final variables
Declaring a primitive type as final means once it is given a value its value cannot change.

```
final int BEARD_LENGTH = 10000;
```

Declaring an object type (a reference type) as final means that once the reference is assigned an object, we cannot change what the reference "points" to. However, you may change the value of the object that the reference "points" to (as long as its not immutable (its value cannot be changes)).

```
final NinjaCat lionArdo = new NinjaCat(16, "katana");
lionArdo.setAge(17); // this is allowed
lionArdo = new NinjaCat(12, "spoon"); // Not allowed (compile time error) because lionArdo is final
```

Inheritance and Polymorphism

Sometimes on data type is a special kind of another data type.

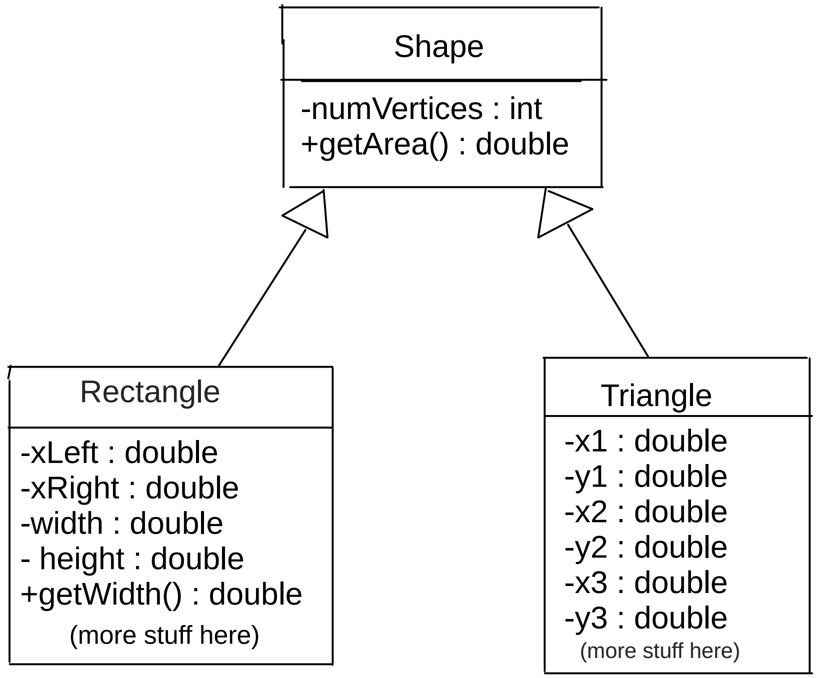
Ex:
(1) Both a Rectangle and a Triangle are special kinds of Shapes.
(2) A Monkey is a special kind of Animal. But not all Animals are Monkeys.

In (1) every data field and method (except constructors) of Shape should also be a data field and method of Rectangle and Triangle (but not necessarily the other way around). To do this, class Rectangle and Triangle will extend class Shape.

```
public class Shape
{
   private int numVertices;

   // ...
}

public class Rectangle extends Shape
{
   // Every data field and method (except constructors) from Shape are included in Rectangle (so don't repeat them).

   private double xLeft;
   private double yUpper;
   private double width;
   private double height;

   // ...
}

public class Triangle extends Shape
{
   // Every data field and method (except constructors) from Shape are included in Triangle (so don't repeat them).
   // ...
}
```

We say that Rectangle and Triangle are subclasses/subtypes/child classes of Shape.
We say that Shape is a superclass/supertype/parent class of Rectangle and Triangle.

We can visualize the relationship between class Rectangle, Triangle, and Shape with a UML diagram:

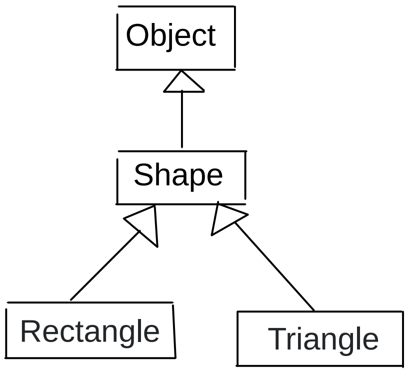


- means private, + means public, # is protected (we haven't gone over protected yet)

We say that a member of a superclass in inherited in a subclass if it is included in the subclass from the superclass.

In code (and in UML diagrams) we do not repeat inherited members in a subclass. They are automatically included.

Every class you create in Java is a subclass of class Object (Object is in package java.lang).
Class Object has no parent class.



Class Object contains the method
  public String toString()
This method returns a String representation of the invoking object. It is called when we display an object to the screen with System.out.println(...);

Every class will have a toString() method in Java (because every class is a subtype of Object).

When creating a class, if we want to specify how instances of the class are represented as a String, then we will need to write our own version of the toString() method in our class.

To write our own implementation (version) of toString() in our subclass, we override it with the @Override annotation in Java. @Override is placed before the method in the subclass.

```
public class Rectangle extends Shape
{
   private double xLeft;
   // ...

   @Override
   public String toString()
   {
       return "Rectangle: upper left = (" + xLeft + ", " + yUpper + "), width = " + width +
                  ", height = " + height;
   }
}

public class Triangle extends Shape
{
    private double x1;
    private double y1;
    // ...

    @Override
    public String toString()
    {
        return "Triangle: (" + x1 + ", " + y1 + "), (" + x2 + "," + y2 + "), (" + x3 + ", " + y3 + ")";
    }
}
```

An instance of a subclass can be used wherever a variable of is superclass is given.
This is what is known as (subtype) polymorphism (from Greek meaning "many forms").

```
Shape s = new Rectangle(2.1, 3.5, 8, 5);
Shape t = new Triangle(2, 4, 7, 4, 7, 1);
```

In Java:

```
   Shape a = new Rectangle(2.1, 3.5, 8, 5);
   // ...
   a = new Triangle(2, 4, 7, 4, 7 ,1);
```

Example:

```
public class Test
{
   public static void sayMe(Object ob)
   {
       System.out.println("Object says: " + ob);
       // Same as doing:
       //    System.out.println("Object says: " + ob.toString());
   }

   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(2.1, 3.5, 8, 5);
      Triangle t = new Triangle(2, 4, 7, 4, 7 ,1);

      sayMe(r);
      sayMe(t);
   }
}
```

Which version of the toString() method to call in sayMe is determined at runtime by the data type of the inoking object.

Java determines which implementation (version) of a non-static method to run based on the data type of the invoking object. Java chooses the most specific implementation to run. This is a kind of dynamic binding (called dynamic dispatch).

Java only uses the data type of the invoking object and not the data type of the other arguments to determine which method implementation to call. This type of dynamic binding is called single dispatch (or message passing).

Dynamic binding that uses more than one argument to determine which method implementation to run is called multiple dispatch. Methods that do this are called multimethods.

Java does not have multimethods.
Examples of a language that has multimethods: Common Lisp

Notes:
- Static methods cannot be overridden because they do not have an invoking object to dispatch on.
- Constructors cannot be overridden because they are not inherited.

Do not confuse method overriding with method overloading.
Method overloading means having multiple methods with the same name but each with a different signature.
Method overriding means to provide a new implementation of an inherited method in a subclass.

Preventing overriding and extending

To prevent a method from being overridden in subclasses, we decare it final in the superclass.

```
public class ManlyPerson
{
   public final void grunt()
   {
      // ...
   }
}

public class MyDad extends ManlyPerson
{
   // cannot override method grunt() in here because grunt() is final.
}
```

To prevent a classs from being extended, declare it final.

```
public final class Caveman extends ManlyPerson
{
   // ...
}

// Error: you cannot extend Caveman because it is final
public class IntelligentCaveMan extends Caveman
{
   // ...
}
```

Abstract Classes

A Rectangle and a Triangle are both Shapes.
But a Shape object cannot be a Shape without being some more specific type of Shape (we know this from geometry). In other words, class Shape should not provide an implementation of its getArea() method.
Class Shape should only provide the declaration of getArea() (no body for getArea() in Shape).

To do this, we declare getArea() abstract in class Shape and do not give it a body. An abstract method is meant to be overridden in subclasses.

If a classs has an abstract method, then the class itself must als be abstract.

```
public abstract class Shape
{
    public abstract double getArea();  // no body for abstract methods, and end it with ;
}
```

```
public class Rectangle extends Shape
{
   // ...

   @Override
   public double getArea() // must override getArea() because it is abstract and class Rectangle is not abstract
   {
      return width*height;
   }
}
```

A non-abstract class must implement all of its inherited abstract methods. If the subclasss is also abstract, then it does not need to override its inherited abstract methods.

```
public abstract Polygon extends Shape
{
   // getArea() does not neeed to be overridden in here since Polygon is abstract
}
```

Making a class C abstract means you cannnot instantiate it (i.e., you cannot use the new operator on C to create an instance of C).
We cannot do:

```
   Shape s = new Shape(14); // Error: Shape is abstract
```

But we can do:

```
   Shape s = new Rectangle(1, 2, 3, 4);
   double a = s.getArea(); // calls Rectangle's getArea() method on s.
```

Constructors and static methods cannot be declared abstract
- Constructors are not inherited, so an abstract constructor could never be implemented.
- Static methods do no have an invoking object. Abstract methods are meant to be overridden to dispatch on the invoking object.

protected access

Suppose we have a class member that we would like subclasses to access, but we want to prevent all other classes not in the same package from accessing it. To do this, we set the access modifier of our member to protected.

```
public class GiantBanana
{
   // ...

   proted void ripen() // can be accessed in subclasses, but not by code outside the package not in subclasses
   {
      // ...
   }
}
```

Who has access

	Inside class	All other classes in same package	subclasses not in same package	non-subclasses not in same package
private	Yes	No	No	No
package (no access modifier)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Keyword super

In a subclass the keyword super refers to its superclass.
It can be used to
- call a constructor of the superclass
- call the superclass' implementation of a method

Example:

```
public abstract class Animal
{
   // ...

   public Animal(String name)
   {
      // ...
   }

   public void sleep()
   {
      // ...
   }
}
```

```
public class NinjaCat extends Animal
{
   // ...

   public NinjaCat(String name)
   {
      super(name); // Calls Animal's one argument constructor
                             // The call to super must be the first statement in a constructor if it is supplied
   }

   @Override
   public void sleep()
   {
       // ...
       super.sleep(); // Call's Animal's implementation of sleep(). This does not need to be the 1st statement
                              // in this method
      // ...
   }
}
```

In a subclass' constructor, a call to super must be the first statement (if supplied). If a call to super is not supplied by us, then Java will add an implicit call to super() (calling the superclass' no-argument constructor) as the first line in the subclass' constructor. In this case, if the superclasss does not have a no-argument constructor, then you will get a compile time error.