# CSE 1325: Object-Oriented Programming

## Lecture 13

# Generics


## Mr. George F. Rice

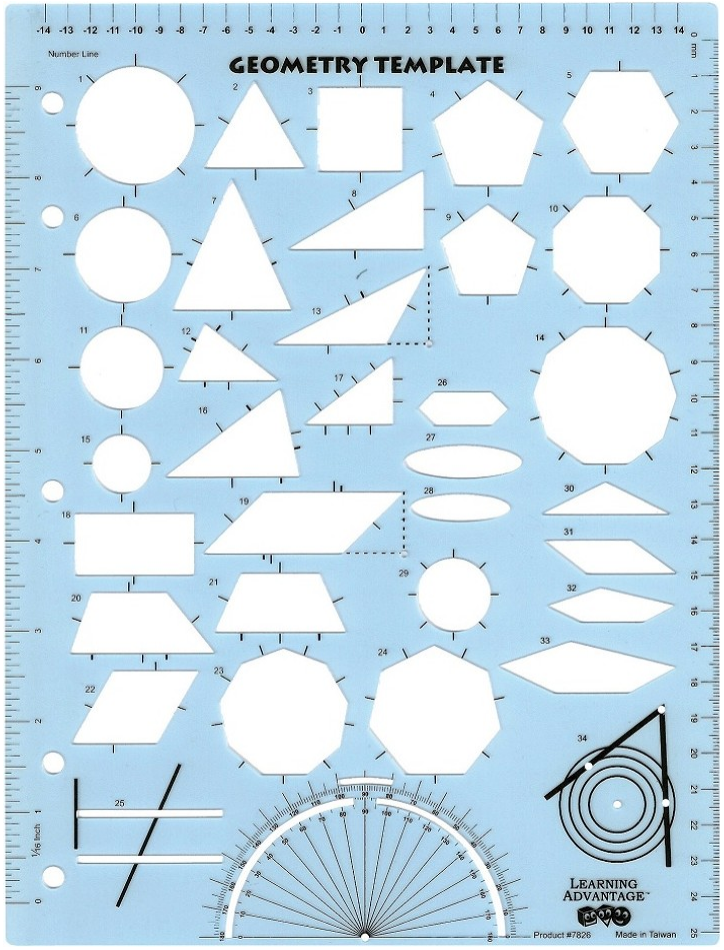
george.rice@uta.edu


Office Hours:
Prof Rice 11:00 Tuesday and
Thursday in ERB 336
For TAs see this web page

Helium walks into a bar and orders a beer.
The bartender replies, "We don't serve noble gases here!"
He doesn't react.

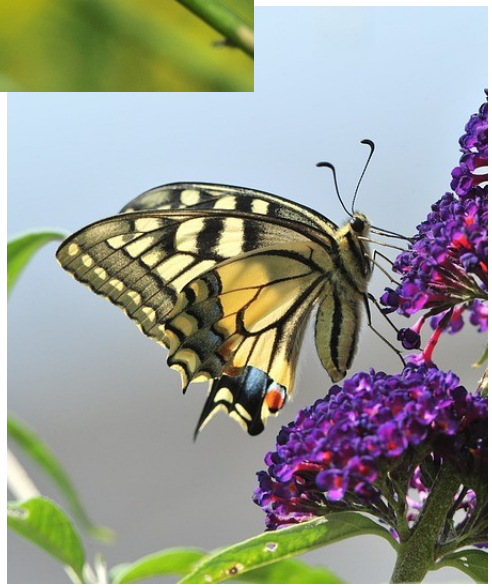
# Overview: Generic Classes

- Polymorphism
- Generics
  - Alternates
  - Java Implementation
  - Bounded & Unbounded Wildcards
- Java Examples
  - SortedArray (Java class)
  - printIt (Java method)
  - ImplementsComparable (Java interface)
  - max (Java method)
  - TaggedArrayList (Java class)



# Polymorphism

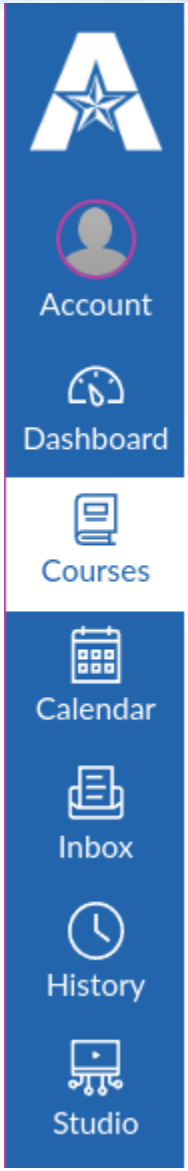
- **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results.

- In Java (unlike C++), no special coding techniques are required.
  - Although have I mentioned `@Override` on the polymorphic methods is both *wise* and *expected* in CSE1325?

Caterpillar and Butterfly by MrLebies per the Pixabay License
https://pixabay.com/photos/dovetail-caterpillar-butterfly-1201461/

# Pop Quiz (in Canvas)




2232-CSE-1325-001 > Assignments

2023 Spring

Search for Assignment

Home
Syllabus
Modules
Assignments
Quizzes
Grades
Echo360
People
Course Evaluations
UTA Libraries
StudyMate

▾ Upcoming Assignments

**Lecture 00 Quiz**
Not available until Jan 17 at 8:00am | Due Jan 19 at 8am | -/5 pts

**P01 - Starting Out With Hello!**
Due Jan 24 at 8am | -/100 pts

▾ Undated Assignments

**Pop Quiz 13**
Not available until Jan 23 at 7:00am | 2 pts | 2 Questions

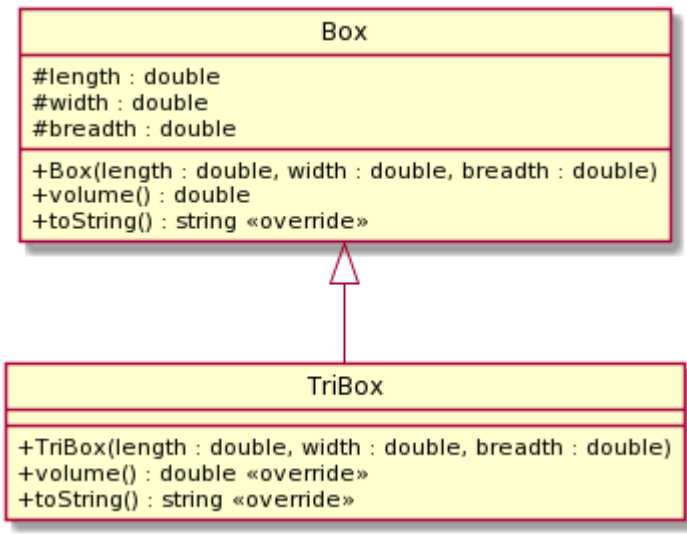
The access code is on the whiteboard.

You have 1 minute.

# The Box Class

- Consider a rectangular box

```
public class Box {
    public Box(double length, double width, double height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
    public double volume() {return length * width * height;}

    @Override
    public String toString() {
        return "Rectangular box ("
            + length + " x " + width + " x " + height + ")";
    }

    protected double length;
    protected double width;
    protected double height;
}
```



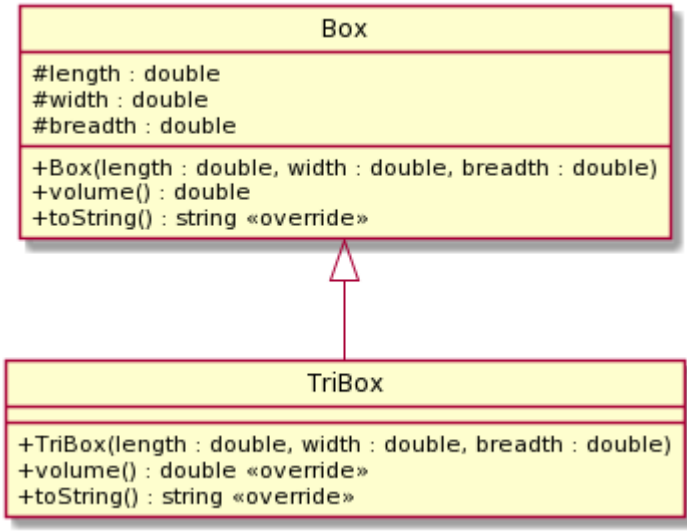

"protected" means that classes derived from Box may read or modify (but NOT INITIALIZE) this field. Only the superclass in which it is defined may initialize it!

Box image by OpenClipart-Vectors per the Pixabay License
https://pixabay.com/vectors/parcel-package-packaging-box-575623/

# The Box Class

## with equals and hashCode implementations

```
import java.util.Objects;

public class Box {
    public Box(double length, double width, double height) {
        this.length = length;  this.width = width;  this.height = height;
    }
    public double volume() {return length * width * height;}

    @Override public String toString() {
        return "Rectangular box ("
            + length + " x " + width + " x " + height + ")";
    }

    @Override public int hashCode() {
        return Objects.hash(length, width, height);
    }

    @Override public boolean equals(Object o) {
        if(o == this) return true;
        if(o == null || o.getClass() != this.getClass()) return false;
        Box b = (Box) o;
        return (b.length == length) && (b.width  == width) && (b.height == height);
    }
    protected double length;
    protected double width;
    protected double height;
}
```




We note and ignore the potential round-off issues here

# What About a TriBox?

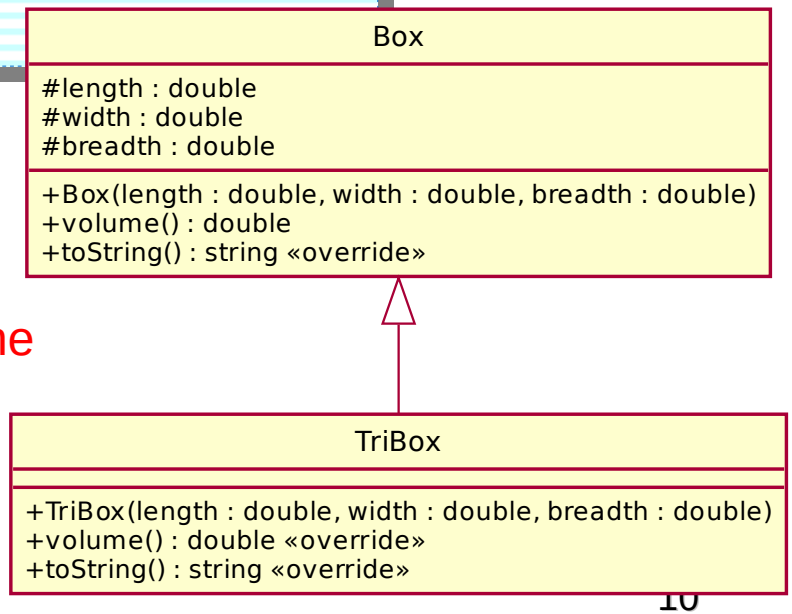
- TriBox could reuse some Box code
  - The same fields as a Box
  - The same constructor parameters
  - The same methods and signatures
- It is likely we would want to store both Boxes and TriBoxes in a Java container of Boxes
- For these reasons – reuse and shared containers – inheritance may be a good choice

# Inheriting From ("Extending") the Box Superclass

```
public class TriBox extends Box {
    public TriBox(double length, double width, double height) {
        super(length, width, height);
    }
    @Override
    public double volume() {return 0.5 * length * width * height;}

    @Override
    public String toString() {
        return "Triangular box ("
            + length + " x " + width + " x " + height + ")";
    }
    // Box.equals and Box.hashCode work for TriBox
}
```


Since constructors don't inherit, we explicitly call ("delegate to") the superclass' constructor (reuse). This must be the FIRST line in the constructor!

We override **`volume()`** and **`toString()`**. ALWAYS use the **@Override** annotation to tell the compiler to warn you if your superclass has no method of the same name and parametric signature to override.
**It will save you HOURS of debugging!**



# Testing Box and Tribox

```
class Boxes {
    public static void main(String[] args) {
        // Let's try a couple of rectangular boxes
        Box box1 = new Box( 6,  7,  5);
        Box box2 = new Box(12, 13, 10);

        System.out.println("Volume of " + box1 + " is " + box1.volume());
        System.out.println("Volume of " + box2 + " is " + box2.volume());

        // Let's try a couple of triangular boxes
        TriBox box3 = new TriBox( 6,  7,  5);
        TriBox box4 = new TriBox(12, 13, 10);

        System.out.println("Volume of " + box3 + " is " + box3.volume());
        System.out.println("Volume of " + box4 + " is " + box4.volume());
    }
}
```

```
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac Boxes.ja
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java Boxes
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```

Here we create objects of the derived class and use them directly.

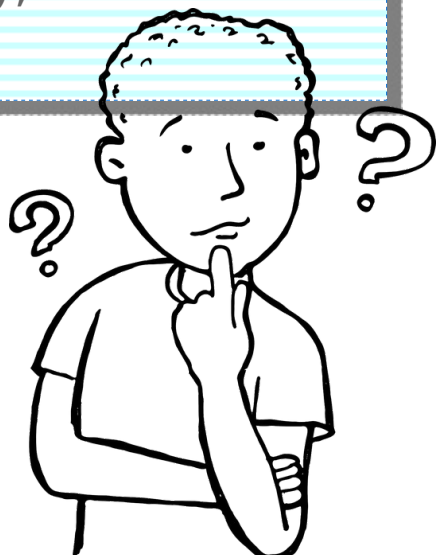
Some of the features of the base class inherit to derived scope – the protected data, for example – and some do not – the volume() method. We could also add additional fields and methods to the derived class as needed.

# (Trying to) Store Extended Object in Superclass Variable

```
class Boxes {
    public static void main(String[] args) {
        // Let's try a couple of rectangular boxes
        Box box1 = new Box( 6,  7,  5);
        Box box2 = new Box(12, 13, 10);

        System.out.println("Volume of " + box1 + " is " + box1.volume());
        System.out.println("Volume of " + box2 + " is " + box2.volume());

        // Let's try a couple of rectangular boxes
        Box box3 = new TriBox( 6,  7,  5);
        Box box4 = new TriBox(12, 13, 10);

        System.out.println("Volume of " + box3 + " is " + box3.volume());
        System.out.println("Volume of " + box4 + " is " + box4.volume());
    }
}
```

What if we try to store a TriBox (subclass object) in a Box variable (superclass variable)? Will we get

- A compiler error
- Output identical to a Box object
- Correct output for a TriBox object?



Questioning Kid by Clker-Free-Vector-Images per the Pixabay License
https://pixabay.com/vectors/thinker-thinking-person-idea-28741/

# Storing an Extended Object in a Superclass Variable

```
class Boxes {
    public static void main(String[] args) {
        // Let's try a couple of rectangular boxes
        Box box1 = new Box( 6,  7,  5);
        Box box2 = new Box(12, 13, 10);

        System.out.println("Volume of " + box1 + " is " + box1.volume());
        System.out.println("Volume of " + box2 + " is " + box2.volume());

        // Let's try a couple of rectangular boxes
        Box box3 = new TriBox( 6,  7,  5);
        Box box4 = new TriBox(12, 13, 10);

        System.out.println("Volume of " + box3 + " is " + box3.volume());
        System.out.println("Volume of " + box4 + " is " + box4.volume());
    }
}
```

```
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac Boxes.ja
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java Boxes
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```

No problems!
As long as the object type is a subclass of the variable type, we may assign the reference without issues.

# Upcasting

- Storing a subclass object in a superclass variable is called “upcasting”
  - No explicit cast syntax is required

```
TriBox t = new TriBox(3,4,5);
Box b = t; // works just fine
```

  - Only methods defined in the variable type are available in an upcast variable
    - If TriBox had added a method named foo() not defined in Box, then
      - `t.foo()` would work just fine
      - `b.foo()` would generate a compiler error
- Upcasting is *very* common in polymorphic programs

# (Trying to) Store Mixed Objects in an ArrayList

```
import java.util.ArrayList;

public class BoxesArray {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes)
            System.out.println("Volume of " + box + " is " + box.volume());
    }
}
```

What if we try to store both Boxes and TriBoxes in a Box container (e.g., ArrayList)? Will we get

- A compiler error
- Output identical to a Box object
- Correct output for a TriBox object?

Questioning Kid by Clker-Free-Vector-Images per the Pixabay License
https://pixabay.com/vectors/thinker-thinking-person-idea-28741/

# Storing Mixed Objects in an ArrayList

```
import java.util.ArrayList;

public class BoxesArray {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes)
            System.out.println("Volume of " + box + " is " + box.volume());
    }
}
```

```
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac BoxesArray.
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java BoxesArray
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```

Still no problem!

We may use a subclass object in virtually any place we could use a superclass object. The initialization is another example of upcasting. Upcasting enables **polymorphism**.

# Downcasting

- Storing a subclass object referenced by a superclass (or interface) variable in a subclass variable is called “downcasting”
  - C-like casting is required

```
Box b = new TriBox(3,4,5);
Tribox t = (TriBox) b; // Explicit cast is required
```

  - If the superclass variable is not referencing an object of the subclass type, an exception will be thrown
- The instanceof operator is helpful to verify types at runtime

```
Box b = new TriBox(3,4,5);
Tribox t;
if(b instanceof Tribox) t = (TriBox) b;
else t = null;
```

- Starting in Java 16, instanceof can auto-cast for you!

```
Box b = new TriBox(3,4,5);
if(b instanceof Tribox t) System.out.println(t.toString());
```

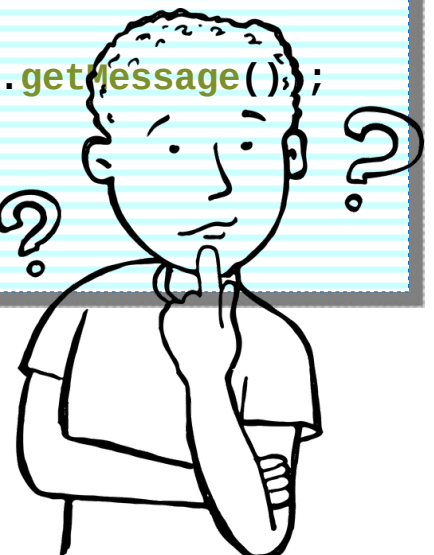
# How to Downcast

```
import java.util.ArrayList;

public class BoxesDowncast {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes) {
            try {
                TriBox tb = (TriBox) box;
                System.out.println("Downcast " + tb);
            } catch(ClassCastException e) {
                System.err.println("Failed to downcast " + box + ": " + e.getMessage());
            }
        }
    }
}
```

Here we add 2 Box and 2 TriBox objects to an ArrayList<Box>. Then we try to cast each element to a Tribox.

Will we get

- A compiler error
- One or more exceptions (How many? When?)
- *Still* no problem!



# How to Downcast

```
import java.util.ArrayList;

public class BoxesDowncast {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes) {
            try {
                TriBox tb = (TriBox) box;
                System.out.println("Downcast " + tb);
            } catch(ClassCastException e) {
                System.err.println("Failed to downcast " + box + ": " + e.getMessage());
            }
        }
    }
}
```

We are attempting to downcast both Box and TriBox instances to a TriBox. TriBox instances correctly downcast, while Box instances throw an (unchecked) ClassCastException.

```
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java BoxesDowncast
Failed to downcast Rectangular box (6.0 x 7.0 x 5.0): class Box cannot be cast to
 class TriBox (Box and TriBox are in unnamed module of loader 'app')
Failed to downcast Rectangular box (12.0 x 13.0 x 10.0): class Box cannot be cast
 to class TriBox (Box and TriBox are in unnamed module of loader 'app')
Downcast Triangular box (6.0 x 7.0 x 5.0)
Downcast Triangular box (12.0 x 13.0 x 10.0)
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$
```

# A Really Simple Challenge

- Write a static method named max that returns the larger of two integers

# A Really Simple Challenge

- Write a static method named max that returns the larger of two Integer objects*

```
public class Max {
    // for Integer
    public static Integer max(Integer lhs, Integer rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
        // more concisely, return lhs.compareTo(rhs) > 0 ? lhs : rhs;
    }
}
```

- Wait – we sometimes need this for doubles, so also provide that version

*While we *could* simply use lhs>rhs, that only works with primitives. We're headed toward objects, so let's use compareTo now.

# A Really Simple Challenge

- Write a static method named max that returns the larger of two Integer objects

```
public class Max {
    // for Integer
    public static Integer max(Integer lhs, Integer rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

- Wait – we sometimes need this for Double objects, so also provide that version

```
    // for Double
    public static Double max(Double lhs, Double rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
```

# A Really Simple Challenge

- Wait – I have a cool custom 3D Coordinate class, so provide THAT version, too!

```
class Coordinate implements Comparable<Coordinate> {
    public Coordinate(double x, double y, double z) {
        this.x = x; this.y = y; this.z = z;
    }
    public Double magnitude() {return Math.sqrt(x*x + y*y + z*z);}
    @Override
    public int compareTo(Coordinate rhs) {
        return magnitude().compareTo(rhs.magnitude());
    }
    @Override
    public String toString() {return "(" + x + "," + y + "," + z + ")";}
    double x, y, z;
}
```

# A Really Simple Challenge

- Wait – I have a cool custom 3D Coordinate class, so provide THAT version, too!

```
class Coordinate implements Comparable<Coordinate> {
    public Coordinate(double x, double y, double z) {
        this.x = x; this.y = y; this.z = z;
    }
    public Double magnitude() {return Math.sqrt(x*x + y*y + z*z);}
    @Override
    public int compareTo(Coordinate rhs) {
        return magnitude().compareTo(rhs.magnitude());
    }
    @Override
    public String toString() {return "(" + x + "," + y + "," + z + ")";}
    double x, y, z;
}
```

```
    // for Coordinate
    public static Coordinate max(Coordinate lhs, Coordinate rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
```

# A Really Simple Challenge

- Wait – I've found a new career!
  - I can write max functions forever and STILL not cover all of the possible types

```
public class Max {
    public static Integer max(Integer lhs, Integer rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static Double max(Double lhs, Double rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static Coordinate max(Coordinate lhs, Coordinate rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static void main(String[] args) {
     System.out.println(max(3,5) + ", " + max(3.14, 2.78)
         + ", " + max(new Coordinate(3, 4, 5), new Coordinate(6, 7, 8)));
    }
}
```

But the *algorithm* never changes – only the *type*!

How could I write the algorithm *independent* of type, and apply the type later?

# Option #1: Duplicate and Edit the Code

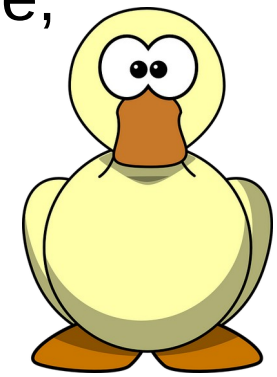
- Sometimes brute force is the best answer
  - But not this time – since bugs are proportional to the number of lines of code, we're duplicating bugs

```
public class Max {
    public static Integer max(Integer lhs, Integer rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static Double max(Double lhs, Double rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static Coordinate max(Coordinate lhs, Coordinate rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

We don't want more bugs. Moving on...

# Option #2: Dynamic Languages

- Dynamic languages (e.g., Python) don't enforce types
  - "Duck typing" – any variable may hold any object type, provided it offers the needed methods
  - Downside: No clear type specifications
  - Downside: If any object DOESN'T offer the needed methods, you find out at *runtime*, not compile time



```
#!/usr/bin/env python3
def max(lhs, rhs):
  return lhs if lhs>rhs else rhs

print("The larger of 3 and 42 is {}".format(max(3,42)))
print("The larger of 3.1415 and 2.718 is {}".format(max(3.1415,2.718)))
print("The larger of (1,2,3) and (2,1,-4) is {}".format(
        max(Coordinate(1,2,3),Coordinate(2,1,-4))))
```

```
student@cse1325:/media/sf_dev/20$ make max.py
python3 max.py
The larger of 3 and 42 is 42
The larger of 3.1415 and 2.718 is 3.1415
The larger of (1,2,3) and (2,1,-4) is (2,1,-4)
student@cse1325:/media/sf_dev/20$
```

And Java isn't a dynamic language anyway. Moving on...

# Option #3: Write a Code Generator

- We can automate the code duplication
  - The bugs are now solo in the code generator
  - Downside – This complicates the build.xml and makes learning to support the code more difficult

```
#!/usr/bin/env python3
def gen_max(type):
    "Generate Java code for static method max"
    print(f"{type} max({type} lhs, {type} rhs)");
    print("{\n    if (lhs > rhs) return lhs; else return rhs;\n}")

gen_max("int")
gen_max("double")
gen_max("Container")
```

```
student@cse1325:/media/sf_dev/20$ make code_generator.py
python3 code_generator.py
int max(int lhs, int rhs)
{
    if (lhs > rhs) return lhs; else return rhs;
}
double max(double lhs, double rhs)
{
    if (lhs > rhs) return lhs; else return rhs;
}
Container max(Container lhs, Container rhs)
{
    if (lhs > rhs) return lhs; else return rhs;
}
student@cse1325:/media/sf_dev/20$
```

We don't want complicated support. Moving on...

# Option #4: Use a Preprocessor

- A preprocessor (as with C++) lexically mangles the code before passing it to the compiler
  - This is essentially a built-in code generator
  - Downside – The preprocessor is tricky and non-obvious, particularly in the *debugger*

```
#define gen_max(item) \
item max(item lhs, item rhs) {if (lhs > rhs) return lhs; else return rhs;}

gen_max(int)
gen_max(double)
gen_max(Coordinate)


int main() {
    std::cout << "The larger of 3 and 42 is " << max(3, 42) << std::endl;
    std::cout << "The larger of 3.1415 and 2.718 is " << max(3.1415, 2.718) << std::endl;
    std::cout << "The larger of (1,2,3) and (2,1,-4) is "
              << max(Coordinate{1,2,3},Coordinate{2,1,-4}) << std::endl;
}
```

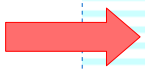
```
student@cse1325:/media/sf_dev/20$ make preprocessor
g++ --std=c++17    preprocessor.cpp   -o preprocessor
student@cse1325:/media/sf_dev/20$ ./preprocessor
The larger of 3 and 42 is 42
The larger of 3.1415 and 2.718 is 3.1415
The larger of (1,2,3) and (2,1,-4) is (2,1,-4)
student@cse1325:/media/sf_dev/20$
```

We don't want tricky and non-obvious. Moving on...

# Option #5: Use Generic Methods

## (or Generic Classes – together, "Generics")

- Generics are "smart" preprocessor macros
  - We parameterize the type inside angle brackets as T
  - Type T for each method call is inferred from the call parameters
  - The *compiler* validates the types when the generic class is instanced or the generic method is called



```
public class MaxGeneric {
    // Generic for any class type that implements Comparable (i.e., has compareTo)
    public static <T extends Comparable<T>> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
    public static void main(String[] args) {
        System.out.println(max(3,5) + ", " + max(3.14, 2.78)
            + ", " + max(new Coordinate(3, 4, 5), new Coordinate(6, 7, 8)));
    }
}
```

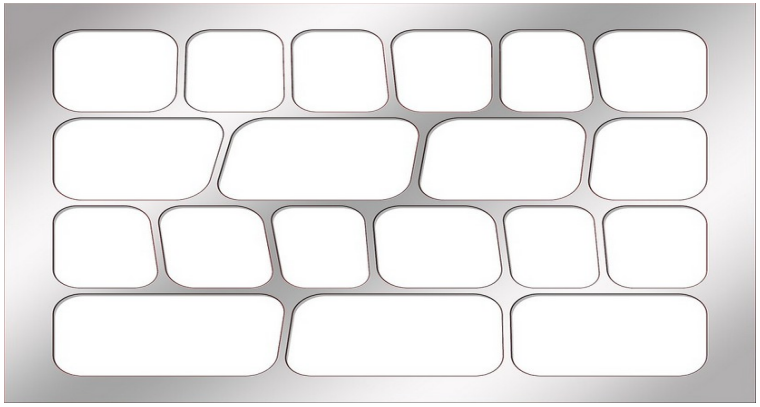
This keeps only one copy of the bugs
and allows the code to be easily readable and maintainable

# Generic

- **Generic** – A Java construct representing a **method** or **class** in terms of generic types


  - This enables the algorithm to be written independent of the types of data to which it applies
  - The type is specified or (often) inferred when the generic class is instanced or the generic method is called

- As with dynamic languages, the type specified must supply the needed methods



  - Our max template will only work for types that override `compareTo`
  - Unlike Python, Java verifies this at *compile* time

# Generic Programming

- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation



  - In Java, generic programming may apply to **methods** or **classes**
  - Java requires that the types be *explicitly* constrained to provide the necessary methods
    - Unlike (say) C++, where the compiler verifies on the fly at compile time
    - Unlike (say) Python, where the virtual machine verifies on the fly at runtime

# Other Names for Generics

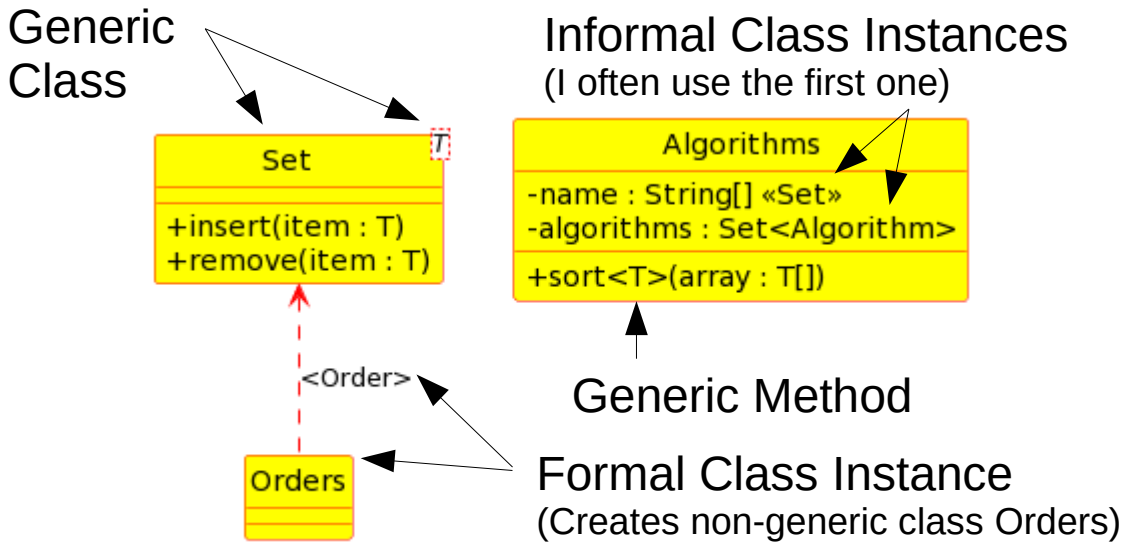
- Generics
  - Ada (among the first to support it), Java, C#, and Objective-C
- Parameterized Types
  - Design patterns, UML (see next page)
- Templates
  - C++
- Parametric Polymorphism
  - Scala, Haskell
  - Also sometimes called **Compile-Time Polymorphism**

In contrast, polymorphism with subclasses that we learned last lecture would be called **Dynamic Polymorphism** or **Runtime Polymorphism**

# Generics in the UML

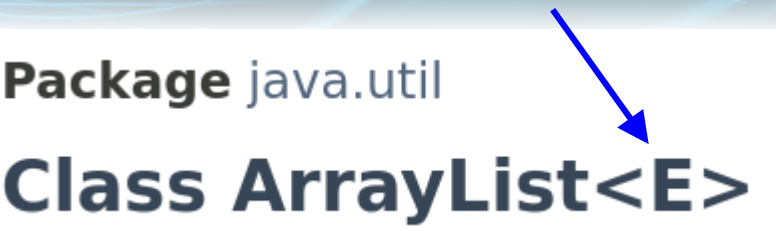
- The T in a dashed box indicates a generic class
  - Multiple letters are permitted, for example, T,U,V
- A <T> immediately following the method name indicates a generic method
- A dependency line labeled with the type indicates an instance for that type
  - Although the generic class name with type in <>, or a stereotype, is often used

This is on the exam!



# Generics We Know and Love

- As you may now suspect, ArrayList is a **generic class**




  - The generic type – what we want to store – is E
  - For example, to store Customer objects, write
    `ArrayList<Customer> = new ArrayList<>();`
  - The generic type E is therefore Customer here

- sort from the Collections class is a **generic method**

```
sort(List<T> list)
```

  - We can sort any List (such as an ArrayList) of any type T
  - With methods, we don't usually specify the type on a call – as with var, javac can figure it out from the parameter!

# Writing a Simple Generic Class: SortedArray

- Let's create a simple array class in which the data (of ANY Object type) is always sorted

```
import java.util.Arrays;
public class SortedArray<E> {
    public SortedArray(int size) {
        array = new Object[size];
        nextElement = 0;
    }
    public int length() {
        return array.length;
    }
    public int size() {
        return nextElement;
    }
    public void add(E e) {
        array[nextElement] = e;
        Arrays.sort(array, 0, ++nextElement);
    }
    public E get(int index) {
        return (E) array[index];
    }

    private Object[] array;
    private int nextElement;
}
```

E is the type that is specified when instanced.

<E> after the class name tells Java our class is generic!

For simplicity, we have a fixed maximum size.

Length returns that fixed maximum size.

Size is the actual number of elements added.

Add appends a new element to the array and sorts the elements.

Get returns the element at that index, using a *downcast* (remember those?).

Java (unlike C++) won't permit E[ ] to be declared, but in this case we can get away with Object[ ].

# Thoughts on SortedArray

- We could write a similar class *without* generics
  - It would store and return Objects, relying on polymorphism to store any subclass of Object
  - BUT we would have to cast the returned Object to our desired type for *every* call to the get method
- So SortedArray gives us a small but notable win

# Testing SortedArray

- Converting to a true regression test is left as an exercise for the discerning student (it's quite easy!)

```
public class TestSortedArray {
    public static void main(String[] args) {
        // Declare the test data
        String[] testData = new String[]{
            "This", "is", "five", "words", "long"
        };

        // Print the test data
        System.out.print("The test data is [");
        for(var s : testData) System.out.print(s + " ");
        System.out.println("]");

        // Declare the SortedArray and print its attributes
        SortedArray<String> sas = new SortedArray<>(testData.length);
        System.out.println("Initial sas size = " + sas.size() +
                           ", length = "         + sas.length());

        // Load the test data
        for(var s : testData) sas.add(s);
```

# Testing SortedArray

```
        // Try to exceed size limit
        try {
            sas.add("I should cause an Exception");
            System.err.println("FAIL - added 6th element to 5 element SortedArray");
        } catch(Exception e) {
        }

        // Check size and length again
        System.out.println("Final   sas size = " + sas.size() +
                           ", length = "         + sas.length());

        // Print the sorted test data
        System.out.print("SortedArray   is [");
        for(int i=0; i<sas.size(); ++i)
            System.out.print(sas.get(i) + " ");
        System.out.println("]");
    }
}
```

```
ricegf@antares:~/dev/202301/22/code_from_slides/SortedArray$ javac TestSortedArray.java
ricegf@antares:~/dev/202301/22/code_from_slides/SortedArray$ java TestSortedArray
The test data is [This is five words long ]
Initial sas size = 0, length = 5
Final   sas size = 5, length = 5
SortedArray   is [This five is long words ]
ricegf@antares:~/dev/202301/22/code_from_slides/SortedArray$
```

# Writing a Generic Method: printIt

- We can also write our own generic methods with <T>
  - We can then use T as a placeholder for the type to be supplied later
  - Again, in Java, with methods T's value is inferred

T is the type that is specified when instanced.
<T> before the return type tells Java our method is generic!

```java
import java.util.ArrayList;

public class SimpleGenericMethod {
    public static <T> void printIt(T value) {
        System.out.println(value);
    }
    public static void main(String[] args) {
        printIt(42);
        printIt("Hello, World!");
        ArrayList<Double> doubles = new ArrayList<>();
        doubles.add(Math.PI); doubles.add(Math.E);
        printIt(doubles);
    }
}
```

```
ricegf@antares:~/dev/202301/22/code_from_slides$ javac SimpleGenericMethod.java
ricegf@antares:~/dev/202301/22/code_from_slides$ java SimpleGenericMethod
42
Hello, World!
[3.141592653589793, 2.718281828459045]
ricegf@antares:~/dev/202301/22/code_from_slides$
```

# Interfaces can be Generic, Too!

java.lang

**Interface Comparable<T>**

Look – Comparable is generic!

**Type Parameters:**

`T - the type of objects that this object may be compared to`

compareTo returns
-x if less than
0 if equal
x if greater than

***Method Summary***

| All Methods | Instance Methods | Abstract Methods |
|---|---|---|

| Modifier and Type | Method and Description |
|---|---|
| int | compareTo(T o)<br>Compares this object with the specified object for order. |

```
import java.lang.Math;

public class Coordinate implements Comparable<Coordinate> {
    public Coordinate(double x, double y, double z) {
        this.x = x; this.y = y; this.z = z;
    }
    public Double magnitude() {return Math.sqrt(x*x + y*y + z*z);}
    @Override
    public int compareTo(Coordinate rhs) {
        return magnitude().compareTo(rhs.magnitude());
    }
    @Override
```

# Breaking Down Coordinate

- Our custom Coordinate class implements Comparable for two Coordinate objects

Compare this type

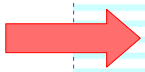
to this type!

As with `boolean equals(Object o)`, the two types don't *have* to match, although they usually do

```
import java.lang.Math;

public class Coordinate implements Comparable<Coordinate> {
    public Coordinate(double x, double y, double z) {
        this.x = x; this.y = y; this.z = z;
    }
    public Double magnitude() {return Math.sqrt(x*x + y*y + z*z);}
    @Override
    public int compareTo(Coordinate rhs) {
        return magnitude().compareTo(rhs.magnitude());
    }
    @Override
    public String toString() {return "(" + x + "," + y + "," + z + ")";}
    double x, y, z;
}
```

# Testing Coordinate.compareTo

- Our custom Coordinate class implements Comparable for two Coordinate objects




```
public class ImplementsComparable {
    public static void main(String[] args) {
        Coordinate c = new Coordinate(3, 4, 5);
        for(int x = 3; x < 7; x += 2) {
          for(int y = 3; y < 7; y += 2) {
            for(int z = 3; z < 7; z += 2) {
                Coordinate c2 = new Coordinate(x, y , z);
                System.out.println("For \"" + c2 + "\", compareTo = "
                                   + c.compareTo(c2));
            }
          }
        }
    }
}
```

```
For "(3.0,3.0,3.0)", compareTo = 1
For "(3.0,3.0,5.0)", compareTo = 1
For "(3.0,4.0,3.0)", compareTo = 1
For "(3.0,4.0,5.0)", compareTo = 0
For "(5.0,3.0,3.0)", compareTo = 1
For "(5.0,3.0,5.0)", compareTo = -1
For "(5.0,4.0,3.0)", compareTo = 0
For "(5.0,4.0,5.0)", compareTo = -1
```

# Attempting a Generic Max Method

- So you would *expect* we could write our long-awaited generic max static method like this:

```
public class MaxGeneric {
    public static <T> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

# Attempting a Generic Max Method

- So you would *expect* we could write our long-awaited generic max static method like this:

```
public class MaxGeneric {
    public static <T> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

But we can't!

```
ricegf@antares:~/dev/202401/13-java-generics/code_from_slides/max$ javac MaxGeneric.java
MaxGeneric.java:6: error: cannot find symbol
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
                ^
  symbol:   method compareTo(T)
  location: variable lhs of type T
  where T is a type-variable:
    T extends Object declared in method <T>max(T,T)
1 error
ricegf@antares:~/dev/202401/13-java-generics/code_from_slides/max@
```

What went wrong?

# Object has no compareTo!

- Interface Comparable adds compareTo

**Module** java.base
**Package** java.lang

## Class Object

All Methods | Instance Methods | Concrete Methods | Deprecated Methods

| Modifier and Type | Method | Description |
|---|---|---|
| protected Object | clone() | Creates and returns a copy of this object. |
| boolean | equals(Object obj) | Indicates whether some other object is "equal to" this one. |
| protected void | finalize() | **Deprecated.** The finalization mechanism is inherently problematic. |
| final Class<?> | getClass() | Returns the runtime class of this Object. |
| int | hashCode() | Returns a hash code value for the object. |
| final void | notify() | Wakes up a single thread that is waiting on this object's monitor. |
| final void | notifyAll() | Wakes up all threads that are waiting on this object's monitor. |
| String | toString() | Returns a string representation of the object. |
| final void | wait() | Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*. |
| final void | wait(long timeoutMillis) | Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed. |
| final void | wait(long timeoutMillis, int nanos) | Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed. |

**Module** java.base
**Package** java.lang

## Interface Comparable<T>

All Methods | Instance Methods | Abstract Methods

| Modifier and Type | Method | Description |
|---|---|---|
| int | compareTo(T o) | Compares this object with the specified object for order. |

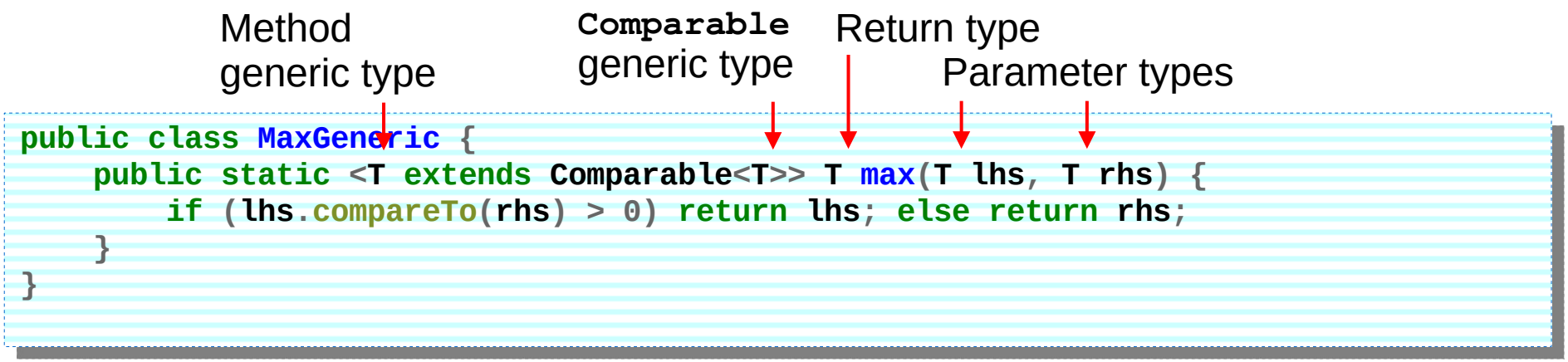
# If a Constraint Exists, Say So!

- Java requires that we *constrain* our generic types to guarantee our code will compile
  - Classes that do not implement the Comparable interface don't have the compareTo method we need
- We need syntax to specify our generic type constraints – T must implement Comparable

```
public class MaxGeneric {
    public static <T> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

# Fixing our Generic Max Method

- Instead of a simple **`<T>`**
  - "Use any non-primitive type that you want"
- We use **`<T extends Comparable<T>>`**
  - "Type T must implement generic interface Comparable for the same type T"




```
public class MaxGeneric {
    public static <T extends Comparable<T>> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

Yes, that's a *lot* of Ts!

# Constraining the Generic Type(s)

- Let's break down this generic method

```
public static <T extends Comparable<T>> T max(T lhs, T rhs) {
    if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
}
```

  - **`public static`** have their usual meaning
  - The generic spec always precedes the return type
  - **`<T extends Comparable`** means that the type supplied to the template must extend / implement the &-separated class and / or interface(s)
    - The class must be listed first: **`<T extends C & I1 & I2>`**
    - **`extends`** is used for both; *never* use **`implements`** here
  - **`<T>`** means Comparable's implementation must be for class T
  - **`T max(T lhs, T rhs)`** replaces each T with the supplied type

# Using our Generic Max Method

- Since our custom Coordinate class implements Comparable for a Coordinate object...

```
import java.lang.Math;

public class Coordinate implements Comparable<Coordinate> {
    public Coordinate(double x, double y, double z) {
        this.x = x; this.y = y; this.z = z;
    }
    public Double magnitude() {return Math.sqrt(x*x + y*y + z*z);}
    @Override
    public int compareTo(Coordinate rhs) {
        return magnitude().compareTo(rhs.magnitude());
    }
    @Override
    public String toString() {return "(" + x + "," + y + "," + z + ")";}
    double x, y, z;
}
```

# Using our Generic Max Method

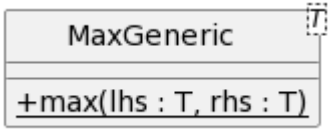
- Our max method now works with Coordinate!

```
import java.util.Arrays;

public class ImplementsMax {
    public static <T extends Comparable<T>> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }

    public static void main(String[] args) {
        Coordinate c = new Coordinate(3, 4, 5);
        for(int x = 3; x < 7; x += 2) {
          for(int y = 3; y < 5; y += 1) {
            for(int z = 3; z < 7; z += 2) {
                Coordinate c2 = new Coordinate(x, y , z);
                System.out.println("Of " + c + " and " + c2 + ", max is " + max(c, c2));
            }
          }
        }
    }
}
```

```
ricegf@antares:~/dev/202401/13-java-generics/code_from_slides/max$ javac ImplementsMax.j
ricegf@antares:~/dev/202401/13-java-generics/code_from_slides/max$ java ImplementsMax
Of (3.0,4.0,5.0) and (3.0,3.0,3.0), max is (3.0,4.0,5.0)
Of (3.0,4.0,5.0) and (3.0,3.0,5.0), max is (3.0,4.0,5.0)
Of (3.0,4.0,5.0) and (3.0,4.0,3.0), max is (3.0,4.0,5.0)
Of (3.0,4.0,5.0) and (3.0,4.0,5.0), max is (3.0,4.0,5.0)
Of (3.0,4.0,5.0) and (5.0,3.0,3.0), max is (3.0,4.0,5.0)
Of (3.0,4.0,5.0) and (5.0,3.0,5.0), max is (5.0,3.0,5.0)
Of (3.0,4.0,5.0) and (5.0,4.0,3.0), max is (5.0,4.0,3.0)
Of (3.0,4.0,5.0) and (5.0,4.0,5.0), max is (5.0,4.0,5.0)
ricegf@antares:~/dev/202401/13-java-generics/code_from_slides/max$
```

# Generic Max is Complete

- We can use our generic max method...
  - With Coordinate
  - And Integer
  - And Double
  - And any other type that implements Comparable




{T extends Comparable<T>}

```
public class MaxGeneric {
    public static <T extends Comparable<T>> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }

    public static void main(String[] args) {
        System.out.println(max(3,5) + ", " + max(3.14, 2.78)
            + ", " + max(new Coordinate(3, 4, 5), new Coordinate(6, 7, 8)));
    }
}
```

Any type that implements Comparable to compare to itself goes here!

Summary

# Using Constrained Types

- If a generic type must extend a class and / or implement one or more interfaces...
  - Our generic declaration must *explicitly* state this
  - **`<T extends Product & Sellable>`**
    - Product (the superclass) comes first if required
    - Zero+ interfaces follow separated by &
    - No | exists (think about why)
- If any class type, enum, or interface will do…
  - A simple **`<T>`** will suffice

# A Generic Method: printAll

- printAll prints all of the elements in a List with commas in-between

```
public static <T> void printAll(List<T> list) {
    String sep = "";
    for(var e : list) {System.out.print(sep + e); sep = ", ";}
    System.out.println("");
}
```

  - We *could* replace **`var`** with **`T`**
  - But if we don't, notice that the generic variable **`T`** is used only once

# A Wildcard Method: printAll

- Since we don't need the generic variable T to express the relationships between types, we can use an unbounded* wildcard (`<?>`) instead
  - Delete `<T>` - we don't need to name the generic type
  - Replace `List<T>` with `List<?>`, which accepts a List of *any* type of objects

```
public static void printAll(List<?> list) {
    String sep = "";
    for(var e : list) {System.out.print(sep + e); sep = ", ";}
    System.out.println("");
}
```

* Yes, Java also supports *bounded* wildcards such as `<T extends Comparable<? super T>>`. Aren't you glad we don't need to cover those in *this* class!

# Unbounded Wildcard `List<?>`

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Arrays;

public class UnboundedWildcard {
    public static void printAll(List<?> list) {
        String sep = "";
        for(var e : list) {System.out.print(sep + e); sep = ", ";}
        System.out.println("");
    }
    public static void main(String[] args) {
        LinkedList<String> strings = new LinkedList<>();
        strings.addAll(Arrays.asList("one", "two", "three"));
        printAll(strings);

        ArrayList<Integer> ints = new ArrayList<>();
        ints.addAll(Arrays.asList(42, 17, 255));
        printAll(ints);
    }
}
```

Method printAll prints all elements in a List (the interface used by ArrayList and LinkedList)

Print LinkedList<String>

Print ArrayList<Integer>

```
ricegf@antares:~/dev/202108/19/code_from_slides$ javac UnboundedWildcard.java
ricegf@antares:~/dev/202108/19/code_from_slides$ java UnboundedWildcard
one, two, three
42, 17, 255
ricegf@antares:~/dev/202108/19/code_from_slides$
```

# Another Generic Class Example: A Time-Tagged ArrayList

- Let's create a variation of ArrayList
  - Support any object type (i.e., except primitives)
  - Add, get, clear, size, et. al. methods
  - Add a **`when()`** method to access the date and time each element was added to the container
- Java.util.Date's default constructor creates the current time and date
  - java.text.SimpleDateFormat gives us exceptional control over a date's String representation

# Inheritance or Composition?

- With **inheritance** we get a lot of methods “free”
  - Add an `ArrayList<Date> dates` field
  - Override methods that add or remove elements so that we can *also* add or remove dates
  - Add `Date when(int index)` method to retrieve date and `String toString(int index)` to get formatted object with date
  - RISK: We may miss (or later add) ArrayList methods that manipulate elements and “leak” data
- With **composition** the interface is explicit
  - Create a package private TaggedObject generic class containing object and date fields
  - Create `ArrayList<TaggedObject<E>> list` field for the array
  - Create constructor, add, get, when, toString(int index), and size methods – the minimum that we need


Vote button by amberzen per Pixabay License
https://pixabay.com/illustrations/vote-button-election-elect-1319435/

Which approach shall we take?

# Winner: Both!*

- I recommend **composition** for this project
  - It's more straightforward
  - It offers less risk of "leaks" and problems syncing the two ArrayLists (synchronized data is a disaster waiting for a critical VP demo to happen)
- After composition we'll also implement it with **inheritance** for contrast
  - So you can judge for yourself

* I know, I know, "A tie is like kissing your sister"
– Eddie Erdelatz, Navy Football Coach, 1953

Composition

# TaggedObject Generic Class

- We can avoid sync problems by storing each date and element together
  - We make the fields public for simplicity, since the ArrayList containing these will be private

In ArrayListTimeTaggedGeneric.java

```
import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedObject<E> {
    public TaggedObject(Date date, E value) {
        this.date = date;
        this.value = value;
    }
    public String toString() {
        return "'" + value + "' (at " + formatDate.format(date) + ")";
    }

    public Date date;
    public E value;
    private static SimpleDateFormat formatDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
}
```

Composition

# TaggedArrayList Generic Class

- We implement only the minimum methods required for our testing – **`add`**, **`get`**, **`when`**, and **`size`**
  - Other methods could be added as needed
  - No sync problems – every element MUST have a correct date

```
import java.util.ArrayList;
import java.util.List;

class TaggedArrayList<E> {
    public TaggedArrayList() {list = new ArrayList<>();}

    public void add(E element) {list.add(new TaggedObject<E>(new Date(), element));}
    public E get(int index) {return list.get(index).value;}
    public Date when(int index) {return list.get(index).date;}
    public int size() {return list.size();}

    public String toString(int index) {return list.get(index).toString();}
    private ArrayList<TaggedObject<E>> list;
}
```

# Composition
# A Quick Demo

- Store two strings and print them out with dates
- The enterprising student could readily write interactive test programs – right?
  - Enter strings at a prompt, store them, and list with dates at the end
  - Enter strings, then find a specified string and print its creation date

```
public class ArrayListTimeTaggedGeneric {
    public static void main(String[] args) throws InterruptedException {
        TaggedArrayList<String> list = new TaggedArrayList<>();
        System.out.print("Working...");

        list.add("The answer is ");
        Thread.sleep(1000 + (long)(Math.random() * 5000)); // wait 1-6 seconds
        list.add("forty-two");

        System.out.println("done!");

        for (int i=0; i<list.size(); ++i) {
            System.out.println(list.toString(i));
        }
    }
}
```

```
Working...done!
'The answer is ' (at 2023-04-11 22:04:12)
'forty-two' (at 2023-04-11 22:04:16)
```

# TaggedArrayList Generic Class

- With inheritance, we MUST override every method that affects the data to avoid “leaks”

In ArrayListTimeTaggedInheritance.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Collection;

import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedArrayList<E> extends ArrayList<E> {
    // Shadow all of ArrayList's known constructors
    public TaggedArrayList() {
        super();  // Note that super() must ALWAYS be first
        dates = new ArrayList<>();
    }
    public TaggedArrayList(int initialCapacity) {
        super(initialCapacity);
        dates = new ArrayList<>(initialCapacity);
    }
    public TaggedArrayList(Collection<? extends E> c) {
        super(c);
        dates = new ArrayList<>();
        for(E e : this) dates.add(new Date());
    }
```

Constructors (delegate to ArrayList’s)

Continued...

# TaggedArrayList Generic Class

```
@Override
public boolean addAll(Collection<? extends E> c) {
    checkSync();
    for(E e : c) dates.add(new Date());
    return super.addAll(c);
}
@Override
public void add(int index, E element) {
    checkSync();
    dates.add(index, new Date());
    super.add(index, element);
}
@Override
public boolean add(E element) {
    checkSync();
    dates.add(new Date());
    return super.add(element);
}
@Override
public void clear() {
    checkSync();
    dates.clear();
    super.clear();
}
```

Overridden methods (delegate to ArrayList's and ensure two lists are synced)

```
@Override
public E remove(int index) {
    checkSync();
    dates.remove(index);
    return super.remove(index);
}
@Override
public E set(int index, E element) {
    checkSync();
    dates.set(index, new Date());
    return super.set(index, element);
}
@Override
public void  trimToSize() {
    checkSync();
    dates.trimToSize();
    super.trimToSize();
}
```

Continued...

NOTE: Iterator methods are NOT overridden, as we haven't covered those yet.

# Inheritance

# TaggedArrayList Generic Class

Methods and Fields

```
    // Additional methods to support time tagging

    // When is the getter for the date, similar to get for data
    public Date when(int index) {
        return dates.get(index);
    }
    // This new overload of toString returns the string representation
    //     of both the element AND the time tag
    public String toString(int index) {
        return "'" + this.get(index) + "' (at "
                   + formatDate.format(dates.get(index)) + ")";
    }

    // Private utility method to verify that the data structures are coherent
    private void checkSync() {
        assert super.size() == dates.size();
    }

    // Fields
    private ArrayList<Date> dates;
    private static SimpleDateFormat formatDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
}
```

# Inheritance
# (The Exact Same) Quick Demo

- Whether with composition or inheritance, the interface is identical
  - So we can use the exact same demo!

```
public class ArrayListTimeTaggedGeneric {
    public static void main(String[] args) throws InterruptedException {
        TaggedArrayList<String> list = new TaggedArrayList<>();
        System.out.print("Working...");

        list.add("The answer is ");
        Thread.sleep(1000 + (long)(Math.random() * 5000)); // wait 1-6 seconds
        list.add("forty-two");

        System.out.println("done!");

        for (int i=0; i<list.size(); ++i) {
            System.out.println(list.toString(i));
        }
    }
}
```

```
Working...done!
'The answer is ' (at 2023-04-11 22:09:43)
'forty-two' (at 2023-04-11 22:09:48)
```

# Strings, Ints, and Coordinates (Oh, My!)

- What follows is one simple menu-driven program to collect String, int, and Coordinate values for 3 TaggedArrayList instances
  - This uses the Menu and MenuItem classes from Lecture 10

```
import java.util.Scanner;

public class TestTaggedArrayList {
    public static void main(String[] args) {
        (new TestTaggedArrayList()).mdi();
    }

    private TaggedArrayList<String> strings = new TaggedArrayList<>();
    private TaggedArrayList<Integer> ints = new TaggedArrayList<>();
    private TaggedArrayList<Coordinate> coords = new TaggedArrayList<>();

    private Menu menu = new Menu();
    private Scanner in = new Scanner(System.in);
    private boolean running = true;
    private String discard;
```

Our test TaggedArrayList instances

Continued...

# Strings, Ints, and Coordinates
## (Oh, My!)

```
// Constructor

public TestTaggedArrayList() {
    menu.addMenuItem(new MenuItem("Exit",           () -> exit()));
    menu.addMenuItem(new MenuItem("Add a String",   () -> addString()));
    menu.addMenuItem(new MenuItem("Add an Integer", () -> addInteger()));
    menu.addMenuItem(new MenuItem("Add a Coordinate", () -> addCoordinate()));
}

// Main loop for Menu-Driven Interface (mdi)

public void mdi() {
    while(running) {
        String input = "";
        try {
            System.out.print("\n".repeat(255) + "MAIN MENU\n=========\n\n"
                             + menu + data() + "\nSelection? ");
            input = in.nextLine();
            menu.run(Integer.parseInt(input));
        } catch(Exception e) {
            System.err.println("#### Invalid input: " + input);
        }
    }
}
```

Continued...

# Strings, Ints, and Coordinates
## (Oh, My!)

```
// Format the data for display

private String data() {
    StringBuilder sb = new StringBuilder();
    if(strings.size() > 0) {
        sb.append("\nStrings\n=======\n");
        for(int i=0; i<strings.size(); ++i)
            sb.append("" + strings.when(i) + " '" + strings.get(i) + "'\n");
        sb.append('\n');
    }
    if(ints.size() > 0) {
        sb.append("\nIntegers\n========\n");
        for(int i=0; i<ints.size(); ++i)
            sb.append("" + ints.when(i) + " '" + ints.get(i) + "'\n");
        sb.append('\n');
    }
    if(coords.size() > 0) {
        sb.append("\nCoordinates\n==========\n");
        for(int i=0; i<coords.size(); ++i)
            sb.append("" + coords.when(i) + " '" + coords.get(i) + "'\n");
        sb.append('\n');
    }
    return sb.toString();
}
```

Continued...

# Strings, Ints, and Coordinates
## (Oh, My!)

```
    // Listeners / Observers (called when corresponding menu item is selected)

    private void exit() {
        running = false;
    }
    private void addString() {
        System.out.print("String? ");
        strings.add(in.nextLine());
    }
    private void addInteger() {
        System.out.print("Integer? ");
        ints.add(in.nextInt()); discard = in.nextLine();
    }
    private void addCoordinate() {
        System.out.print("Coordinate (x y z)? ");
        coords.add(new Coordinate(
            in.nextDouble(),
            in.nextDouble(),
            in.nextDouble())); discard = in.nextLine();
    }
}
```

# Strings, Ints, and Coordinates
## (Oh, My!)

```
MAIN MENU
=========

 0] Exit
 1] Add a String
 2] Add an Integer
 3] Add a Coordinate

Strings
=======
Thu Feb 29 15:28:26 CST 2024 'The quick brown fox'
Thu Feb 29 15:28:31 CST 2024 'jumps over'
Thu Feb 29 15:29:02 CST 2024 'the lazy dog'


Integers
========
Thu Feb 29 15:28:34 CST 2024 '42'
Thu Feb 29 15:29:09 CST 2024 '65535'
Thu Feb 29 15:29:23 CST 2024 '255'


Coordinates
===========
Thu Feb 29 15:28:39 CST 2024 '(3.0,4.0,5.0)'
Thu Feb 29 15:28:50 CST 2024 '(11.5,13.5,17.333)'
Thu Feb 29 15:29:18 CST 2024 '(-1.0,0.0,1.0)'


Selection?
```

# Your Thoughts and Observations?

- **Engineers never ask “Which is better?”**
  We ask...
  - “Which is more concise / performant / supportable / backward compatible / secure / portable / ...?”
  - “Which will best fix *this* critical design issue?”
  - “Which is better for our specific needs *this* release?”

## What do YOU think?

(Yes, I **will** call on you if I have to! )

# Generics with 2 Types
# HashMap & TreeMap

- **HashMap** is a collection of key-value pairs,
  **TreeMap** is the same but sorted by key
  - Essentially an ArrayList of objects with (almost) any type as the key (index)
- As with TreeSet, a Comparator implementation may be provided for TreeMap's constructor for custom key sorting
- If YOU wrote the class being used as the key (index), **you *must* define its equals() and hashCode() methods**
  - See Lecture 12 for help

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/HashMap.html
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/TreeMap.html

# Map, SortedMap Collection with Comparator
# HashMap & TreeMap Example

- Let's store vehicles' make (String) and model year (Integer)
  - We'll use a HashMap (unsorted) and a TreeMap (default sort)
    - We'll also use a Comparator below with TreeMap (custom sort)
      - We sort first by length of the make String
      - If equal, then we delegate back to String.compareTo
- BEWARE: If your compare method return 0 ("equals") then the new element will *overwrite* an existing Map entry!
  - Hence the two-level compare

MapExample.java

```
import java.util.HashMap;
import java.util.TreeMap;
import java.util.Scanner;
import java.util.Comparator;

class SortByLength implements Comparator<String> {
    public int compare(String lhs, String rhs) {
        int key = lhs.length() - rhs.length();
        if (key == 0) key = lhs.compareTo(rhs);
        return key;
    }
}
```

Note: A default Comparator for a custom class is usually implemented as part of that class.

SortByLength is different from String's default Comparator.

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Comparator.html

# Map, SortedMap Collection with Comparator
# HashMap & TreeMap Example

```
public class MapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> cars = new HashMap<>();
        TreeMap<String, Integer> sortedCars = new TreeMap<>();
        TreeMap<String, Integer> lenSortCars = new TreeMap<>(new SortByLength());
        Scanner in = new Scanner(System.in);

        while(true) {
            System.out.print("Enter model year (0 to exit) and vehicle: ");
            //System.out.flush(); // text only output on \n on some systems (not Linux)
            int year = in.nextInt(); if(year == 0) break;
            String vehicle = in.nextLine();
            cars.put(vehicle, year);
            sortedCars.put(vehicle, year);
            lenSortCars.put(vehicle, year);
        }

        System.out.println("\n\nCars: ");
        for(String key : cars.keySet())
            System.out.println("  " + cars.get(key) + " " + key);
        System.out.println("\nSorted: ");
        for(String key : sortedCars.keySet())
            System.out.println("  " + sortedCars.get(key) + " " + key);
        System.out.println("\nSorted by Length: ");
        for(String key : lenSortCars.keySet())
            System.out.println("  " + lenSortCars.get(key) + " " + key);

    }
}
```

The vehicle's make (String) is the key, year (Integer) is the value

An object implementing Comparator is the constructor parameter here

Add pairs to each of the 3 maps (notice the method is "put" not "add")

Print out all pairs in key order (notice that "keySet()" returns the keys)

lenSortCars is using our custom Comparator and thus sorts the keys in a custom order

# Map, SortedMap Collection with Comparator
# HashMap & TreeMap Example

```
ricegf@antares:~/dev/202108/20$ java MapExample
Enter model year (0 to exit) and vehicle: 2020 Tesla Model 3
Enter model year (0 to exit) and vehicle: 2015 Ford F-150
Enter model year (0 to exit) and vehicle: 2004 Ford Mustange Convertible
Enter model year (0 to exit) and vehicle: 2018 Chevy Volt
Enter model year (0 to exit) and vehicle: 0


Cars:
  2020  Tesla Model 3
  2018  Chevy Volt
  2015  Ford F-150
  2004  Ford Mustange Convertible

Sorted:
  2018  Chevy Volt
  2015  Ford F-150
  2004  Ford Mustange Convertible
  2020  Tesla Model 3

Sorted by Length:
  2018  Chevy Volt
  2015  Ford F-150
  2020  Tesla Model 3
  2004  Ford Mustange Convertible
```

**Yes, it should be Mustang not Mustange!**

**Per SortByLength Comparator object, vehicles in lenSortCars are sorted by String *length* first, then by String**

# Map, SortedMap Collection with Comparator another equals & hashCode example

EqualsAndHashCode.java

For this example class, assume Coordinate has already defined equals() and hashCode() (see code on cse1325-prof/22)

```
class Treasure {
    public Treasure(Coordinate c, String name, double value) {
        this.coordinate = c;
        this.treasureName = name;
        this.treasureValue = value;
    }
    @Override
    public boolean equals(Object o) {
        if(this == o) return true;
        if(o == null || this.getClass() != o.getClass()) return false;
        Treasure t = (Treasure) o; // Downcast to a Treasure
        return coordinate.equals(t.coordinate)
            && treasureName.equals(t.treasureName)
            && (treasureValue == t.treasureValue);
    }
    @Override
    public int hashCode() {
        return Objects.hash(coordinate, treasureName, treasureValue);
    }
    private Coordinate coordinate;  // Our custom class (Roving Robots, enhanced)
    private String treasureName;    // A JCL class
    private double treasureValue;   // A primitive
}
```

# Map, SortedMap Collection with Comparator
# another equals & hashCode example

```java
public class EqualsAndHashCode {
    public static void main(String[] args) {
        HashMap<Treasure, Integer> map = new HashMap<>();
        map.put(new Treasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 1);
        map.put(new Treasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 0);
        System.out.println("This should be 1: " + map.size());
    }
}
```

For HashMap, we add two key-value pairs with separate key objects that are .equals. Thus, the second put should *overwrite* the first, leaving map size as 1.

```
ricegf@antares:~/dev/202301/23/code_from_slides$ javac EqualsAndHashCode.java
ricegf@antares:~/dev/202301/23/code_from_slides$ java EqualsAndHashCode
This should be 1: 1
ricegf@antares:~/dev/202301/23/code_from_slides$
```

# Generics Summary

- We've implemented several algorithms and collections (max, sorted array, time-tagged array list) independent of the types to which the algorithms should apply (generic programming) using Java generics
- Potentially reusable algorithms that are suitable should generally be defined as generic classes
  - The overhead in programmer time and execution resources is very small
  - The potential reuse benefits are significant