administrivia...
- assignment 1 due on Thursday at midnight

- clickers start on Thursday
last time...
disclaimer: this class is *not* about teaching you Java
disclaimer: this class is *not* about teaching you Java

**disclaimer:** I am *not* a Java expert...
**disclaimer:** this class is *not* about teaching you Java

**disclaimer:** I am *not* a Java expert…

…but the TA’s are!
- A variable is a piece of data in memory with:
  - an identifier (name)
  - a type

- What is a type?
  - A basic building block in a programming language
  - Determines what kind of data a variable holds, and what operations can be performed on it

- Java defines eight primitive types
  - byte, short, int, long, float, double, char, boolean
  - Each primitive type can hold a single value
    - 'r', 12, 2.64, true
type conversion

- widening conversions

- narrowing conversions
type conversion

-widening conversions
   short  ->  int
   int    ->  long
   int    ->  float

-narrowing conversions
type conversion

- widening conversions
  short  -> int
  int    -> long
  int    -> float

- narrowing conversions
  double -> float
  float  -> int
type conversion

- **widening conversions**
  - short → int
  - int → long
  - int → float

- **narrowing conversions**
  - double → float
  - float → int

5 / 2 * 3.0 + 10 / 3
type conversion

-widening conversions
  short  ->  int
  int    ->  long
  int    ->  float

-narrowing conversions
  double  ->  float
  float  ->  int

5 / 2 * 3.0 + 10 / 3  =  9.0
type conversion

- **widening conversions**
  - short  $\rightarrow$ int
  - int    $\rightarrow$ long
  - int    $\rightarrow$ float

- **narrowing conversions**
  - double $\rightarrow$ float
  - float  $\rightarrow$ int

$5 \div 2 \times 3.0 + 10 \div 3 = 9.0$

"6+3=“ + 6 + 3
type conversion

-widening conversions
  short  ->  int
  int    ->  long
  int    ->  float

-narrowing conversions
  double ->  float
  float  ->  int

5 / 2 * 3.0 + 10 / 3  9.0

“6+3=” + 6 + 3  “6+3=63”
Control flow determines how programs make decisions about what to do, and how many times to do it:

- Decision making: if-else, switch-case
- Looping: for, while, do-while
- Jumping: break, continue, return
- Exceptions: try-catch, throw
-All non-primitive types are **reference types**

-a reference is a variable that stores the memory address where an object (a group of values) resides

Point p1, p2, p3;
p1 = new Point(7,19);
p2 = p1;
today...
- inheritance
- polymorphism
- abstract classes
- interfaces
object-oriented programming

-data is treated as encapsulated in objects
  -objects contain data and define functions meaningful to that data
object-oriented programming

- data is treated as encapsulated in objects
  - objects contain data and define functions meaningful to that data

- objects are instantiations of classes
  - actual written piece of code which is used to define the behavior of any given class
object-oriented programming

- data is treated as encapsulated in objects
  - objects contain data and define functions meaningful to that data

- objects are instantiations of classes
  - actual written piece of code which is used to define the behavior of any given class

A class is a general concept, while an object is a very specific embodiment of that class
object-oriented programming

- data is treated as encapsulated in objects
  - objects contain data and define functions meaningful to that data

- objects are instantiations of classes
  - actual written piece of code which is used to define the behavior of any given class

a class is a general concept, while an object is a very specific embodiment of that class

- OOP supports and enables…
  - modularity
  - code re-use
  - better code design
  - …
- inheritance is one of the most powerful features of OOP
- allows a class to inherent properties from another class
- used when multiple types of data have something in common
- avoid duplication of code
example...
shape class

-a shape has (fields):
  -a color (String)
  -an area (double)

-different shapes:
  -circle
  -triangle
  -rectangle
  -square
public class Triangle{
    String color;
    double area;
}

public class Circle{
    String color;
    double area;
}

public class Rectangle{
    String color;
    double area;
}

public class Square{
    String color;
    double area;
}
public class Triangle{
    String color;
    double area;
}

public class Circle{
    String color;
    double area;
}

public class Rectangle{
    String color;
    double area;
}

public class Square{
    String color;
    double area;
}
public class Triangle{
    String color;
    double area;
}

public class Circle{
    String color;
    double area;
}

public class Rectangle{
    String color;
    double area;
}

public class Square{
    String color;
    double area;
}

what if I want to redefine color as an integer array (R,G,B)?

What if I want to give each shape an outline color?
public class Triangle{
    String color;
    double area;
}

public class Circle{
    String color;
    double area;
}

public class Rectangle{
    String color;
    double area;
}

public class Square{
    String color;
    double area;
}
public class Triangle{
    String color;
    double area;
}

public class Circle{
    String color;
    double area;
}

public class Rectangle{
    String color;
    double area;
}

public class Square{
    String color;
    double area;
}

what if I want to redefine color as an integer array \(R,G,B\)?

What if I want to give each shape an outline color?

what can I do?

extends
public class Shape{
    String color;
    double area;
}

public class Triangle extends Shape{
}

public class Circle extends Shape{
}

public class Rectangle extends Shape{
}

public class Square extends Rectangle{
}
public class Shape{
    String color;
    double area;
}

public class Triangle extends Shape{
}

public class Circle extends Shape{
}

public class Rectangle extends Shape{
}

public class Square extends Rectangle{
}

inherit all public fields and methods of Shape
public class Shape{
    String color;
    double area;
}

called a **base class**
(or superclass)

public class Triangle extends Shape{
}

public class Circle extends Shape{
}

public class Rectangle extends Shape{
}

public class Square extends Rectangle{
}

inherit all public fields and methods of Shape
-now we have several shape classes, all with common fields associated with every shape

-but...
- circles have a radius
- rectangles have a width and height
- triangles have three Points
-now we have several shape classes, all with common fields associated with every shape

-but...
  -circles have a radius
  -rectangles have a width and height
  -triangles have three Points

-does it make sense for all shapes to have a radius? a width and height? three Points?
-now we have several shape classes, all with common fields associated with every shape

-but...

- circles have a radius
- rectangles have a width and height
- triangles have three Points

-does it make sense for all shapes to have a radius? a width and height? three Points?

-can inherited classes add their own fields and methods?
Shape
String color
double area

Triangle
Point p1
Point p2
Point p3

Circle
int radius
Point center

Rectangle
int width
int height

Square
if (width!=height)
error
How many fields does a triangle have?
How many fields does a triangle have?
How many fields does a square have?
inherited classes also inherit methods

```java
public class Shape{
    String color;
    double area;

    public String toString(){
        return color + " shape";
    }
}

Triangle t = new Triangle();
t.color = "red";
System.out.println(t.toString());
```
Inherited classes also inherit methods

public class Shape{
    String color;
    double area;

    public String toString(){
        return color + " shape";
    }
}

Triangle t = new Triangle();
t.color = "red";
System.out.println(t.toString());

red shape
what can’t inherited classes do?

- a **derived** class can:
  - add new fields
  - add new methods

- a **derived** class cannot:
  - remove fields
  - remove methods
  - inherit private fields
  - inherit private methods
overriding a method

-ability of a class to **override** a method allows a class to inherit from a base class whose behavior is close enough, then modify behavior as needed
  - method must have the same **signature**
    - same name, parameters, return type

```java
public class Circle extends Shape{
    int radius;
    Point center;

    // override
    public String toString(){
        return color + " circle with radius:" + radius;
    }
}
```
why override?

- there may be a method that makes sense for all shapes to have, but with drastically different implementations

```java
public double getArea(){
    ...
}
```
why override?

- there may be a method that makes sense for all shapes to have, but with drastically different implementations

```java
public double getArea()
{
    ...
}
```

Is the area computation the same for a Circle and a Square?
partial overriding

-derived classes can explicitly invoke the base class’s version of a method using super

```java
public void doSomething(){
    super.doSomething();
    // then do a little more
}
```
partial overriding

-derived classes can explicitly invoke the base class’s version of a method using super

```java
public void doSomething(){
    super.doSomething();
    // then do a little more
}
```

why would we do this?
partial overriding

-derived classes can explicitly invoke the base class’s version of a method using super

```java
public void doSomething(){
    super.doSomething();
    // then do a little more
}
```

**why would we do this?**

in case we want to do something just slightly different than the base class, but most of the code is done for us…
**option 1**
- copy/paste implementation of *Circle*, modify slightly for *Triangle*, *Rectangle*, and *Square*
  - debug same code in several places
  - extend/modify same code several times
  - no relationship between classes
    - *can’t pass a Circle to a method that expects a Shape*
option 1
- copy/paste implementation of Circle, modify slightly for Triangle, Rectangle, and Square
  - debug same code in several places
  - extend/modify same code several times
  - no relationship between classes
    - can’t pass a Circle to a method that expects a Shape

option 2
- base class Shape, others extend
  - can write one function that operates on any Shape
  - automatic code reuse through inheritance
a more interesting example...
suppose you are making a video game about skiing

```java
public class Ski{
    public void turn();
}

public class AlpineSki extends Ski{
    // override
    public void turn(){
        //how to turn on alpine skis
    }
}

public class TelemarkSki extends Ski{
    //override
    public void turn(){
        //how to turn on tele skis
    }
}
```
suppose you are making a video game about skiing

**without inheritance:**

```java
switch(skier.ski_type) {
    case ALPINE:
        turnAlpine();
        break;
    case TELEMARK:
        turnTelemark();
        break;
    ...
}
```

**with inheritance:**

```java
skier.ski.turn();
```
polymorphism
type compatibility

-a derived class is compatible with its base class

```java
public static boolean isLarger(Shape s1, Shape s2){
    return s1.getArea() > s2.getArea();
}

Triangle t = new Triangle(...);
Circle c = new Circle(...);

if (isLarger(t,c)){
    ...
}
```
type compatibility

-a derived class is compatible with its base class

```java
public static boolean isLarger(Shape s1, Shape s2) {
    return s1.getArea() > s2.getArea();
}

Triangle t = new Triangle(...);
Circle c = new Circle(...);

if (isLarger(t, c)) {
    ...
}
```

**why can I pass** `isLarger` **a Circle and a Triangle?**
**polymorphism** is a fancy word for automatically determining an object’s type at runtime

**the most specific type possible is used**

```java
Shape s1 = new Circle();
Shape s2 = new Triangle();

s1.getArea();
s2.getArea();
```
- **polymorphism** is a fancy word for automatically determining an object’s type at runtime

- the most specific type possible is used

  ```java
  Shape s1 = new Circle();
  Shape s2 = new Triangle();
  
  s1.getArea();
  s2.getArea();
  ```

  *What type is s1 treated as?*
-**polymorphism** is a fancy word for automatically determining an object’s type at runtime

-the most specific type possible is used

```java
Shape s1 = new Circle();
Shape s2 = new Triangle();

s1.getArea();
s2.getArea();
```

**what type is** s1 **treated as?**

**what type is** s2 **treated as?**
**polymorphism** is a fancy word for automatically determining an object’s type at runtime

-the most specific type possible is used

```
Shape s1 = new Circle();
Shape s2 = new Triangle();

s1.getArea();
s2.getArea();
```

what type is `s1` treated as?
what type is `s2` treated as?

-suppose Triangle does not override `toString()`

```
s2.toString();
```
- **polymorphism** is a fancy word for automatically determining an object’s type at runtime

- the most specific type possible is used

```java
Shape s1 = new Circle();
Shape s2 = new Triangle();

s1.getArea();
s2.getArea();
```

**what type is** `s1` **treated as?**

**what type is** `s2` **treated as?**

- **suppose** `Triangle` does not override `toString()`

```java
s2.toString();
```

**what type is** `s2` **treated as?**
- Java takes OOP to the extreme

- every reference type is polymorphic
  - every reference type inherits from Object

- when you write your own `toString()` or `equals(Object o)` methods, you are overriding Object’s version
- Java takes OOP to the extreme

- every reference type is polymorphic
  - every reference type inherits from Object

- when you write your own `toString()` or `equals(Object o)` methods, you are overriding Object’s version

```java
Matrix m = new Matrix(4,2);
System.out.println(m.toString());
```
- Java takes OOP to the extreme

- every reference type is polymorphic
  - every reference type inherits from `Object`

- when you write your own `toString()` or `equals(Object o)` methods, you are overriding `Object`'s version

Matrix m = new Matrix(4,2);
System.out.println(m.toString());

is polymorphism happening?
Shape shape_array = new Shape[5];
shape_array[0] = new Triangle();
shape_array[1] = new Circle();
shape_array[2] = new Rectangle();
...

//find the total area of all the shapes
int total_area = 0;
for(int i=0; i<5; i++)
    total_area += shape_array[i].getArea();
abstract classes
-we never intend for anyone to call the `Shape` class’s `getArea()` method directly
  -meant to be called from a specific shape

-we don’t have to provide an implementation in the base class if we make the method `abstract`
  `public abstract double getArea();`
  -semicolon immediately following definition!

-remove `abstract` keyword in derived class’s definition
- a class with at least one **abstract** method is an **abstract** class

- derived classes MUST implement **abstract** methods

- **abstract** classes cannot be instantiated
a class with at least one **abstract** method is an **abstract** class

derived classes **MUST** implement **abstract** methods

**abstract** classes cannot be instantiated

Shape s = new Shape();
Shape s = new Triangle();

which of these is illegal?
- A class with at least one `abstract` method is an `abstract class`.

- Derived classes **MUST** implement `abstract` methods.

- `abstract` classes **cannot be instantiated**.

  ```java
  Shape s = new Shape();
  Shape s = new Triangle();
  } which of these is illegal?
  ```

- `abstract` classes are **ONLY** designated as base classes.
interfaces
- an **interface** is the ultimate abstract class
  - every method is **abstract**
  - can contain only **public static final** fields
  - declared with the **interface** keyword instead of **class**

- derived **classes use keyword** **implements** instead of **extends**

- subclasses can implement multiple interfaces, but can only extend one **base class**
interfaces

-provide a contract that guarantees objects of a certain type can do specific things

-java.lang.Comparable interface has one method: compareTo()

classes that implement Comparable have a natural ordering

-can be sorted without knowing any details about the class (just use the compareTo() method!)
next time...
- **reading**  
  - chapters 3 & 4

- **homework**  
  - assignment 1 due Thursday at midnight  
    - *must complete on your own!*

- **clickers start on Thursday**