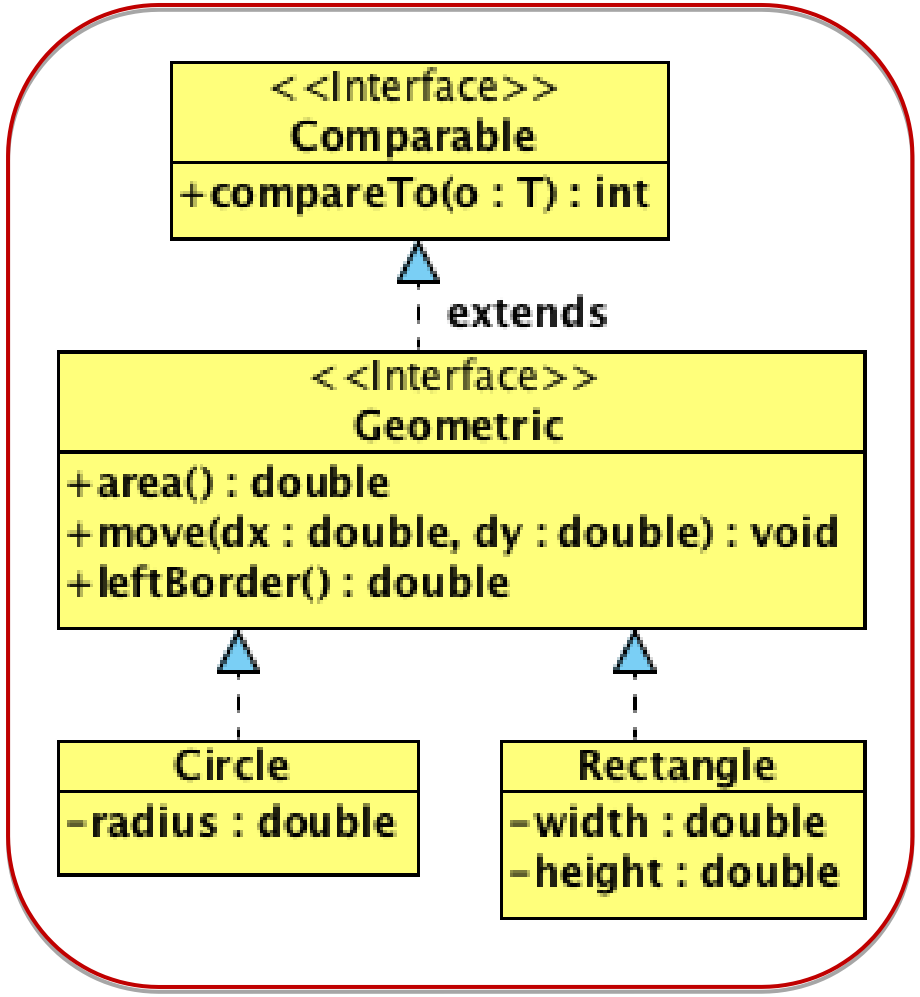




design decision

- which design is better?
- generally a hard question to answer
- in the case of Comparable, however:





extending interfaces vs. implementing multiple interfaces

- Interface: all methods **every** class **implementing that interface** should have
- sometimes this means extending an interface **from the standard library**
- `compareTo` is broadly applicable, `toString` is another example
- however, if you find yourself **extending** your **own** interfaces, **rethink your design**
  - not impossible, but more often than not a sign of bad design
- use **different interfaces** for **unrelated sets** of methods

SORTING

sorting arrays

- many programs sort arrays: how can we make a **reusable** `sort` method?
- *question*: how does `sort` compare objects?
- *answer*: by using the `compareTo` method from the `Comparable` interface
- *hence*: ensure that array element type implements the `Comparable` interface

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

- desired results:
  - x < y: `x.compareTo(y)` < 0
  - `x.equals(y)`: `x.comapareTo(y)` == 0
  - x > y: `x.compareTo(y)` > 0

T: type variable; this will be explained in week 5

Java is designed such that every class has an equals, but compareTo is optional

https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html

comparable example

```
public class Person implements Comparable<Person> {
  private String name;
  private int ssn;
  public Person(String name, int ssn) {
    this.name = name;
    this.ssn = ssn;
  }
  public int getSsn() {
    return ssn;
  }
  @Override
  public String toString() {
    return name + " (" + ssn + ")";
  }
  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
}
```

interface needed for sorting

implementation of the interface

application

```
void run() {
  Person [] persons =
    {new Person("Alice", 5)
    ,new Person("Dave", 7)
    ,new Person("Carol", 1)
    ,new Person("Bob", 3)
    };
  print(persons);
  Arrays.sort(persons);
  print(persons);
}

private void print(Person [] array) {
  System.out.println("Persons:");
  for (Person p: array) {
    System.out.println("  " + p);
  }
}
```

```
Persons:
  Alice (5)
  Dave (7)
  Carol (1)
  Bob (3)
Persons:
  Alice (5)
  Bob (3)
  Carol (1)
  Dave (7)
```

enhanced for-loop

ad hoc sorting

- sorting using comparable seems great:
  a single efficient sorting algorithm that works for comparable objects
- However, every now and then we need to sort in a different order
  - e.g. persons based on their ssn
- solutions:
  - change `compareTo` of Person ☹
  - new sort method for Person on ssn ☹
  - a **`Comparator`** for a reusable sort object that has the desired version of compare ☺

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

same integer results as before

https://docs.oracle.com/javase/10/docs/api/java/util/Comparator.html

comparator for Person

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

same rules for results as compareTo

```
public class PersonSsnComparator implements Comparator<Person> {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.getSsn() - p2.getSsn();
  }
}
```

application

```
void run() {
    Person [] persons =
      {new Person("Alice",5)
      ,new Person("Dave",7)
      ,new Person("Carol",1)
      ,new Person("Bob",3)
      };
  print(persons);
  PersonComperator comparator = new PersonSsnComparator();
  Arrays.sort(persons, comparator);
  print(persons);
  Arrays.sort(persons, comparator.reversed());
  print(persons);
  Arrays.sort(persons);
  print(persons);
}
```

make new comparator object

```
Persons:
  Alice (5)
  Dave (7)
  Carol (1)
  Bob (3)
Persons:
  Carol (1)
  Bob (3)
  Alice (5)
  Dave (7)
Persons:
  Dave (7)
  Alice (5)
  Bob (3)
  Carol (1)
Persons:
  Alice (5)
  Bob (3)
  Carol (1)
  Dave (7)
```

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
  public Person(int id) {
    this.id = id;
  }
```

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
  public Person(int id) {
    this.id = id;
  }
```

there is no `compareTo` method defined in `Person`

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
```

there is no compareTo method defined in Person

```
Exception in thread "main" java.lang.ClassCastException:
class scratchpad.Person cannot be cast to class java.lang.Comparable …
   at java.base/java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort
   at java.base/java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
   at java.base/java.util.Arrays.sort(Arrays.java:1250)
   at scratchpad.Scratchpad.main(Scratchpad.java:22)
```

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
  public Person(int id) {
    this.id = id;
  }
  public int compareTo(Person p) {
    return id - p.id;
  }
}
```

there is no `compareTo` method defined in `Person`

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
  public Person(int id) {
    this.id = id;
  }
  public int compareTo(Person p) {
    return id - p.id;
  }
}
```

there is no `compareTo` method defined in `Person`

there is no interface reference here; Java does not know that Persons can be compared

correct?

```
public class Test {
  public static void main(String[] args) {
    Person [] a = {new Person(1), new Person(2)};
    Arrays.sort(a);
  }
}
public class Person {
  private int id;
  public Person(int id) {
    this.id = id;
  }
```

there is no `compareTo` method defined in `Person`

there is no interface reference here; Java does not know that Persons can be compared

```
Exception in thread "main" java.lang.ClassCastException:
class scratchpad.Person cannot be cast to class java.lang.Comparable …
   at java.base/java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort
   at java.base/java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
   at java.base/java.util.Arrays.sort(Arrays.java:1250)
   at scratchpad.Scratchpad.main(Scratchpad.java:22)
```

another example

ANIMALS

the interface

- define a new interface `Animal`
- make sure there is a `compareTo`
  - provide a default implementation of this method

```
public interface Animal extends Comparable<Animal> {
  String getName();
  int legs();
  default int compareTo(Animal a) {
    return toString().compareTo(a.toString());
  }
}
```

my design decision to implement this method here, this is **not** required

why decide to implement as `default` in the interface?

- Interface: all methods **every** class **implementing that interface** should have
- `compareTo` is for sorting, and meaningful sorting needs ***transitivity:***
  - `if x ≤ y and y ≤ z then x ≤ z`
  - if all classes implement their own sort for the interface, what can go wrong?
  - `Frog < Fish; Fish < Duck; Duck < Frog`
    - `Arrays.sort( [Fish, Duck, Frog] );`
  - `Circle` could sort on the y-coordinate and `Rectangle` on the x-coordinate
    - `Arrays.sort( [Circle(5,10), Rectangle(10,5)] );`

- Conclusion: They'd all need the exact same implementation – DRY principle

some types for animals

```
public class Fish implements Animal {
  private String name;
  public Fish(String name) {
    this.name = name;
  }
  @Override
  public String getName() {
    return name;
  }
  @Override
  public int legs() {
    return 0;
  }
  @Override
  public String toString() {
    return "A fish called " + name;
  }
}
```

```
public class Frog implements Animal {
  private String name;
  public Frog(String name) {
    this.name = name;
  }
  @Override
  public String getName() {
    return name;
  }
  @Override
  public int legs() {
    return 4;
  }
  @Override
  public String toString() {
    return "Frog(" + name + ')';
  }
}
```

ad-hoc sorting of animals based on leg count

```
public class LegComparator implements Comparator<Animal> {

  @Override
  public int compare(Animal o1, Animal o2) {
    int d = o2.legs() – o1.legs();
    return d == 0 ? o2.getName().compareTo(o1.getName()) : d;
  }
}
```

more legs is smaller !

by construction inverse sorting

conditional expression; equivalent to

```
if (d == 0) {
  return o2.getName().compareTo(o1.getName());
} else {
  return d;
}
```

private helper method used

```
private void print(Animal[] animals) {
  StringBuilder sb = new StringBuilder();
  for (Animal a: animals) {
    sb.append(a).append('\n');
  }
  System.out.println(sb);
}
```

- a slightly nonstandard way of printing an array of animals

constructing a single string and printing it might be slightly (or a lot) more efficient than printing all animals individually

using some animals

```
private void run() {
  Animal animals [] =
      { new Frog("Kermit")
      , new Sapiens("Sjaak")
      , new Fish("Nemo")
      , new Sapiens("Pol")
      , new Fish("Wanda")
      , new Duck("Donald")
      , new Frog("Robin")
      };
  Arrays.sort(animals);
  print(animals);
  Arrays.sort(animals, new LegComparator());
  print(animals);
}
```

```
A fish called Nemo
A fish called Wanda
Duck: Donald
Frog(Kermit)
Frog(Robin)
Human Pol
Human Sjaak

Frog(Robin)
Frog(Kermit)
Human Sjaak
Human Pol
Duck: Donald
A fish called Wanda
A fish called Nemo
```

so if `Comparator` is so great, why does `Comparable` exist?

- honestly, `Comparable<T>` is conceptually not a very good interface
- makes the most sense for types that have a sensible default sort
  - e.g. the natural numbers
  - e.g. lexical ordering of Strings (and even this is arguable)
  - but for `Geometric`, why is the default the x-coordinate?
  - why do Animals sort based on their string description?

- personally I'd not define default sorts for most types, but rather implement `Comparators` for every sort

should we use `default` methods?

- I learnt Java without them and turned out okay
- they are *implicit implementation inheritance* which can be problematic
  - (see, for instance, discussions about inheritance vs. inclusion / mixins)
- the rationale[1] for their introduction was actually *backwards compatibility* when an existing interface is *changed*
- for every `default` method there's an equally powerful, more explicit application of the Strategy pattern
- they *can* be used productively, e.g. the functional composition of `Comparator`
- so use them when you think is appropriate, but realize that there are other, possibly better ways of doing things
- choosing the right way is something that only real-world experience will teach

[1] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

finally

questions?

ready to make the assignment?

tip: use NetBeans to generate a skeleton for all methods of an interface

IDE DEMO: REFACTORING & DEBUGGING