CSE 143
Lecture 6

Interfaces; Complexity (Big-Oh)

reading: 9.5, 11.1, 13.1 - 13.3

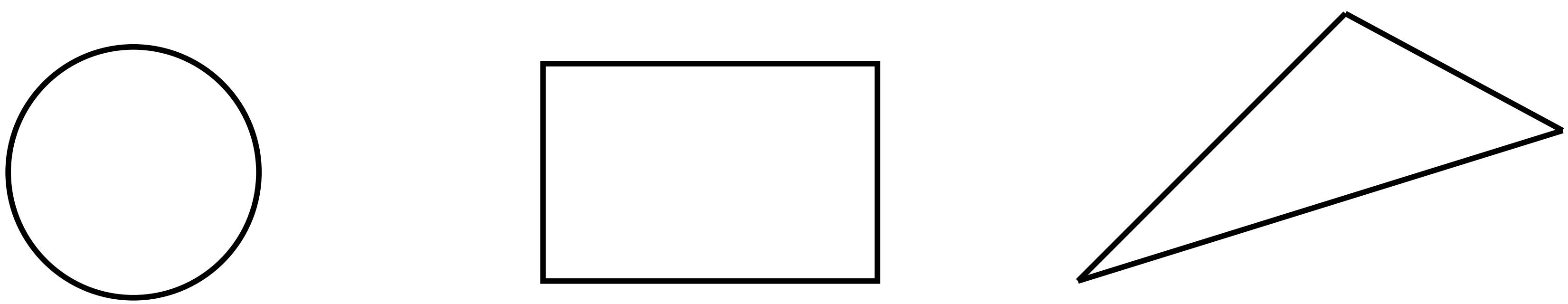
slides created by Marty Stepp
http://www.cs.washington.edu/143/

Related classes

- 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.

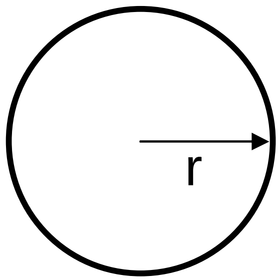


Shape area and perimeter

- Circle (as defined by radius r):

  area = πr^2

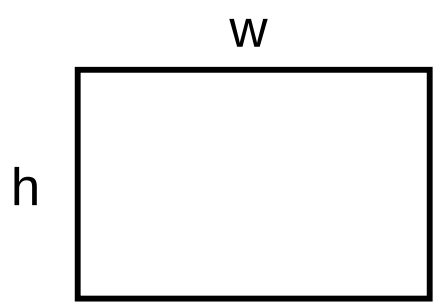
  perimeter = $2 \pi r$



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

  area = $w h$

  perimeter = $2w + 2h$

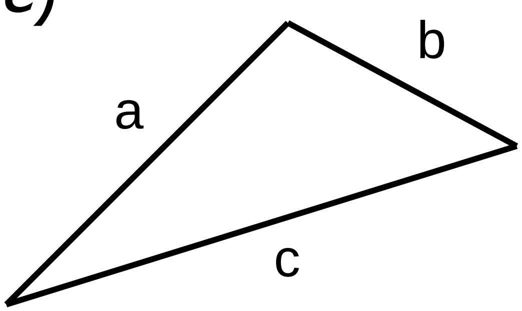


- Triangle (as defined by side lengths a, b, and c)

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

  where $s = ½ (a + b + c)$

  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 classes can promise to implement.
  - Inheritance gives you an is-a relationship and code sharing.
    - A `Lawyer` object can be treated as an `Employee`, and `Lawyer` inherits `Employee`'s 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. The certification assures you that I know how to do taxes, perform audits, and do consulting."
    - "I'm a Shape. 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);
}
```

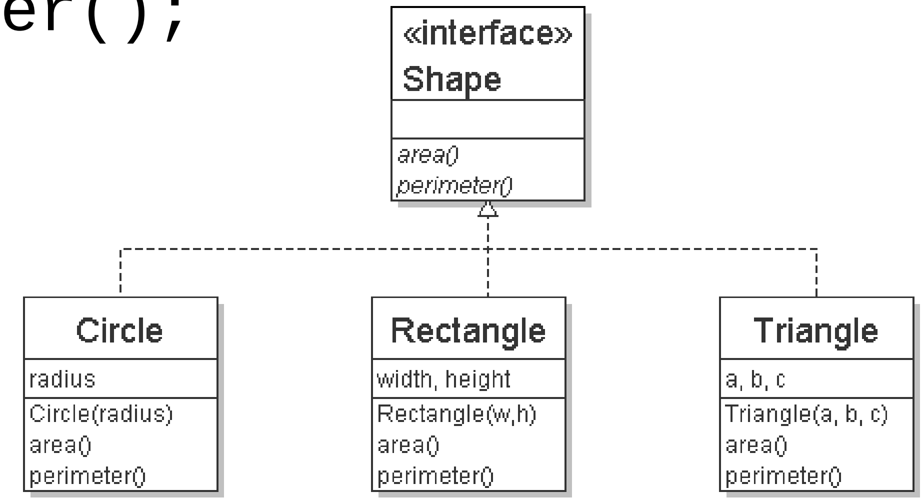
Example:

```
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
```

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 promises to contain each method in that interface.
    (Otherwise it will fail to compile.)

  - Example:

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

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.
  - They allow client code to take advantage of **polymorphism** (the same code is able to 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();
}
```

  - Any shape can be passed as the parameter to the method.

```
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

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 describes ADTs with interfaces:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`, `SortedMap`

- 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

- 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.
  - `HashSet` can search a huge data set for a value in short time; `TreeSet` is slower but keeps the set of data in a sorted order.
  - 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.

Algorithm growth rates (13.2)

- We measure runtime efficiency not in seconds, but in proportion to the input data size N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs $\mathbf{0.4N^3 + 25N^2 + 2N + 17}$ statements.
  - Consider the runtime when N is extremely large.
  - We ignore constants like 25 because they are tiny next to N.
  - We only look at the highest-order term (N^3) because it dominates.

  - We say that this algorithm runs "on the order of" N^3.
  - or $\mathbf{O(N^3)}$ for short ("**Big-Oh** of N cubed")

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

Collection efficiency

- Efficiency of various operations on different collections:

Method	ArrayList	SortedIntList	Stack	Queue
`add` (or push)	O(1)	O(N)	O(1)	O(1)
`add(`**`index`**`, `**`value`**`)`	O(N)		-	-
`indexOf`	O(N)	O(?)	-	-
`get`	O(1)	O(1)	-	-
`remove`	O(N)	O(N)	O(1)	O(1)
`set`	O(1)	O(1)	-	-
`size`	O(1)	O(1)	O(1)	O(1)

Binary search (13.1, 13.3)

- **binary search**: An algorithm that searches a sorted array or list by successively eliminating half of the elements.
  - Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.
  - Which indexes does the algorithm examine to find value **22**?
  - What is the runtime complexity class of binary search?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
value	-4	-1	0	2	3	5	6	8	11	14	22	29	31	37	56