

## **MODULE 06: POLYMORPHISM**

**Professor :** Dave Houtman

**Office:** T323

**Office Hrs:** Wednesday 14:15 – 15:30  
Wednesday (after lecture\*)

\* confirm beforehand

**Email:** [houtmad@algonquincollege.com](mailto:houtmad@algonquincollege.com)

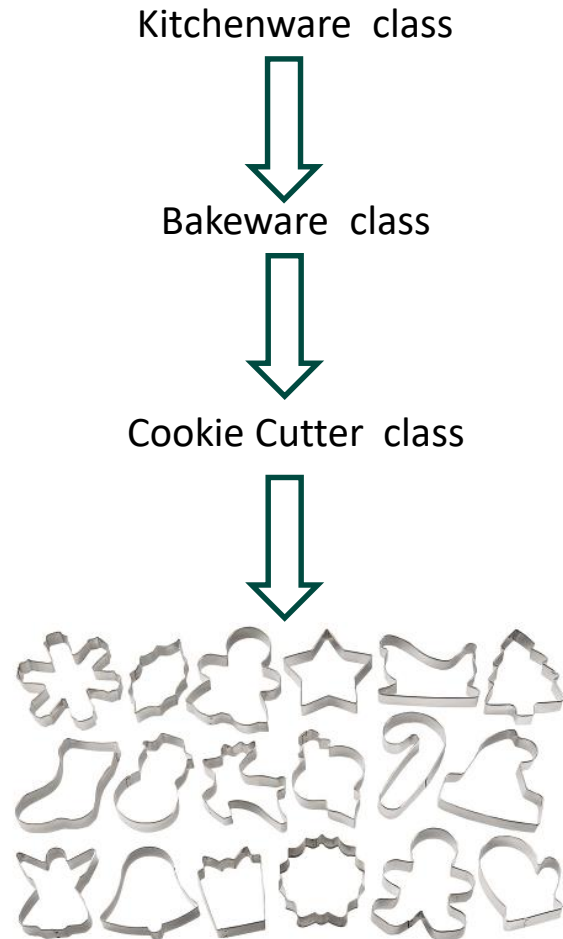
## 6.0 Polymorphism

Recall from Module 01 that inheritance and abstraction allow for the creation of **class hierarchies**, in which classes near the top of the hierarchy tend to be general, abstract and have few members, while those near the bottom tend to be concrete classes more richly-endowed with specialized methods.

**Polymorphism** is a feature which allows an object of a subclass to be treated *as if* it was a member of the superclass in the same class hierarchy due to the compositional (i.e. *is a*) relationship between derived objects. For example, any object derived from a `Cookie` *is a* member of the `Cookie` class, *and* the `Bakeware` class, *and* the `Kitchenware` class—and ultimately, the `Object` class.

D&D 10.2

D&D 10.3



*SnowflakeCookie class,  
GingerbreadManCookie class, etc.*



## 6.0 Polymorphism

For example, consider the case in which we wish to store objects in an array declared as being of type `SnowflakeCookie`. Then, only objects instantiated directly from `new SnowflakeCookie()` will fit into the array. But if the array is declared as the superclass `Cookie`, then any subclass type of `Cookie` can be stored.



So if `CookieCutter` is a superclass (abstract or concrete) and we declare an array:

```
CookieCutter[] cookieBox = new SomeCookie[35];
```

then any object derived from the `Cookie` superclass, i.e.

```
public class SnowflakeCookie extends Cookie{...}
```

can be stored into the `cookieBox` array, without casting:

```
CookieCutter[0] = new SnowflakeCookie();
```

Polymorphism allows us to do this, even though the data types on either side of the equals sign are different.



## 6.0 Polymorphism

*Polymorphism means that a variable of a superclass type can be used to store or pass a subclass object.*

For example, in the previous module, `Circle` and `Rectangle` were both declared as subclasses of `GeometricObject` via the declarations

```
public class Circle extends GeometricObject {...}
```

and

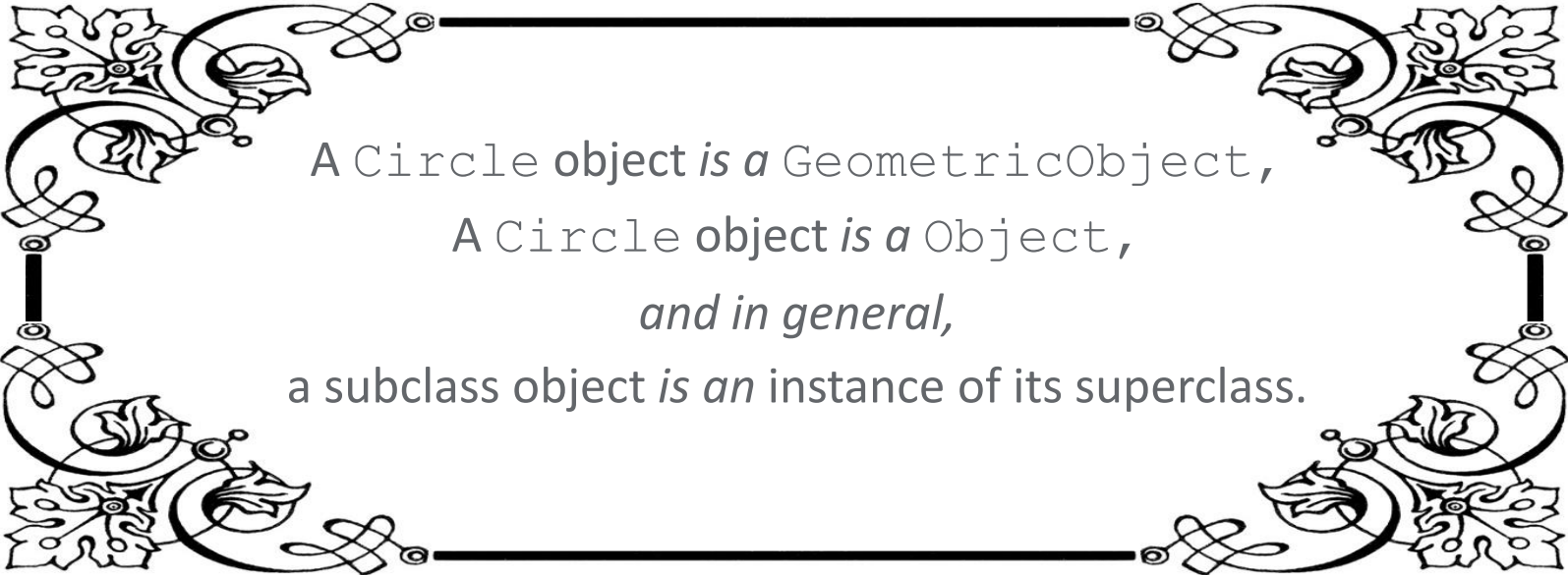
```
public class Rectangle extends GeometricObject {...}
```



## 6.0 Polymorphism

*Polymorphism means that a variable of a superclass type can be used to store or pass a variable of one of its subclasses.*

*And thus,*



A Circle object *is a* GeometricObject,  
A Circle object *is a* Object,  
*and in general,*  
a subclass object *is an* instance of its superclass.



## 6.0 Polymorphism

Recall from Module 04 that we declared classes `Circle` and `Rectangle` that extended `GeometricObject`. Every instance of a `GeometricObject` subclass is also an instance of its superclass. So if we have a method that takes as a formal parameter a `GeometricObject`:

```
public void displayObject(GeometricObject object) {...}
```

Then we can then pass *any of its subclasses* into this method as the actual parameter, provided it is a subclass in the `GeometricObject`. So the above method can be used with, for example, a `Circle` object:

```
Circle circ = new Circle();  
...  
displayObject(circ);
```

Thus while the *formal* parameter type is a `GeometricObject`, the *actual* type passed is a subclass, `Circle`.

Liang 11.7

D&D 10.2

D&D 10.3



## 6.0 Polymorphism

Similarly, as we saw earlier (with the `CookieCutter` class) an array of objects of the superclass type may be used to store objects of the subclass type. So

```
GeometricObject[] geoObjArray = new GeometricObject[3];  
...  
geoObjArray[0] = new Rectangle(3,12);  
geoObjArray[1] = new Circle(3);
```

We can then print out the area of each `GeometricObject` using:

```
for (GeometricObject geoObjElement: geoObjArray)  
    System.out.println("Area = " + geoObjElement.getArea());
```



## 6.0 Polymorphism

Alternately, since every object extends from the `Object` class, we could use `Object` in place of `GeometricObject` in the declaration above (as we did when discussing the `instanceof` operation).

```
Object[] objArray = new GeometricObject[3];

...
objArray[0] = new Rectangle(3,12);
objArray[1] = new Circle(3);

for (Object objElement: objArray)
    System.out.println("Area = " +
        ((GeometricObject) objElement).getArea());
```

But then any kind of object could be used in the `for` loop...until we tried to get the area of, say, a `Scanner`. This is prevented by the correct use of types (using `GeometricObject` rather than `Object`), and by the use of generic types, to which we shall return shortly.



## 6.0 Polymorphism

Examples of *is a* relationships (and polymorphism) can be found in `java.lang` and `java.util`. For example:

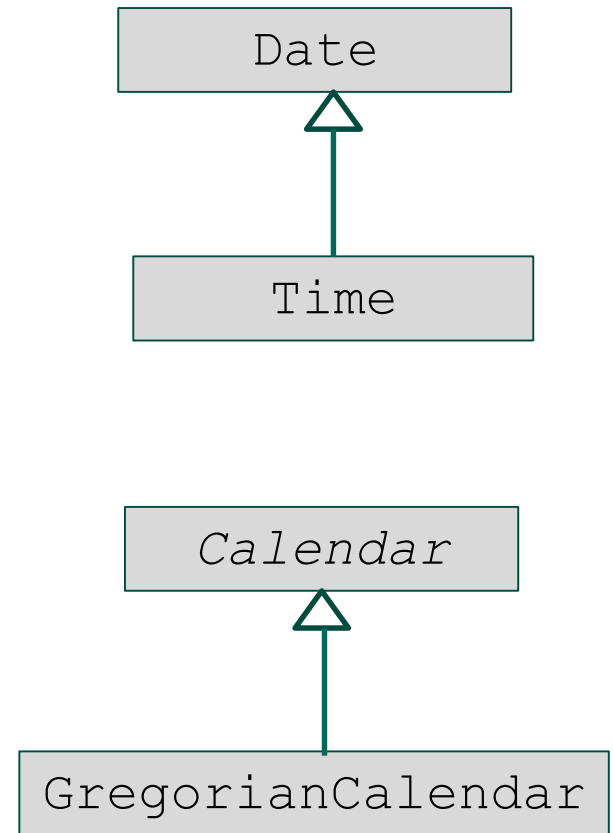
```
public class Time extends Date
```

So a `Time` object *is a* `Date` object as well.

```
public class GregorianCalendar  
    extends Calendar
```

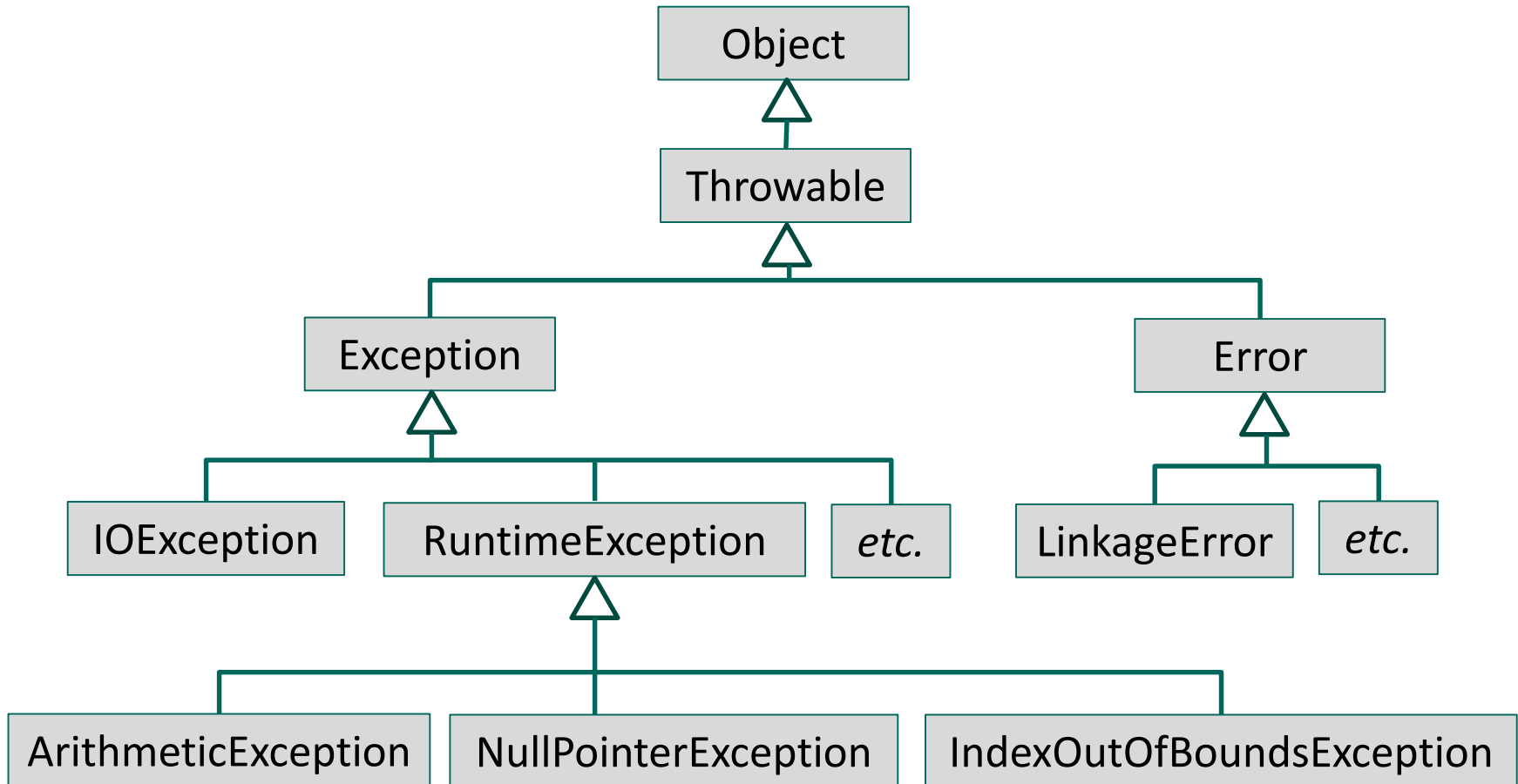
So a `GregorianCalendar` object *is a* `Calendar` object as well. We can write:

```
Calendar calendar =  
    new GregorianCalendar();
```



# Questions

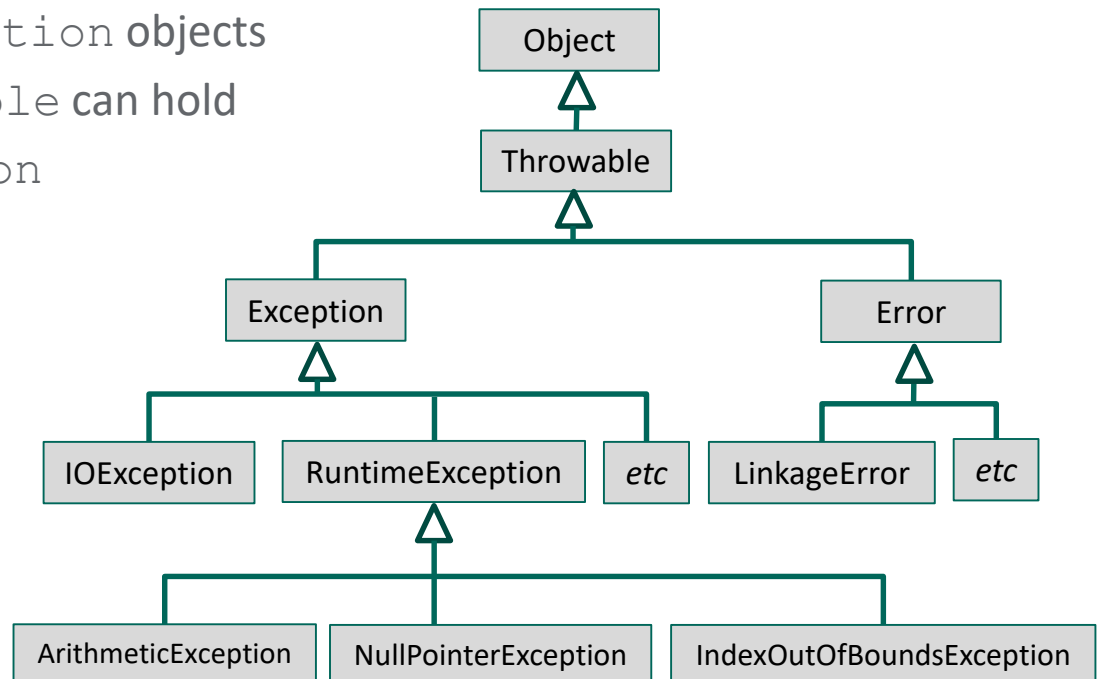
The `Throwable` class has a rich object hierarchy:



# Questions:

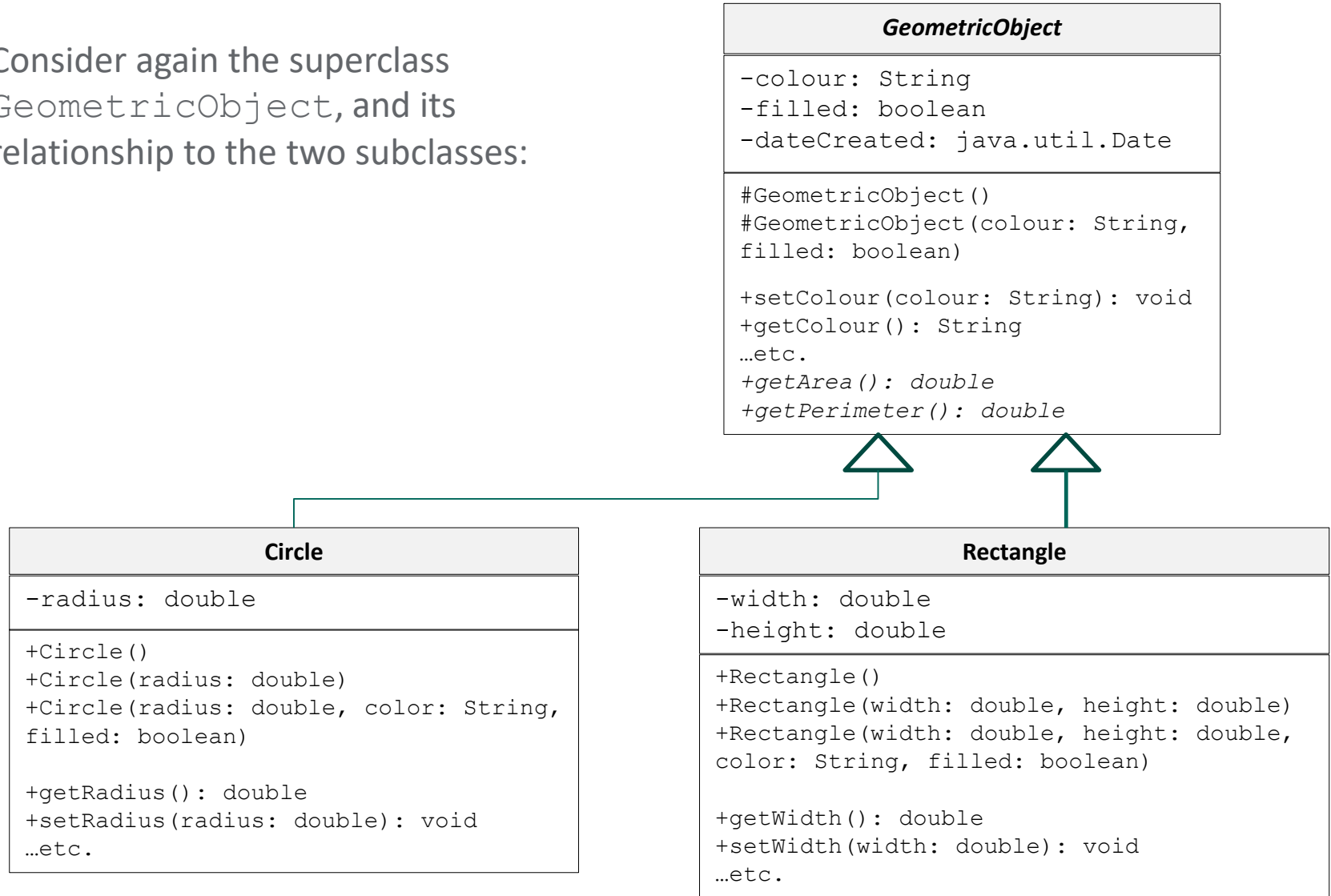
Answer true or false for each of the following:

1. An `IOException` object is an instance of `Throwable`
2. A `LinkageError` object is an instance of `Error`
3. A `LinkageError` object is an instance of `Exception`
4. An array of type `RuntimeException` can hold `ArithmeticException` objects
5. An array of type `Throwable` can hold `NullPointerException` objects



# 6.1 Polymorphism in action

Consider again the superclass `GeometricObject`, and its relationship to the two subclasses:



# 6.1 Polymorphism in action

Recall that the `GeometricObject` declared two abstract methods...

```
public abstract class GeometricObject
    private String colour = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        dateCreated =new java.util.Date();
    }

    protected GeometricObject(
        String colour, boolean filled)
        dateCreated =new java.util.Date();
        this.colour = colour;
        this.filled = filled;
    }

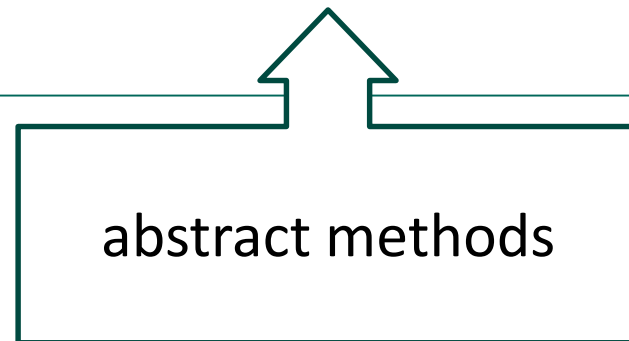
    public void setColour(String colour){
        return colour;
    }
```

```
public String getColour(){
    return colour;
}

...etc.

public abstract double getArea();

public abstract double getPerimeter();
}
```



## 6.1 Polymorphism in action

...and because they were abstract, they *had* to be overridden in their subclasses:

```
public class Rectangle
    extends GeometricObject{
    ...
    public double getPerimeter(){
        return (2 * (width + height));
    }

    public double getArea(){
        return (width * height);
    }
}
```

```
public class Circle
    extends GeometricObject{
    ...
    public double getPerimeter(){
        return (2 * Math.PI * radius);
    }

    public double getArea(){
        return (Math.PI * radius * radius);
    }
}
```

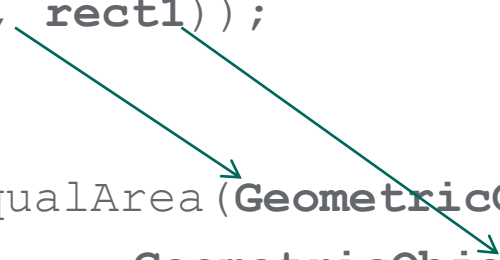


## 6.1 Polymorphism in action

To actually put these classes to use, we'll need a test class in `main()`:

```
public static TestGeometricObject {
    public static void main(String[] args){
        GeometricObject circ1 = new Circle(5);
        GeometricObject rect1 = new Rectangle(5, 3);
        System.out("Same area? " +
            equalArea(circ1, rect1));
    }

    private static boolean equalArea(GeometricObject obj1,
        GeometricObject obj2) {
        return obj1.getArea() == obj2.getArea();
    }
}
```



## 6.1 Polymorphism in action

Polymorphism allows us to pass the two instantiated objects as special subclass objects of the `GeometricObject` type. If we'd declared the `equalArea()` method with two *specific* parameters, say a `Circle` and a `Rectangle`, like this...

```
...
private static boolean equalArea(Circle obj1,
                                  Rectangle obj2) {
    return (obj1.getArea() == obj2.getArea());
}
}
```

...the code still works (since both object have a `getArea()` method) but then we could *only* pass a `Circle` object *and* a `Rectangle` object into our method (in that order)...*and nothing else*.

Polymorphism allows us to reuse the same code using generalized parameters, i.e. it promotes *code reuse*.



## 6.1 Polymorphism in action

Declaring the `equalArea()` method with the two superclass parameters ensures that we can pass *any* `GeometricObject`-derived subclass object into this method, e.g.

```
public class Circle extends GeometricObject {...}
public class Rectangle extends GeometricObject {...}
public class Ellipse extends GeometricObject {...}
public class Triangle extends GeometricObject {...}
public class Polygon extends GeometricObject {...}
```

```
System.out("Same area? " +
    equalArea(new Circle(3), new Triangle (3, 4, 5)));
System.out("Same area? " +
    equalArea(new Ellipse(12,2), new Polygon()));
System.out("Same area? " +
    equalArea(new Ellipse(5, 12), new Ellipse(12, 5)));
```



## 6.2 Generic Methods and Polymorphism

As we've already seen, generic methods can be used inside classes to help eliminate the need for overloading. Similarly, we can eliminate the need for `instanceOf()` by the thoughtful use of generics:

```
public class BoundedTypeDemo {
    public static <T extends GeometricObject>
        boolean equalArea ( T obj1, T obj2 ) {
        return (obj1.getArea() == obj2.getArea());
    }

    public static void main(String[] args) {
        Rectangle rect = new Rectangle(2,2);
        Circle circle = new Circle(2);

        System.out.println("Same area? " +
            = equalArea(rect, circle));
    }
}
```

This example is taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed.* Toronto, ON: Pearson. pg. 743



## 6.2 Generic Methods and Polymorphism

Note that the format of the generic type is the same as we saw in Module 05:

```
public static <T extends GeometricObject>
    boolean equalArea ( T obj1, T obj2) {
```

Formal Type

How its going to be used

Here, the generic type is said to be **bounded**. That is, `T` is bounded to being either a `GeometricObject` or one of its subclasses.

As with all generic declarations, this helps ensure that we catch errors at compile-time rather than run-time—which is what happens if we use `instanceof`.

Therefore,

```
<T extends GeometricObject>
```

specifies that `equalArea`'s two parameters *must* be instance's of the `GeometricObject` class.



## 6.2 Generic Methods and Polymorphism

Note that whenever a formal type, say  $T$ , appears inside the  $\langle \rangle$ , this is to alert the compiler that  $T$  stands for the class used to parameterize the class. If  $T$  is unbounded, then any class can be used in an expression. So, for example, the definition of `ArrayList` is:

```
ArrayList<T> list = new ArrayList<>();
```

This effectively says that `list` can be an `ArrayList` of *any* type.



## 6.2 Generic Methods and Polymorphism

You've seen bounded types in use in an earlier example. The declaration for an `EventHandler` is parameterized to only use `Event` and its subclasses:

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP JavaFX 8

**PREV CLASS** **NEXT CLASS** FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD    DETAIL: FIELD | CONSTR | METHOD

javafx.event

### Interface `EventHandler<T extends Event>`

**Type Parameters:**  
T - the event class this handler can handle

**All Superinterfaces:**  
`EventListener`

---

```
@FunctionalInterface
public interface EventHandler<T extends Event>
    extends EventListener
```

Handler for events of a specific class / type.

---



## 6.2 Generic Methods and Polymorphism

Thus the `EventHandler` may *only* be parameterized with `Event` objects, where the event type `T` is passed as to the `handler()` method as shown below:

**Method Summary**

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
void	<b>handle</b> ( <code>T event</code> ) Invoked when a specific event of the type for which this handler is registered happens.	

Note that the declaration of `handle()` does not need to be written as:

```
void handle(<T extends Event> event)
```



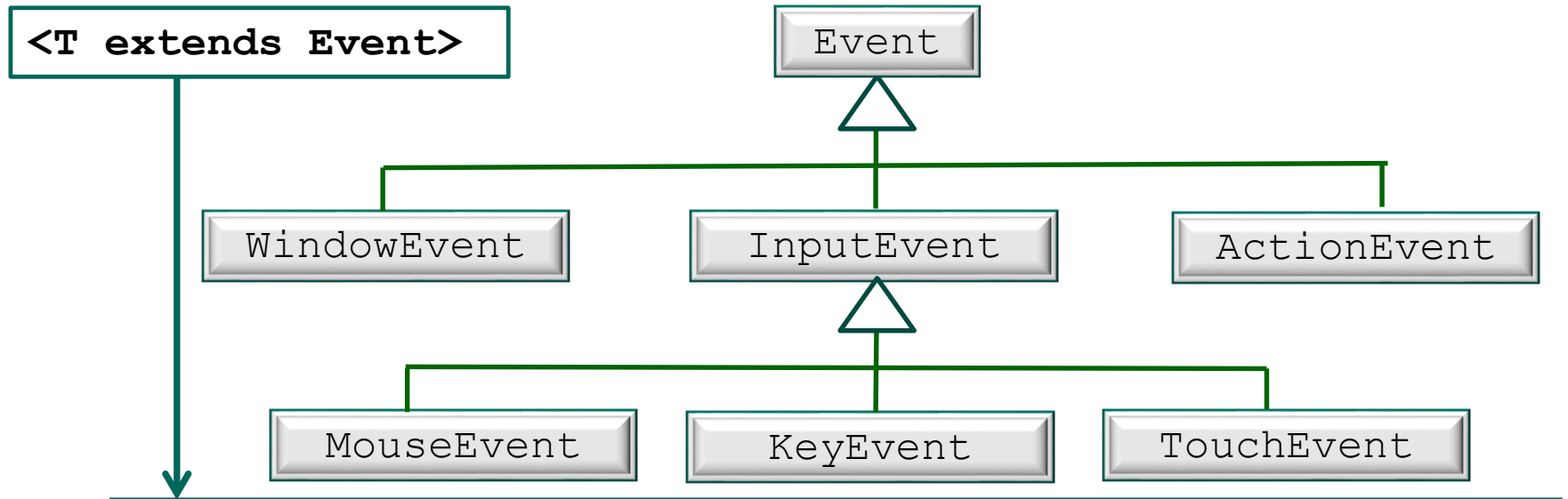
**T** already bounded

since `T` has already been bounded in the class declaration as a subclass of `Event`.



## 6.2 Generic Methods and Polymorphism

Finally, recall again the `Event` object hierarchy:



Thus an `Event`, or any of its subclasses (including `ActionEvent`) can be passed to the `EventHandler`; the notation `<T extends Event>` guarantees that only objects that are polymorphically related to `Event` as subclasses can be passed to the `EventHandler` at compile-time.



## 6.2 Generic Methods and Polymorphism

Consider the following example, which uses an `ArrayList` as a **stack** of objects of type `E`. A stack is a type of **data structure**, a specialized program designed to organize and manage data. A typical data structure consists of both an array of some type, and a set of related methods designed to manipulate this information.

The standard analogy for a stack is of a stack of cafeteria plates. Plates may be **push**ed onto the top of the stack, or **pop**ped off the top. (Other operations, like removing a plate from the middle of the pile without disrupting the order, require more sophisticated data structures.)



Liang 19.3

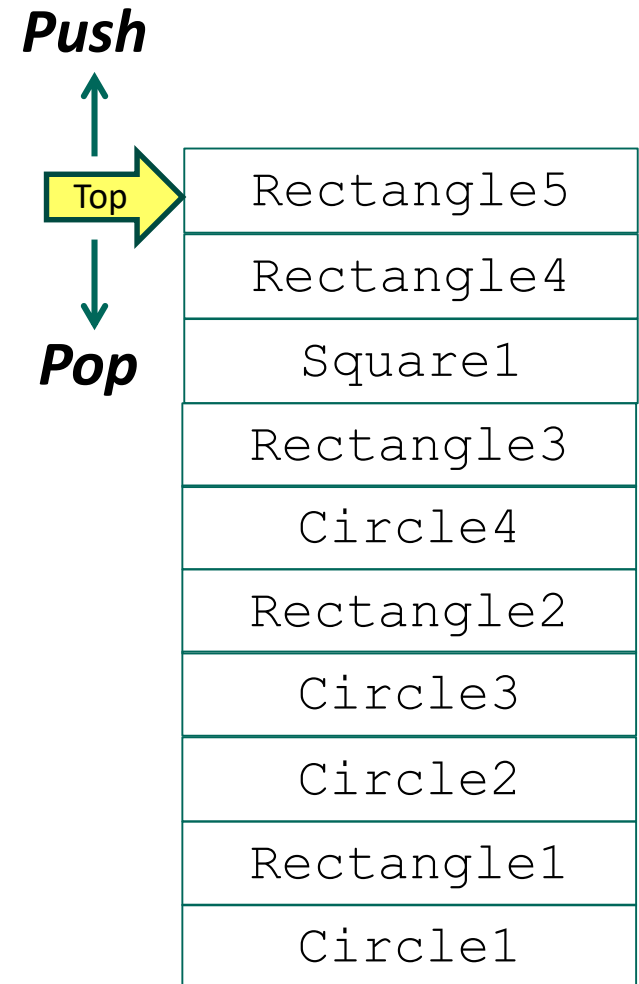
D&D 20.6



## 6.2 Generic Methods and Polymorphism

Each time we push an object on to the stack, the stack grows; each time we pop an object off the stack, the stack shrinks. The yellow arrow at right indicates the top of the stack.

The code below\* uses an `ArrayList` to support a generic stack of objects.



\* Taken, with modifications, from Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed.* Toronto, ON: Pearson. pp. 741-742.

## 6.2 Generic Methods and Polymorphism

```
import java.util.ArrayList;

public class GenericStack<E> {
    private ArrayList<E> list = new ArrayList<>();

    public void push(E o) {
        list.add(o);
    }

    public E pop() {
        E o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Boolean isEmpty() {
        return list.isEmpty();
    }
}
```



## 6.2 Generic Methods and Polymorphism

```
//... continued

public int getSize() {
    return list.size();
}

public E getTop() {
    return list.get(getSize() - 1);
}

@Override
public String toString() {
    return "Stack: " + list.toString();
}
}
```



## 6.2 Generic Methods and Polymorphism

We can parameterize a `GenericStack` with the `String` type by calling

```
GenericStack<String> stack1 = new GenericStack<>();
```

and then adding `String` objects to the stack

```
stack1.push("London");  
stack1.push("Paris");  
stack1.push("Rome");
```



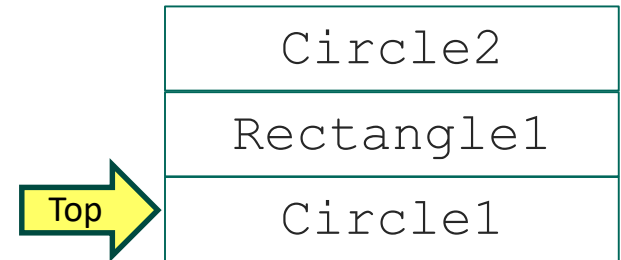
## 6.2 Generic Methods and Polymorphism

We can parameterize a `GenericStack` with `GeometricObjects` by calling

```
GenericStack<GeometricObject> stack2 =  
    new GenericStack<>();
```

and then adding `GeometricObjects` to the stack

```
stack2.push(Circle2);  
stack2.push(Rectangle1);  
stack2.push(Circle1);
```



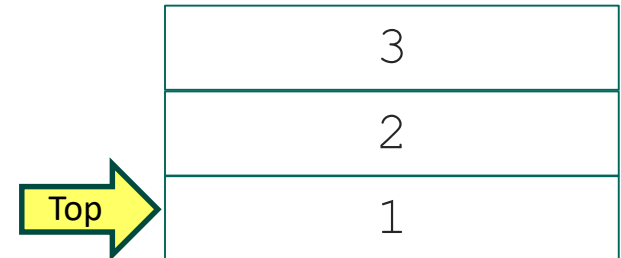
## 6.2 Generic Methods and Polymorphism

We can even use primitive numeric types like `ints` in our stack, provided we parameterize it with its numeric object equivalent:

```
GenericStack<Integer> stack3 =  
    new GenericStack<>();
```

and then adding `GeometricObjects` to the stack

```
stack3.push(1);  
stack3.push(2);  
stack3.push(3);
```



This feature, in which Java automatically wraps a primitive data type in its class equivalent, is called **autoboxing**. Thus a primitive `int` is converted to an `Integer`, a `double` is converted to a `Double`, and a `boolean` is converted to `Boolean`. (Note that `Integer`, `Double`, `Byte`, `Float`, `Boolean`, `Long`...etc. are subclasses of the `Number` type)



## 6.2 Generic Methods and Polymorphism

Generic classes *may* be used without specifying a type. For example, the `GenericStack` may be called using

```
GenericStack stack = new GenericStack();
```

This is equivalent to

```
GenericStack<Object> stack = new GenericStack<>();
```

This is basically saying that we have a `GenericStack` of any type, since every type is a subclass of `Object`.

When used in this way, without a specific type, a generic class is called a **raw type**. Raw types allow for backward compatibility with earlier versions of Java.

Whenever possible you should *avoid using raw types*, since they are considered *unsafe*. Always use parameterized types, wherever possible.

Liang 19.6

D&D 20.7

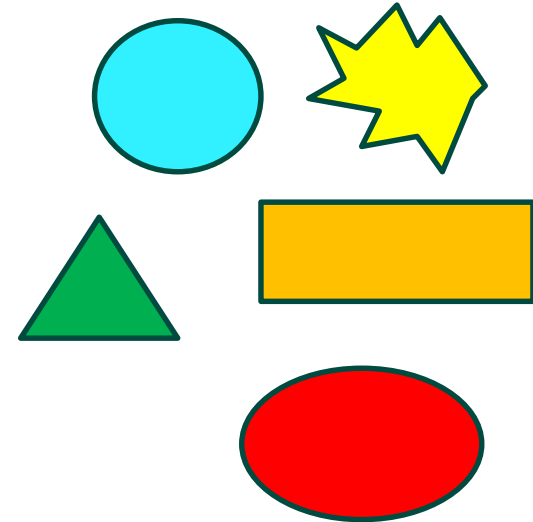


## 6.3 Dynamic Binding

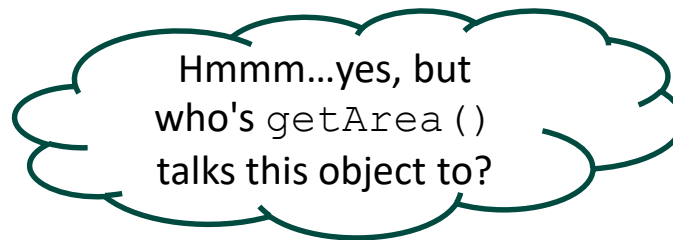


Prior to execution, the JVM can't know exactly which version of a method will be called upon when a program is running, since this may vary according to the user's selection at run-time.

```
Object obj =  
    new GeometricObject();  
  
...  
  
System.out.println(obj.getArea());
```



Liang 11.8



## 6.3 Dynamic Binding



**Dynamic Binding** allows the JVM to 'bind' (or load) the object and its members at runtime. Given a hierarchy of inherited classes...



...the JVM starts with the lowest subclass in the chain and works upward until it finds the first instance of the method.



# 6.3 Dynamic Binding



So given the following hierarchy:



and code to override `toString()` in each class...

```
// MotorVehicle's toString()
public String toString(){
    StringBuilder sb
        = new StringBuilder();
    sb.append(super.toString())
        .append("manufacturer: "...
    ...
}
```

```
// Car's toString()

public String toString(){
    StringBuilder sb = ...
    sb.append(super.toString())
        .append("trunk capacity..."
    ...
}
```

```
// Object's toString()
public String toString(){
    //Output Car@F320
    ...
}
```

...the JVM starts with `Car's toString()` method. Thus the lowest subclass's methods get bound (and executed) first

