Building Java Programs

Interfaces and Comparable

reading: 9.5 - 9.6, 10.2, 16.4

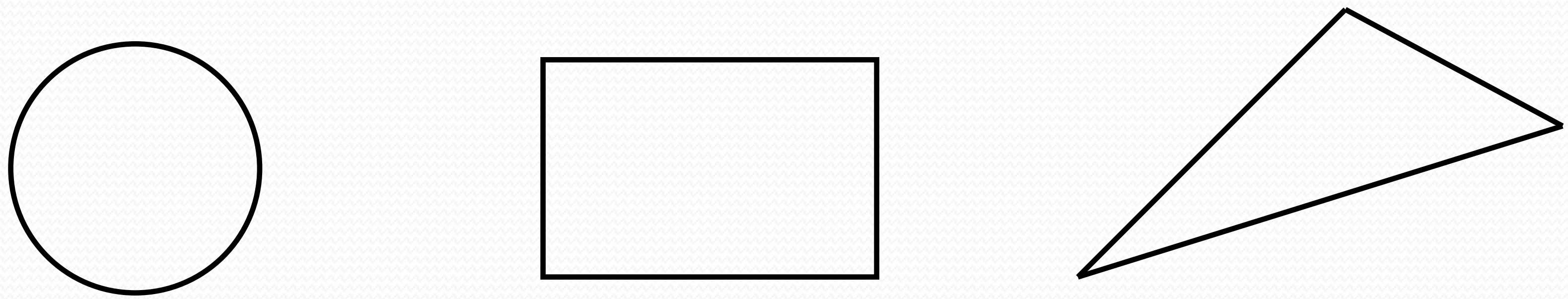

露天風呂
ゆ

Shapes

- Consider the task of writing classes to represent 2D shapes such as `Circle`, `Rectangle`, and `Triangle`.

- Certain operations are common to all shapes:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape

  - Every shape has these, but each computes them differently.

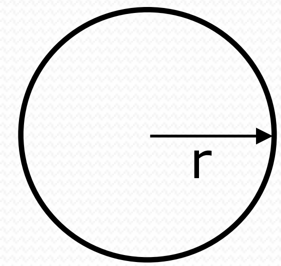


Shape area and perimeter

- Circle (as defined by radius r):

  area $= \frac{1}{2} \pi r^2$

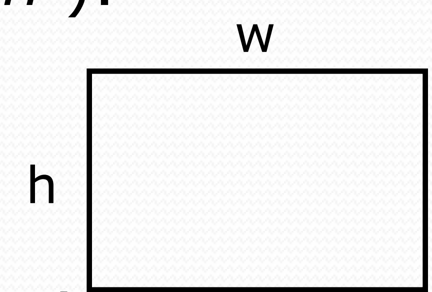
  perimeter $= 2 \pi r$



- Rectangle (as defined by width w and height h):

  area $= w\,h$

  perimeter $= 2w + 2h$

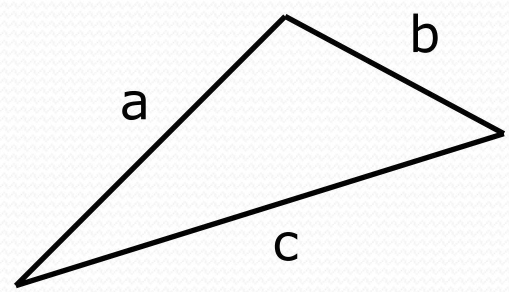


- Triangle (as defined by side lengths a, b, and c)

  area $= \sqrt{(s\,(s - a)\,(s - b)\,(s - c))}$

  where $s = \frac{1}{2}(a + b + c)$

  perimeter $= a + b + c$



Common behavior

- Suppose we have 3 classes `Circle`, `Rectangle`, `Triangle`.
  - Each has the methods `perimeter` and `area`.

- We'd like our client code to be able to treat different kinds of shapes in the same way:
  - Write a method that prints any shape's area and perimeter.
  - Create an array to hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
  - Make a `DrawingPanel` display many shapes on screen.

Interfaces (9.5)

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

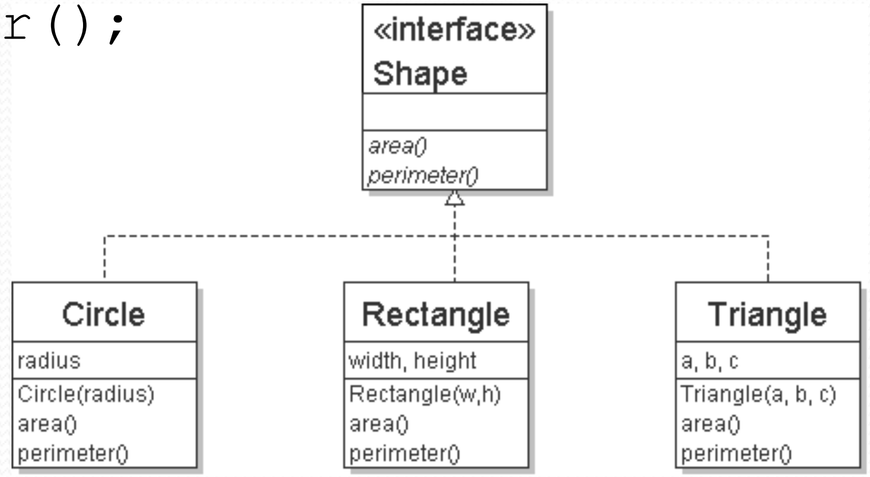
Example:

```
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

Shape interface

```
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

- Saved as `Shape.java`



- **abstract method**: A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

Implementing an interface

```
public class name implements interface {
    ...
}
```

- A class can declare that it "implements" an interface.
  - The class must contain each method in that interface.

```
public class Bicycle implements Vehicle {
    ...
}
```

(Otherwise it will fail to compile.)

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

Interface requirements

```
public class Banana implements Shape {
    // haha, no methods! pwned
}
```

- If we write a class that claims to be a `Shape` but doesn't implement `area` and `perimeter` methods, it will not compile.

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

Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
  - They allow **polymorphism**.
    (the same code can work with different types of objects)

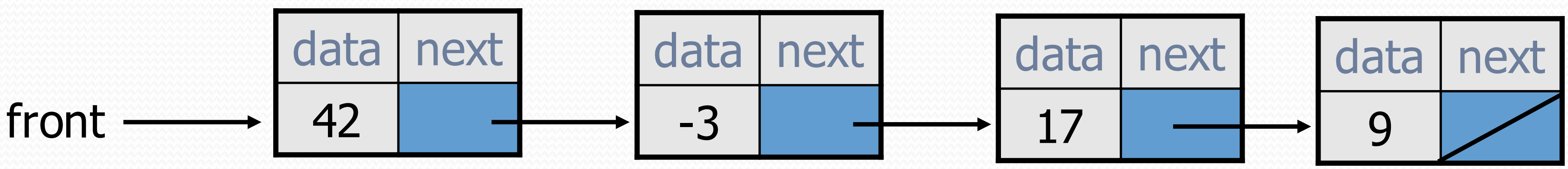
```
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

Linked vs. array lists

- We have implemented two collection classes:
  - `ArrayIntList`

index	0	1	2	3
value	42	-3	17	9

  - `LinkedIntList`



  - They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.

Redundant client code

```
public class ListClient {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        list1.add(18);
        list1.add(27);
        list1.add(93);
        System.out.println(list1);
        list1.remove(1);
        System.out.println(list1);

        LinkedIntList list2 = new LinkedIntList();
        list2.add(18);
        list2.add(27);
        list2.add(93);
        System.out.println(list2);
        list2.remove(1);
        System.out.println(list2);
    }
}
```

An `IntList` interface

```java
// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public int remove(int index);
    public void set(int index, int value);
    public int size();
}
```

```java
public class ArrayIntList implements IntList { ...

public class LinkedIntList implements IntList { ...
```

Client code w/ interface

```
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);

        IntList list2 = new LinkedIntList();
        process(list2);
    }

    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
```

ADTs as interfaces (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`
    - They messed up on `Stack`; there's no `Stack` interface, just a class.

Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:

```
public void stutter(List<String> list) {
    ...
}
```

Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?

- Answer: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end; `LinkedList` is faster for adding/removing at the front/middle. Etc.

  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

The Comparable Interface

reading: 10.2

Binary search and objects

- Can we `binarySearch` an array of Strings?
  - Operators like `<` and `>` do not work with `String` objects.
  - But we do think of strings as having an alphabetical ordering.

- **natural ordering**: Rules governing the relative placement of all values of a given type.

- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
  - A < B,   A == B,   A > B

`Collections` class

Method name	Description
`binarySearch(`**list**`, `**value**`)`	returns the index of the given value in a sorted list (< 0 if not found)
`copy(`**listTo**`, `**listFrom**`)`	copies **listFrom**'s elements to **listTo**
`emptyList(),emptyMap(), emptySet()`	returns a read-only collection of the given type that has no elements
`fill(`**list**`, `**value**`)`	sets every element in the list to have the given value
`max(`**collection**`),min(`**collection**`)`	returns largest/smallest element
`replaceAll(`**list**`, `**old**`, `**new**`)`	replaces an element value with another
`reverse(`**list**`)`	reverses the order of a list's elements
`shuffle(`**list**`)`	arranges elements into a random order
`sort(`**list**`)`	arranges elements into ascending order

The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of **A**.`compareTo(`**B**`)` will return:

a value < 0	if **A** comes "before" **B** in the ordering,
a value > 0	if **A** comes "after" **B** in the ordering,
0	if **A** and **B** are considered "equal" in the ordering.

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

Binary search w/ strings

```
// Returns the index of an occurrence of target in a,
// or a negative number if the target is not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(String[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid].compareTo(target) < 0) {
            min = mid + 1;
        } else if (a[mid].compareTo(target) > 0) {
            max = mid - 1;
        } else {
            return mid;    // target found
        }
    }

    return -(min + 1);    // target not found
}
```

`compareTo` and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet`/`Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Ordering our own types

- We cannot binary search or make a `TreeSet`/`Map` of arbitrary types, because Java doesn't know how to order the elements.
  - The program compiles but crashes when we run it.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
...

Exception in thread "main"
  java.lang.ClassCastException
        at java.util.TreeSet.add(TreeSet.java:238)
```

Interfaces (9.5)

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

`Comparable` (10.2)

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call to your `compareTo` method should return:

a value < 0	if the `this` object comes "before" `other` one,
a value > 0	if the `this` object comes "after" `other` one,
0	if the `this` object is considered "equal" to `other`.

`Comparable` template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;   // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

`compareTo` tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;    // different x
    } else {
        return y - other.y;    // same x; compare y
    }
}
```

  - The idea:
    - if `x > other.x`, then `x - other.x > 0`
    - if `x < other.x`, then `x - other.x < 0`
    - if `x == other.x`, then `x - other.x == 0`
      - NOTE: This trick doesn't work for `double`s (but see `Math.signum`)

`compareTo` tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *`toString` trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

Exercises

- Make the `HtmlTag` class from HTML Validator comparable.
  - Compare tags by their elements, alphabetically by name.
  - For the same element, opening tags come before closing tags.

```
// <body><b></b><i><b></b><br/></i></body>
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));   // <body>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("i", true));      // <i>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("br"));           // <br />
tags.add(new HtmlTag("i", false));     // </i>
tags.add(new HtmlTag("body", false));  // </body>
System.out.println(tags);
// [<b>, </b>, <body>, </body>, <br />, <i>, </i>]
```

Exercise solution

```
public class HtmlTag implements Comparable<HtmlTag> {
    ...
    // Compares tags by their element ("body" before "head"),
    // breaking ties with opening tags before closing tags.
    // Returns < 0 for less, 0 for equal, > 0 for greater.
    public int compareTo(HtmlTag other) {
        int compare = element.compareTo(other.getElement());
        if (compare != 0) {
            // different tags; use String's compareTo result
            return compare;
        } else {
            // same tag
            if ((isOpenTag == other.isOpenTag()) {
                return 0;   // exactly the same kind of tag
            } else if (other.isOpenTag()) {
                return 1;   // he=open, I=close; I am after
            } else {
                return -1;  // I=open, he=close; I am before
            }
        }
    }
}
```