Generics & Comparators
administrivia...
- assignment 1 due today at midnight

- assignment 2 is out
  - due next Wednesday at midnight
  - another solo assignment
  - next assignment is a pair — find a friend this week!
last time...
inheritance
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;
}

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?
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 \(\text{R}, \text{G}, \text{B}\)?

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

what can I do?

extends
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{
}

}
public class Sport{
    public void compete();

    float world_record;
    ...
}

public class Gymnastics extends Sport{
    // override
    public void compete(){
        //how to do a killer floor routine
    }
}

public class Swimming extends Sport{
    //override
    public void compete(){
        //how to swim freestyle really fast
    }
}
without inheritance:

```java
switch(athlete.sport_type) {
    case GYMNASTICS:
        competeInFloorRoutine();
        break;
    case SWIMMING:
        competeInFreestyleRace();
        break;
    ...
}
```

with inheritance:

```java
athlete.sport.compete();
```
**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(); // what type is s1 treated as?
s2.getArea(); // what type is s2 treated as?
```

Suppose `Triangle` does not override `toString()`

```
s2.toString(); // what type is s2 treated as?
```
-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) first line
B) second line

Abstract classes are ONLY designated as base classes
-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**
today...
- generic programming
- generic placeholder
- why generics
- primitive types and generics
- generic static methods
- function objects
- collections
- iterators
generic programming
-suppose we want a data structure that just contains “things”

-we want it to:
  - automatically grow if it gets full
  - be able to remove items from it
  - be able to add items to it
suppose we want a data structure that just contains “things”

we want it to:
- automatically grow if it gets full
- be able to remove items from it
- be able to add items to it

will an array work?

Shape[] shape_array = new Shape[5];
what if we implement an ArrayList?
here’s what the code might look like:

```java
public class ArrayList {
    Shape storage[];
    int capacity, numItems;

    public void addItem(Shape item) {
        /*some code*/
    }

    public void autoGrow() {
        /*some code*/
    }
}
```
what if we implement an ArrayList?

here’s what the code might look like:

```java
public class ArrayList {
    Shape storage[];
    int capacity, numItems;

    public void addItem(Shape item) {
        /*some code*/
    }

    public void autoGrow() {
        /*some code*/
    }
}
```

what’s the problem with this?
-this is why we always see <> associated with ArrayList

ArrayList<Shape> list = new ArrayList<Shape>();

-ArrayList is a generic class — we can create any version of it that we want

-generic programming: algorithms are written in terms of types to-be-specified-later
  -algorithms instantiated when needed for specific types defined by parameters
-here’s what the code actually looks like:

```java
public class ArrayList<T> {
    T storage[];
    int capacity, numItems;

    public void add(T item) {
        ... }
}
```

-the placeholder T is replaced with the real type when you instantiate an ArrayList with <>

-T can be used as a type anywhere in ArrayList class
generic placeholder
what is the dynamic type of T?

ArrayList<Shape>
what is the dynamic type of \( T \)?

ArrayList\<Shape\>

ArrayList\<ClassThatArrayListDoesntKnowAbout\>
generic placeholder <>

**What is the dynamic type of T?**

ArrayList<Shape>
ArrayList<ClassThatArrayListDoesntKnowAbout>

-the generic placeholder type is VERY specific
  -ArrayList<Triangle> is not an ArrayList<Shape>, even though Triangle is a Shape!

-ArrayList<type> is only EXACTLY an ArrayList<type>, regardless of type’s heritage
inheritance and generics

-example:

```java
public void doStuff(ArrayList<Shape>) {...}

ArrayList<Triangle> tri_list;
ArrayList<Shape> shape_list;

doStuff(tri_list);  // ILLEGAL
doStuff(shape_list); // OK
```

-we can still add Triangles to shape_list

-restriction applies only to the generic object itself
Java has a way around the restriction: the wildcard placeholder `?`.

`<? extends Shape>` refers to `Shape` or anything that extends `Shape`.

`Shape`, `Triangle`, `Circle`, ...
- Java has a way around the restriction: the wildcard placeholder `?`.

- `<? extends Shape>` refers to `Shape` or anything that extends `Shape`.
  - `Shape`, `Triangle`, `Circle`, ...
Java has a way around the restriction: the wildcard placeholder `<?> extends Shape` refers to `Shape` or anything that extends `Shape`.

- `Shape`, `Triangle`, `Circle`, ...

**What types can be used here?**

<table>
<thead>
<tr>
<th>A) Circle</th>
<th>B) Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>C) Shape</td>
<td>D) Object</td>
</tr>
<tr>
<td>E) A &amp; B</td>
<td>F) A &amp; C</td>
</tr>
<tr>
<td>G) A &amp; C &amp; D</td>
<td></td>
</tr>
</tbody>
</table>

<? super Circle>
Java has a way around the restriction: the wildcard placeholder `?`:

- `<? extends Shape>` refers to `Shape` or anything that extends `Shape`
  - `Shape`, `Triangle`, `Circle`, ...

**what types can be used here?**

- `<? super Circle>`
- `<?>`

A) `Object`
B) `everything`
Java has a way around the restriction: the wildcard placeholder `?`.

`<? extends Shape>` refers to `Shape` or anything that extends `Shape`:

- `Shape`, `Triangle`, `Circle`, ...

What types can be used here?

`<? super Circle>`

`<?>`

Is this a good idea?

A) Object
B) everything
public void addRectangle(ArrayList<? extends Shape> shapes) {
    shapes.add( new Rectangle() ); // compile error!
}
public void addRectangle(ArrayList<? extends Shape> shapes) {
    shapes.add( new Rectangle() ); // compile error!
}

you don't get something for nothing: it is now illegal to modify shapes in this method.
why generics?
-everything in Java is an Object
- so, why not just make all data structures hold Objects?
-everything in Java is an **Object**
  - so, why not just make all data structures hold **Objects**?

- **generics allow for type-checking at compile time instead of run-time**
-everything in Java is an Object
  -so, why not just make all data structures hold Objects?

-generics allow for type-checking at compile time instead of run-time

-can detect type mismatch BEFORE your code runs
before generics:

ArrayList l;
l.add(new String("hi"));
Shape i = (Shape)l.get(0); // crash
before generics:

```java
ArrayList l;
l.add(new String(“hi”));
Shape i = (Shape)l.get(0); // crash
```

alternative:

```java
ArrayList<String> l;
l.add(new String(“hi”));
Shape i = (Shape)l.get(0); // compile error
```
before generics:

```java
ArrayList l;
l.add(new String("hi"));
Shape i = (Shape)l.get(0); // crash
```

alternative:

```java
ArrayList<String> l;
l.add(new String("hi"));
Shape i = (Shape)l.get(0); // compile error
```

**compile-time errors are always better than run-time!**
primitive types and generics
- generics only work with reference types
  - no int, char, float, double, ...

- what if we need an ArrayList of ints?

- Java has “wrapper” classes
  - Integer, Float, Double
  - these are reference types containing a single primitive...
  - ...and methods to access it
    - intValue(), doubleValue()
Java will automatically insert the appropriate code to convert between primitive/reference

```java
ArrayList<Integer> l;

l.add(5);
```

**equivalent to**

```java
l.add(new Integer(5));
```

```java
int i = l.get(n);
```

**equivalent to**

```java
int i = l.get(n).intValue();
```
questions...
questions...

*what types are included in* <? super Triangle>
A. Shape
B. Triangle, Circle, Rectangle, Square
C. Triangle, Shape, Object
questions...

what types are included in `<? super Triangle>`
A. Shape
B. Triangle, Circle, Rectangle, Square
C. Triangle, Shape, Object

what types are included in `<Shape>`
A. Shape
B. Triangle, Circle, Rectangle, Square
C. both 1 and 2
generic static methods
-static methods can have their own generic types

-declare the generic type before the return type:

```java
public static <T> boolean doWork(...) {...}
```

-we can refer to T as a type within that method only!
- **Static methods can have their own generic types**

- **Declare the generic type before the return type:**
  
  ```java
  public static <T> boolean doWork(...) {...}
  ```

- **We can refer to T as a type within that method only!**

- **Example:**
- **static** methods can have their own generic types

- **declare the generic type before the return type:**
  ```java
  public static <T> boolean doWork(...) {...}
  ```

- **we can refer to T as a type within that method only!**

- **example:**
  ```java
  public static <T> boolean contains(T[] array, T item) {
      for(int i=0; i < array.length; i++)
          if(array[i].equals(item))
              return true;
  
  return false;
  }
  ```
function objects
-suppose we want a generic sorting function

-and we want it to be able to sort ANY type...
-suppose we want a generic sorting function
  -and we want it to be able to sort ANY type…
  -what can we do?
-suppose we want a generic sorting function
  -and we want it to be able to sort ANY type…
  -what can we do?
  -what do we need to be able to do?
-suppose we want a generic sorting function
- and we want it to be able to sort ANY type…
- what can we do?
- what do we need to be able to do?

decide which item is larger
Comparable interface

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

-defines a natural ordering (*in fact, it is contractually obligated to!*)
    -String, Integer,... all implement Comparable

-what if we want a different ordering? or to order Shapes? or to order Strings based on length?
function objects

-a function object is an object that defines a single method

-example:

-a Comparator has a single method: compare
  -takes two arguments
  -decides which one is greater

-we write a sorting function that takes a Comparator
Comparator interface

public interface Comparator<T> {
    int compare(T left, T right);
}

- returns a number <0 if left < right
- returns a 0 if they are equal
- returns a number >0 if left > right
Collection interface
-a Collection **is a data structure that holds items**
  -very unspecific as to how the items are held
    - *ie. the data structure*

-supports various operations:
  -add, remove, contains, ...

-examples:
  -ArrayList
  -PriorityQueue
  -LinkedList
  -TreeSet
you’ll be working with Collections for assignment 3
- backed by an array for storage that you will implement yourselves
- items will be sorted as they are inserted
- no duplicates allowed

what are some of the issues with using an array?
add

int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

size = 0
add

int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

| 5 |   |   |   |   | size = 1 |


add

```java
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
```

5 17

size = 2
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

size = 3
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);

// don't forget size++
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
data.add(12);
data.add(1);
data.add(33);

5 17 9

size = 3

don't forget size++
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
data.add(12);
data.add(1);
data.add(33);

5 17 9 12 1 33

don't forget size++
size = 6
add

```java
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
data.add(12);
data.add(1);
data.add(33);
size = 6
```

Don't forget `size++`

data.add(22);
```java
int[] data = new int[6];
data.add(5);
data.add(17);
data.add(9);
data.add(12);
data.add(1);
data.add(33);

size = 6
```

```
data.add(22);
```

*now what?***
- we need to grow our array!

- avoid allocating slightly larger arrays
  - you will most likely need to grow again soon

- good rule of thumb is to double the size
  - **con:** wastes up to 2x space
  - **pro:** growth will be rare
grow

data → 5 17 9 12 1 33
grow

data → [5 17 9 12 1 33]

tmp = new int[data.length*2];

tmp →
grow

data $\rightarrow$ \[5\ 17\ 9\ 12\ 1\ 33\]

tmp = new int[data.length*2];

tmp $\rightarrow$ 

copy all from data to tmp

tmp $\rightarrow$ \[5\ 17\ 9\ 12\ 1\ 33\]
```java
grow

data = [5, 17, 9, 12, 1, 33]
tmp = new int[data.length*2];
tmp = [5, 17, 9, 12, 1, 33]

copy all from data to tmp

tmp = [5, 17, 9, 12, 1, 33]
data = tmp;

data = [5, 17, 9, 12, 1, 33]
```
remove

\[
\begin{array}{cccccc}
5 & 17 & 9 & 12 & 1 & 33 \\
\end{array}
\]

size = 6
data.remove(9);
remove

5 17 9 12 1 33 size = 6

data.remove(9);

5 17 12 1 33 size = 5
remove

5 17 9 12 1 33 size = 6

data.remove(9);

5 17 12 1 33 size = 5

is this correct?
A) yes
B) no
remove

5 17 9 12 1 33  \hspace{1cm} \text{size} = 6

\text{data.remove(9);}  

5 17 \hspace{0.5cm} 12 1 33  \hspace{1cm} \text{size} = 5

5 17 12 1 33  \hspace{1cm} \text{size} = 5
wait, what were we talking about?
Collection

-a Collection is an object that groups multiple elements into a single unit
  -used to store, retrieve, manipulate, and communicate stored data
  -is like an array except their size can change dynamically, and have more advanced behaviors

-the Java Collections API provides a set of classes and interfaces for storing data in different types of data structures
  -you will be implementing many of these on your own in assignments!
iterators
- not all data structures are guaranteed to use an array
  - thus, we can’t just do:

    ```java
    for (i = 0; i < size; i++)
        data[i]...
    ```

- the `Iterator` interface provides generic retrieval of items from a data structure

- the `Collection` interface requires an `Iterator`
  - for example, `ArrayCollection` has `iterator()` method that returns an `Iterator`
**Iterator**

- An Iterator is specific to a data structure, and knows how to traverse the structure.
  - **hasNext**: determines if iteration is complete
  - **next**: gets the next item
  - **remove**: removes the last seen item

- Internally, keeps track of where the next item is (as well as other state)

- Actually points to *between* items
next

<table>
<thead>
<tr>
<th>iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 17 9 12 1 33</td>
</tr>
</tbody>
</table>
`next`

```
iterator

5 17 9 12 1 33

iterator.next(); // returns 9
```
next

```java
iterator = 5 17 9 12 1 33

iterator.next(); // returns 9
iterator.next(); // returns 12
```
next

iterator

5 17 9 12 1 33

iterator.next(); // returns 9
iterator.next(); // returns 12
iterator.next(); // returns 1
`remove` removes the last item seen
**remove** removes the last item seen

```
iterator

```

5 17 9 12 1 33

```
iterator.remove(); // removes 12

```
remove removes the last item seen

```java
iterator

5 17 9 12 1 33

iterator.remove(); // removes 12

iterator

5 17 9 1 33
```
**remove** removes the last item seen

```java
iterator.remove(); // removes 12
```

**iterator** must be preceded by at least one call to `next()`
enhanced for-loop

- allows for simplification of code by representing a visit to each element of an array or Collection without explicitly expressing how you get from element to element

**STANDARD WAY:**
for(int i=0; i < things.length; i++)
  // do something with things[i]

**ENHANCED LOOP:**
for(Thing t : things)
  // do something with t

- uses an Iterator behind the scenes!
```java
ArrayList<String> catNames = new ArrayList<String>();
catNames.add("Mooches");
catNames.add("Butterscotch");
catNames.add("Mr.Pickles");

useWhileLoop(catNames);
useForEnhancedForLoop(catNames);

void useWhileLoop(Collection<String> c)
{
    Iterator<String> iter = c.iterator();
    while (c.hasNext())
        System.out.println(iter.next());
}

void useEnhancedForLoop(Collection<String> c)
{
    for (String cn : c)
        System.out.println(cn);
}
```
next time...
- no class on Monday

- reading
  - chapters 5 & 6

- homework
  - assignment 1 due today at midnight
  - assignment 2 due next Wednesday at midnight

- lab on Friday
  - timing