2023-03-01

Today

- **08:35-08:45** | **9.5: Inheritance Hierarchies**
- **08:45-09:45** | Replit: Inheritance Hierarchies
- **09:45-10:00** | **9.6: Polymorphism**

Friday

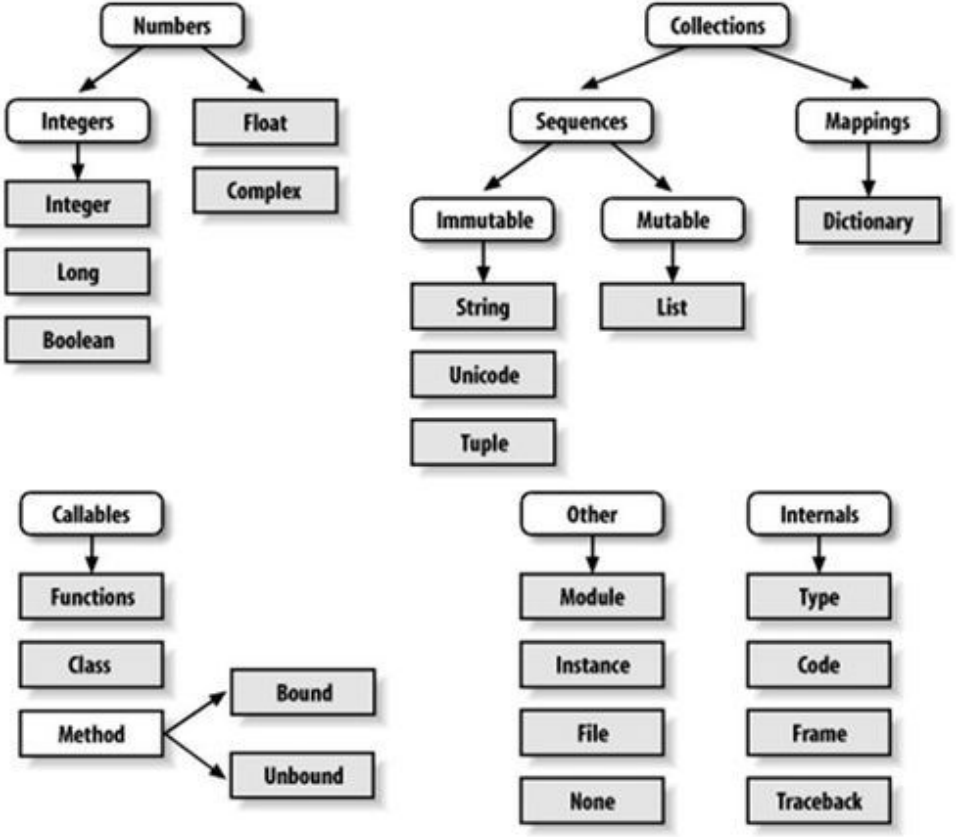
- **09:20-09:30** | **9.6: Polymorphism (if needed)**
- **09:30-10:15** | Replit TBD

9.5: Inheritance Hierarchies

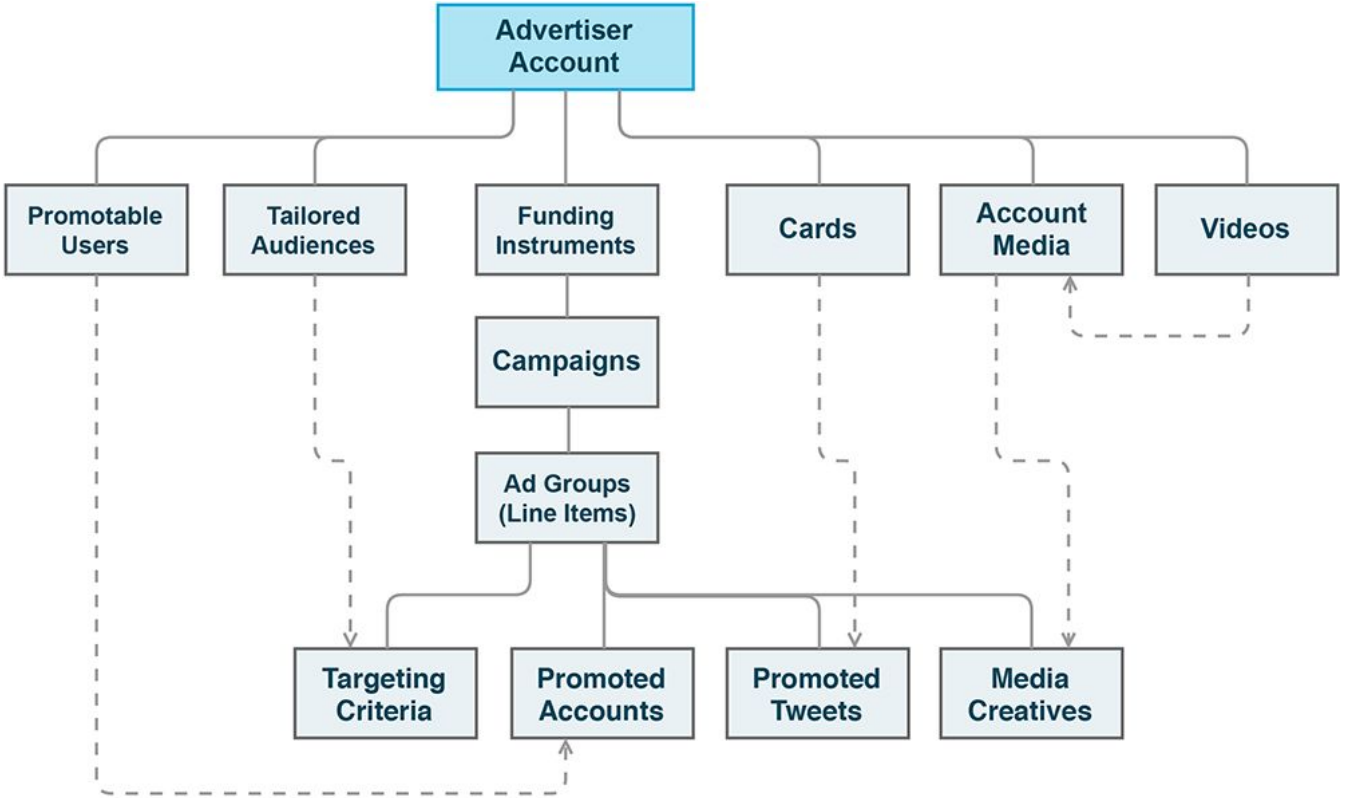
Inheritance Hierarchy

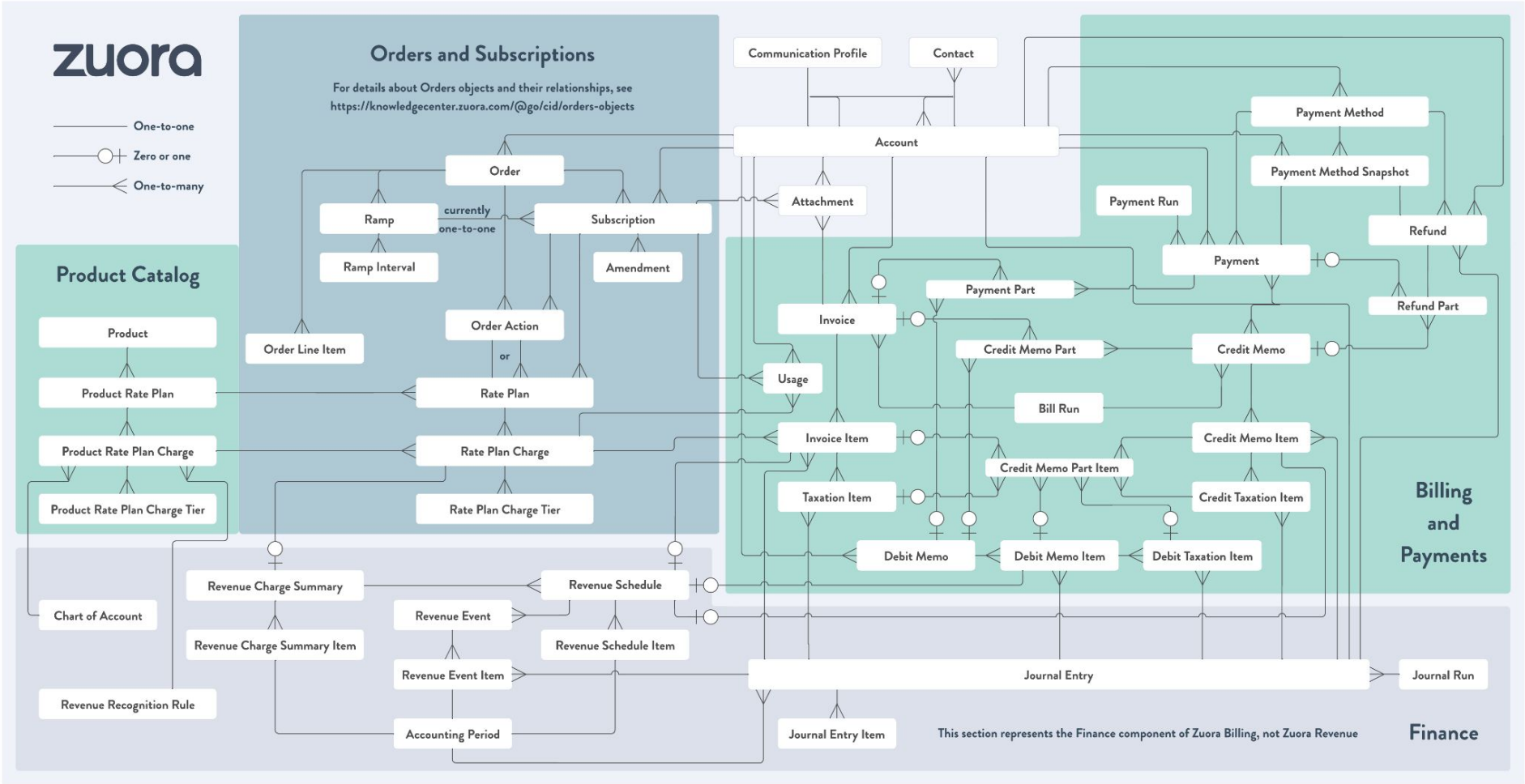
- **Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.
- When you use multiple layers of **Inheritance** in your program you end up with a set of relationships called an Inheritance Hierarchy - most often illustrated as a tree

Python's built-in type hierarchy

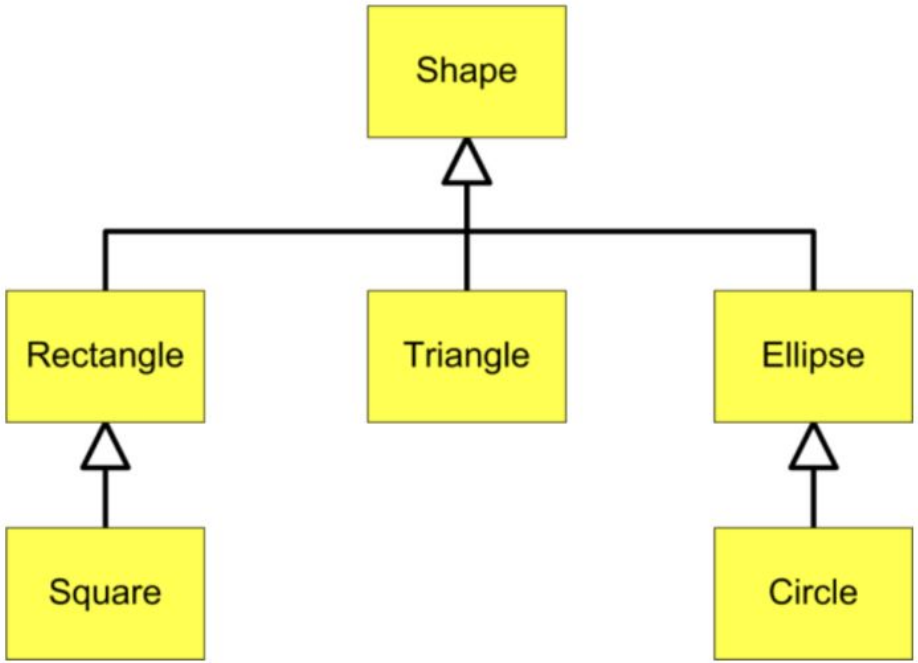


Twitter Ads API Hierarchy

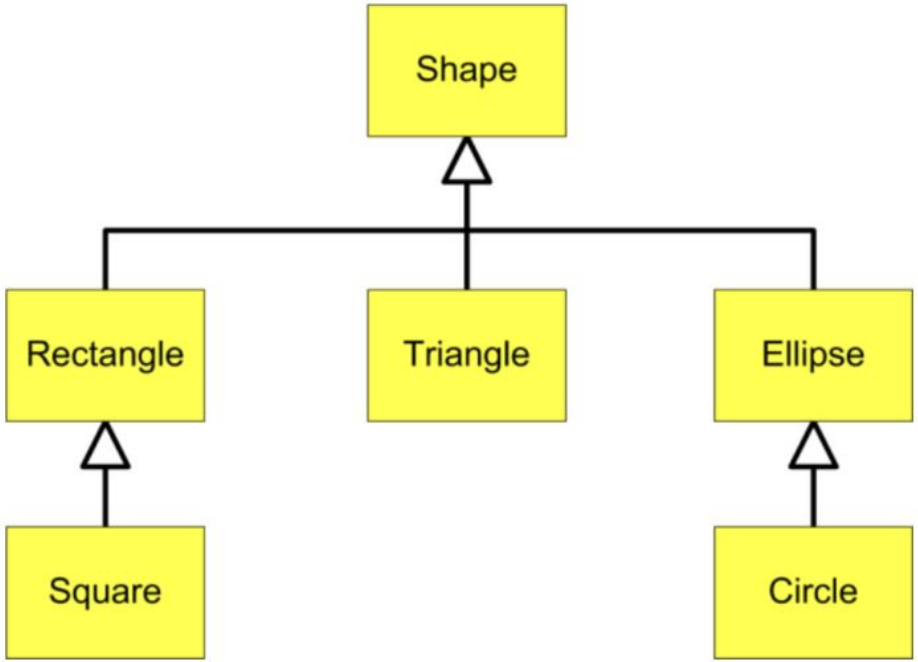



zuora
One-to-one
Zero or one
One-to-many
Orders and Subscriptions
For details about Orders objects and their relationships, see
https://knowledgecenter.zuora.com/@go/cid/orders-objects
Communication Profile
Contact
Account
Order
Ramp
currently
one-to-one
Subscription
Attachment
Ramp Interval
Amendment
Order Line Item
Order Action
or
Rate Plan
Rate Plan Charge
Rate Plan Charge Tier
Usage
Product Catalog
Product
Product Rate Plan
Product Rate Plan Charge
Product Rate Plan Charge Tier
Payment Method
Payment Method Snapshot
Payment Run
Refund
Payment
Payment Part
Refund Part
Invoice
Credit Memo Part
Credit Memo
Bill Run
Invoice Item
Credit Memo Item
Credit Memo Part Item
Taxation Item
Credit Taxation Item
Debit Memo
Debit Memo Item
Debit Taxation Item
Billing and Payments
Revenue Charge Summary
Revenue Schedule
Chart of Account
Revenue Event
Revenue Charge Summary Item
Revenue Schedule Item
Revenue Event Item
Journal Entry
Journal Run
Revenue Recognition Rule
Accounting Period
Journal Entry Item
This section represents the Finance component of Zuora Billing, not Zuora Revenue
Finance

Geometric Shapes

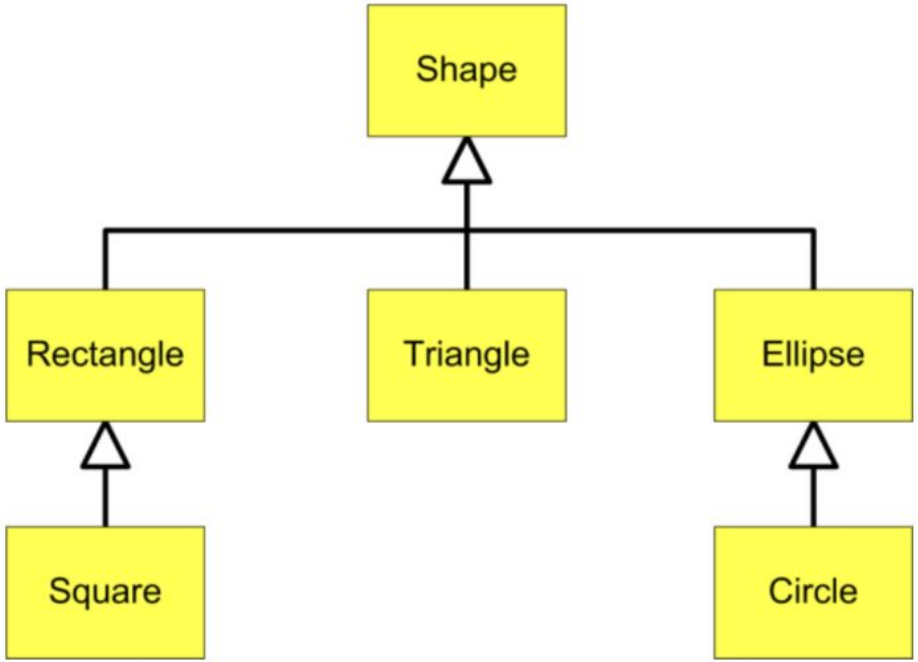


Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

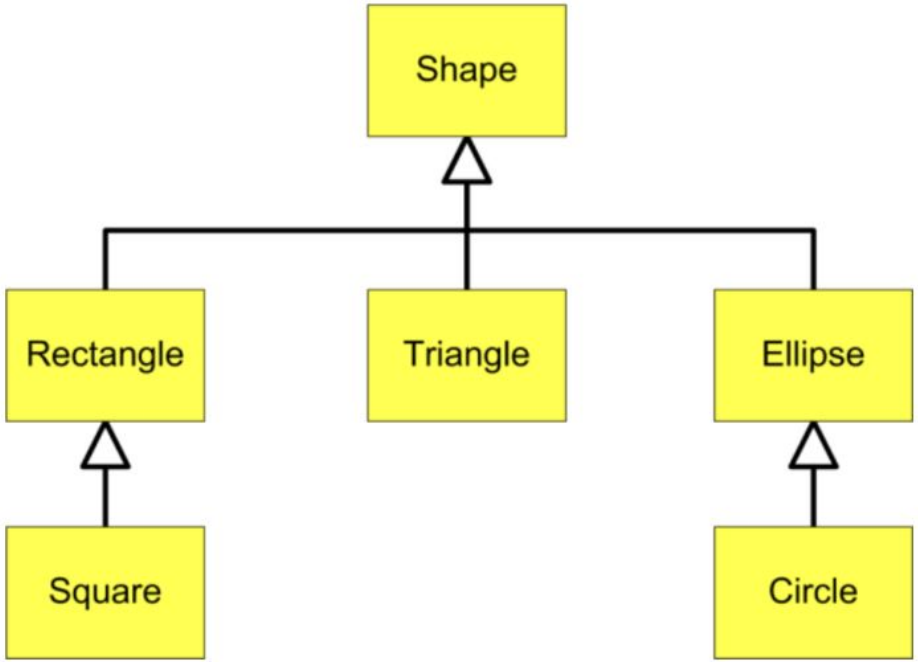
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse
```

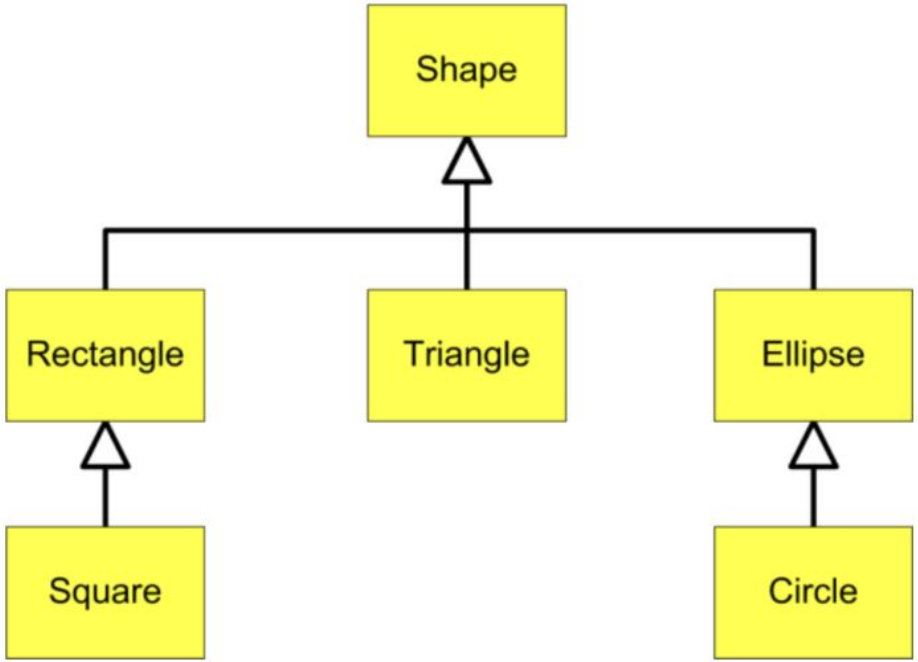
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape
```

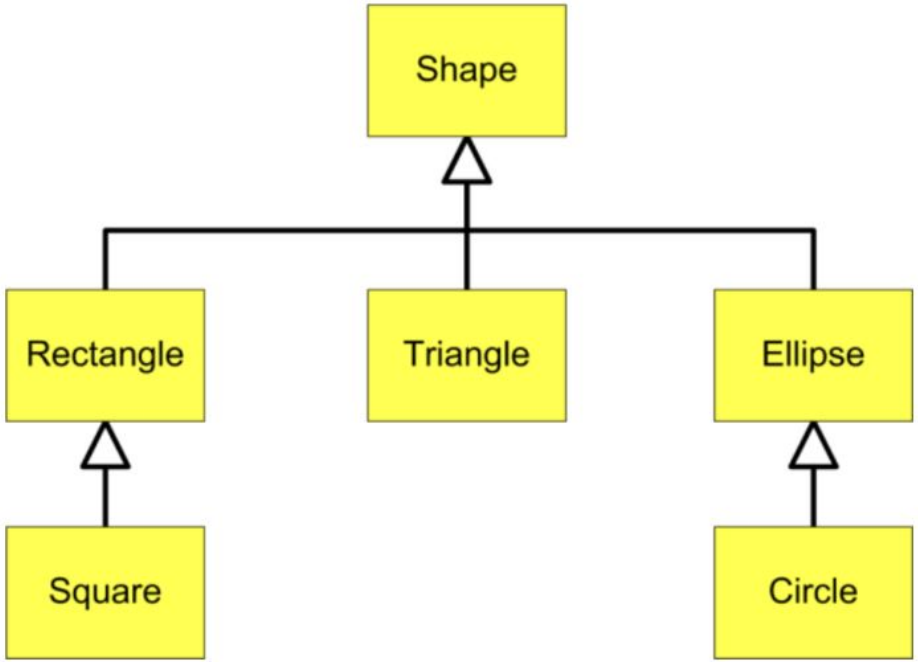
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape
```

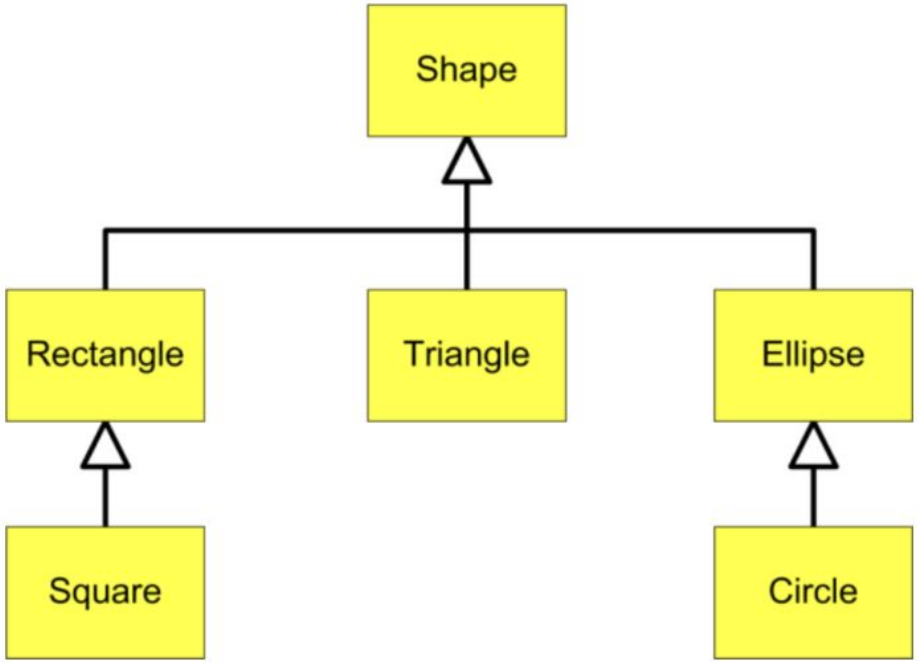
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle
```

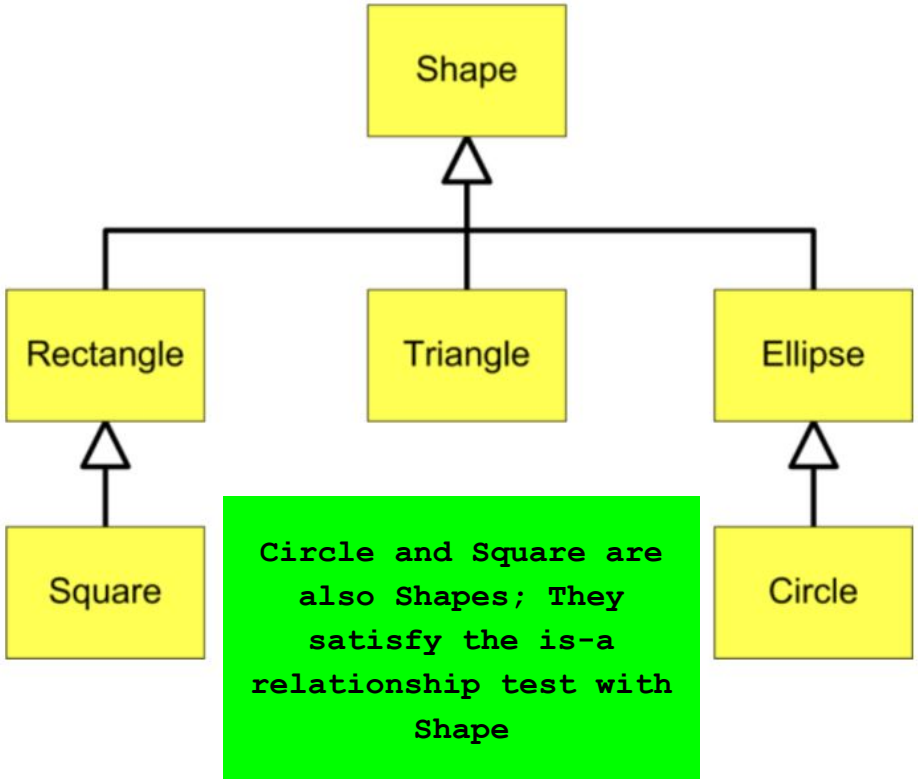
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

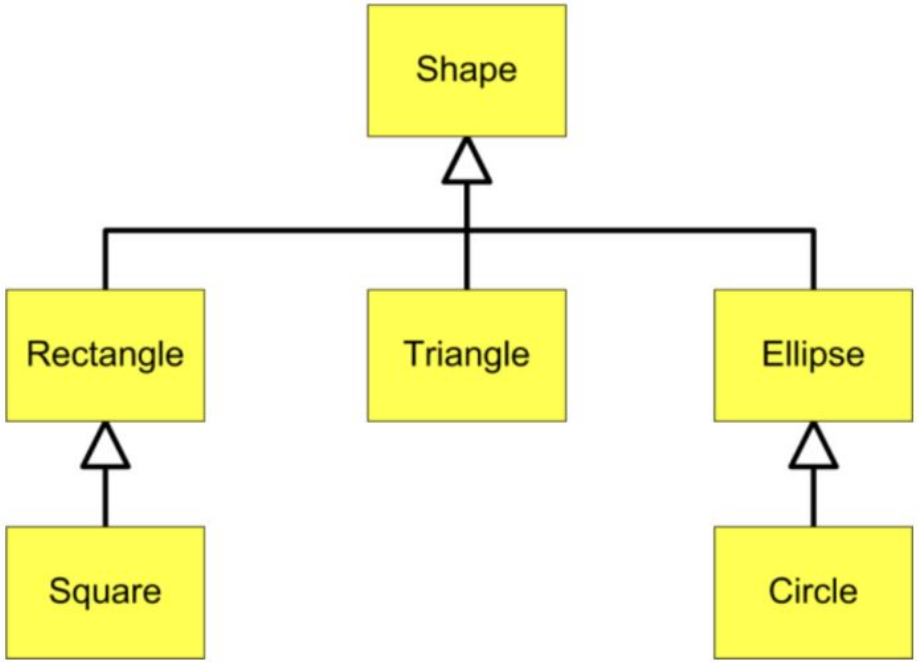
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

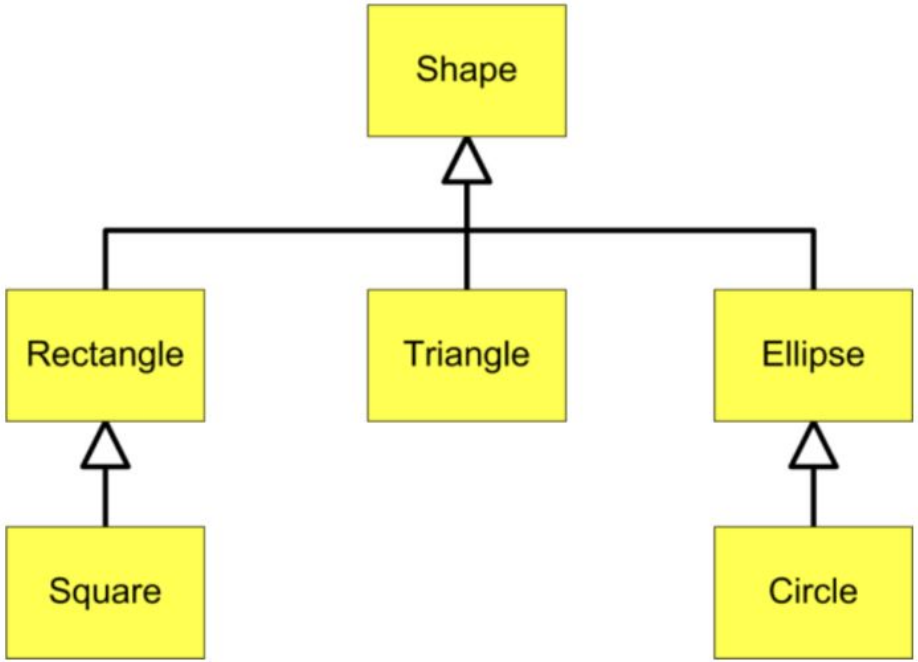
```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

Inheritance Hierarchy



- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

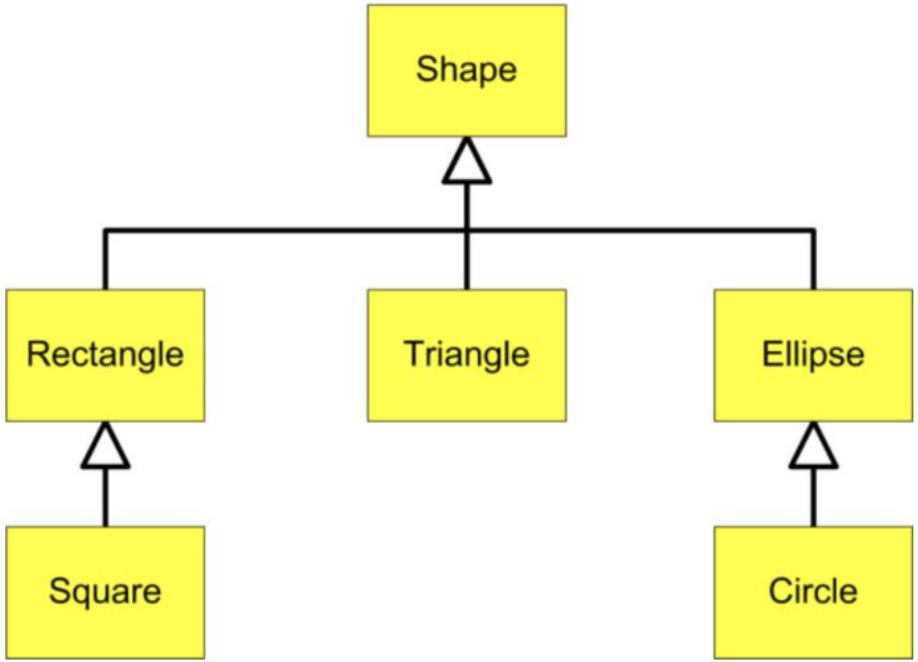
Inheritance Hierarchy



- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape
```

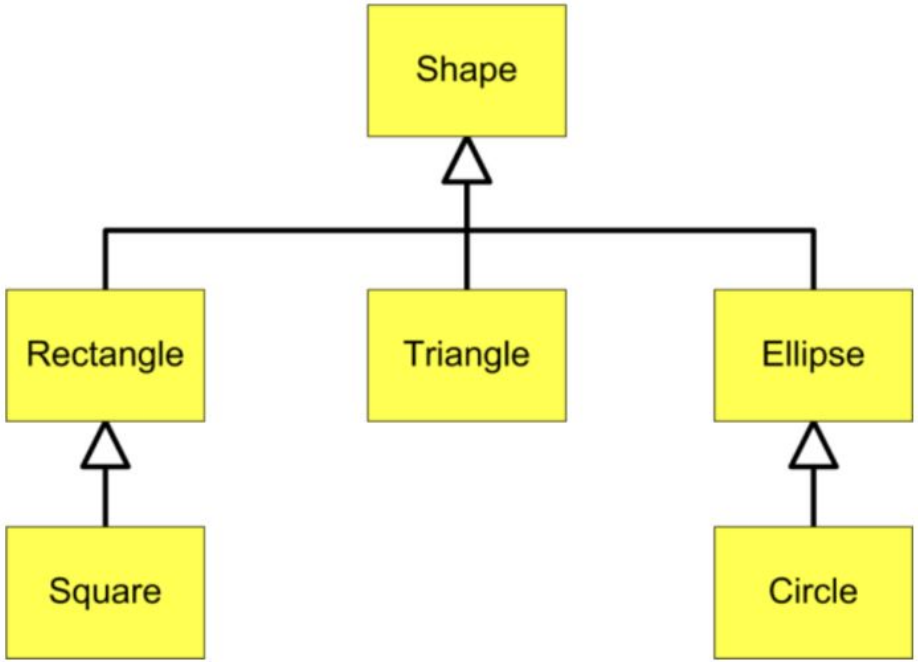
Inheritance Hierarchy



- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape

Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

Inheritance Hierarchy



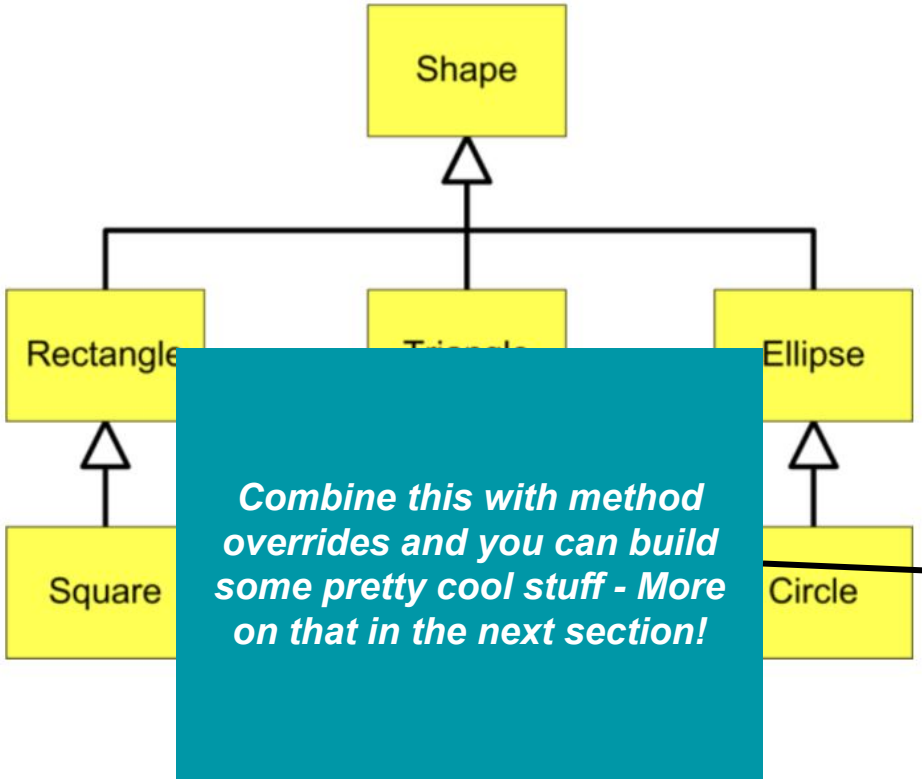
- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape
```

```
Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

This means you can create an `Array/ArrayList` of `Shapes` and add any one of these Objects to it!

Inheritance Hierarchy

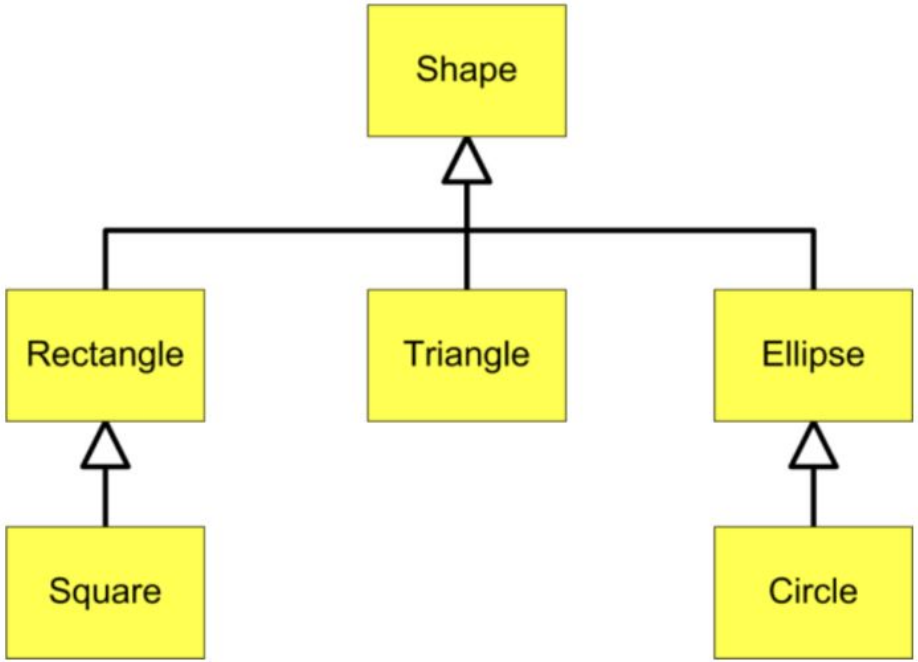


- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape

Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

This means you can create an `Array/ArrayList` of `Shapes` and add any one of these Objects to it!

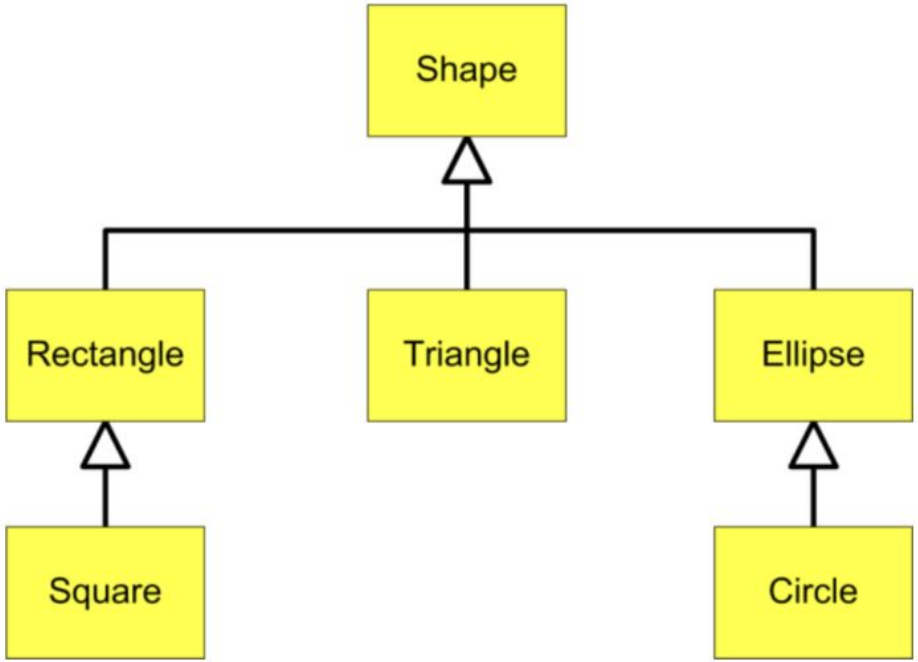
Inheritance Hierarchy



- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape
```

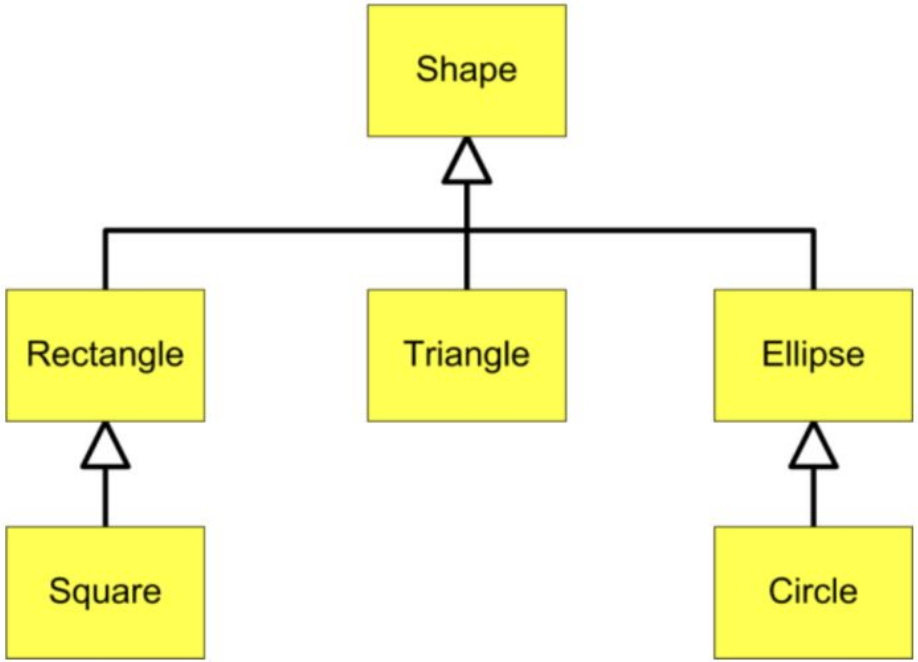
Inheritance Hierarchy



- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s;
Triangle t = s;
Ellipse e = e;
```

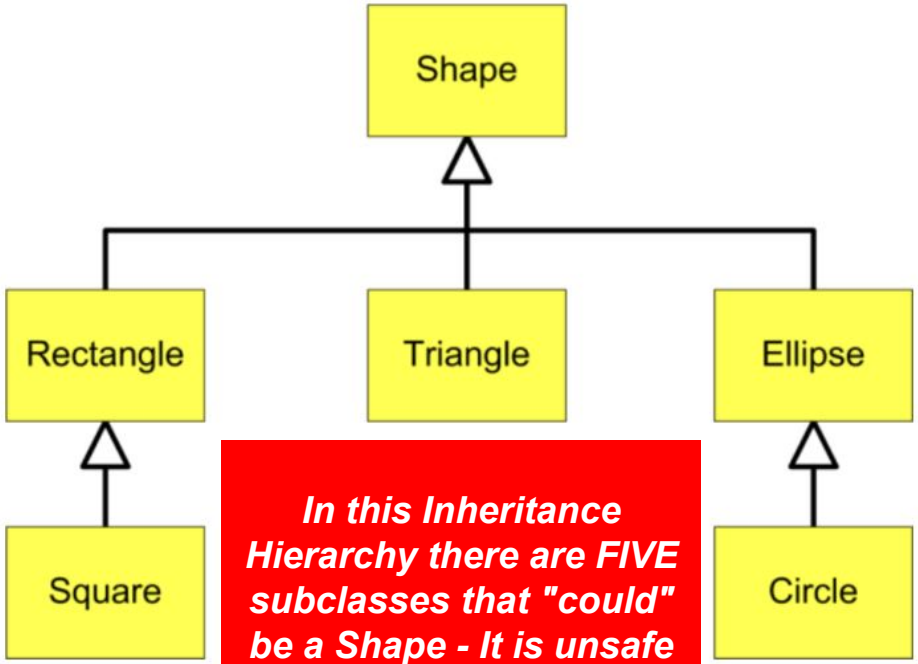
Inheritance Hierarchy



- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s; **ERROR**
Triangle t = s; **ERROR**
Ellipse e = e; **ERROR**
```

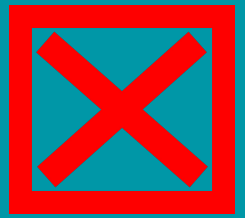
Inheritance Hierarchy



In this Inheritance Hierarchy there are FIVE subclasses that "could" be a Shape - It is unsafe to assume that a Shape is any one of them!

- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s; **ERROR**
Triangle t = s; **ERROR**
Ellipse e = e; **ERROR**
```



8:45-9:45

Replit: Inheritance Hierarchies

9.6: Polymorphism

Quick review: compile time vs. runtime

- Executing a java program has two steps. First, the program must be **compiled** (e.g. turned into 1s and 0s that your computer can understand) and then **run**
- The first step of this process is orchestrated by a program called a **compiler.**
  - This is the program that yells at you if you try to use a variable before you've initialized it
  - Compilation involves checking a bunch of syntactic "rules" to make sure that your program logic is well-defined
- But you can also encounter error messages generated at **runtime**
  - For example, an error message that says you tried to divide by zero
  - These kinds of error can't be identified a priori–your code needs to be run for these issues to be caught

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}

> ?
> ?
> ?
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}

> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

The `Object` instances living in the `shapes` array have compile-time types of `Rectangle`, `Triangle`, and `Ellipse` (because that is the type that was created with `new`)

```
class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}

> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

As we perform the `for-in` loop - each element of `shapes` is assigned to a `Shape` variable - this is the run-time type of each `Object` instance inside the loop (the compile-time class never changes)

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

```
> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- The **compiler**
  - Uses the **compile-time type** to verify that the methods you are trying to use are available to an object of that type.
  - The code won't compile if the methods don't exist in that class or some parent class of that class.
- During **runtime**
  - Uses the **run-time type** to determine which methods are used
  - When a method is called the first place that is checked for that method is the class that created the object. If the method is found there it will be executed. If not, the parent of that class will be checked and so on until the method is found.

Polymorphism

- Polymorphic Assignment
  - `Shape s = new Rectangle();`
- Polymorphic Parameters
  - `public void print(Shape s){}`
- Polymorphic Collections
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

Polymorphism

- Polymorphic Assignment
  - `Shape s = new Rectangle();`
- Polymorphic Parameters
  - `public void print(Shape s){}`
- Polymorphic Collections
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

There are no errors at compile-time because the compiler checks that the "subclass is-a superclass" relationship is true.

Polymorphism

- Polymorphic Assignment
  - `Shape s = new Rectangle();`
- Polymorphic Parameters
  - `public void print(Shape s){}`
- Polymorphic Collections
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

There are no errors at compile-time because the compiler checks that the "subclass is-a superclass" relationship is true.

At run-time, the Java runtime will use the object's actual subclass type and call the subclass methods for any overridden methods.

Polymorphism

- Polymorphic Assignment
  - `Shape s = new Rectangle();`
- Polymorphic Parameters
  - `public void print(Shape s){}`
- Polymorphic Collections
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

There are no errors at compile-time because the compiler checks that the "subclass is-a superclass" relationship is true.

At run-time, the Java runtime will use the object's actual subclass type and call the subclass methods for any overridden methods.

This is why they are polymorphic – the same code can have different results depending on the object's actual type at run-time.

Replit: Inheritance Hierarchies (Part 2)

Replit: Inheritance Hierarchies (Part 2)

- Copy the **VerifyShoppingCart.java** file (found in Inheritance Hierarchies Part 2) into your Inheritance Hierarchies project
- Add the following line to your Main.run function

```
public void run() {
  Verify.validateClassesAreDefined();
  Verify.validateInheritanceHierarchy();
  Verify.validateConstructors();
  Verify.validateEggs();
  Verify.validateMilk();
  VerifyShoppingCart.validateShoppingCart();
  System.out.println("");
}
```

- Follow the instructions in **VerifyShoppingCart.java**