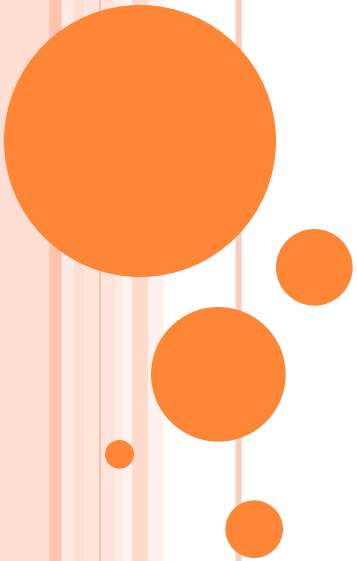


OOP Principles

CSI 203: Object Oriented Programming

Tanjina Helaly

3 Principles

- All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.
  - Inheritance
  - Encapsulation
  - Polymorphism

Inheritance

Inheritance

- Inheritance is the process by which one object acquires the properties of another object.
- It is a way to form new classes using classes that have already been defined.
- The new classes (child classes), take over (or inherit) **attributes** and **behavior** of the pre-existing classes (parent classes).

Inheritance

- Java uses “extends” keyword to show inheritance relationship. Example

  class Child extends Parent{}
- The class that is extended is a **superclass**
  - Other terms: parent class, base class, ancestor class
- The extended class is a **subclass** of its superclass
  - Other terms: child class, extended class, derived class
- An object created from the subclass has its **own copy** of all the **nonstatic fields** defined in its superclass

Inheritance

- This is important because it supports the concept of hierarchical classification.
- Inheritance provides a powerful and natural mechanism for organizing and structuring your software
- It is intended to help reuse existing code with little or no modification.

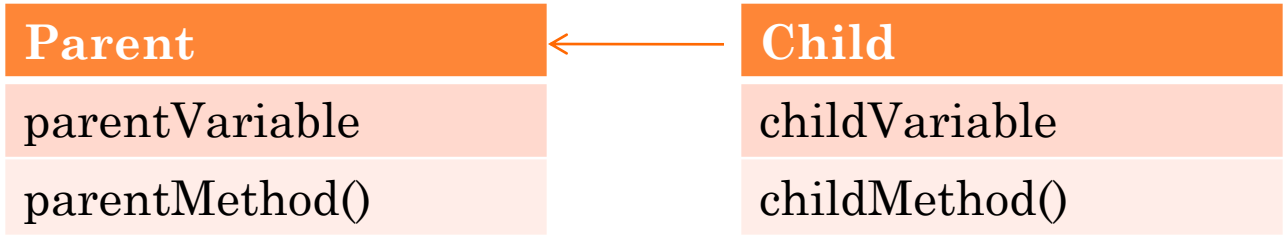
Inheritance - Example

```
class Parent {
    public int parentVariable = 10;
    public void parentMethod() {
        System.out.println( "Parent Method" );
    }
}
class Child extends Parent {
    public int childVariable = 5;
    public void childMethod() {
        parentMethod();
        System.out.printf( "In Child ParentVariable=%d, ChildVariable=%d“,
        parentVariable, childVariable );
    }
}
class Inheritance {
    public static void main( String args[] ) {
        Child example = new Child();
        example.childMethod();
        example.parentMethod();
        System.out.println( example.parentVariable );
    }
}
```

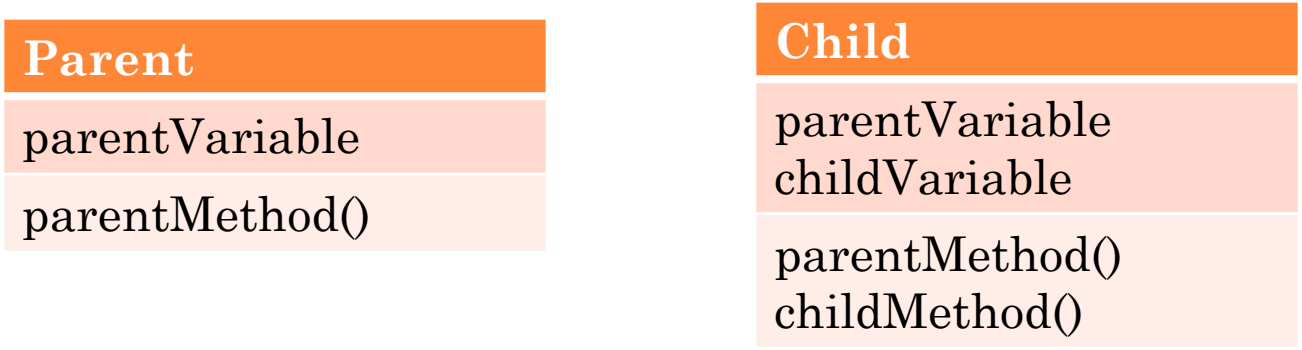
Output

```
Parent Method
In Child ParentVariable=10, ChildVariable=5
Parent Method
10
```

Inheritance - Example

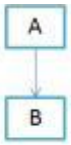


Act like

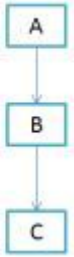


Types of Inheritance

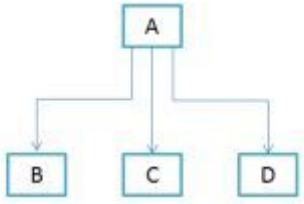
- Single inheritance
  - One parent-> One child



- Multi level inheritance
  - one can inherit from a derived class, thereby making this derived class the base class for the new class.

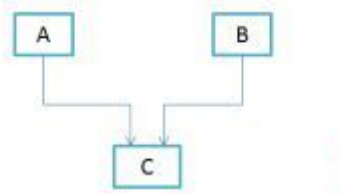


- Hierarchical inheritance
  - One parent – multiple children

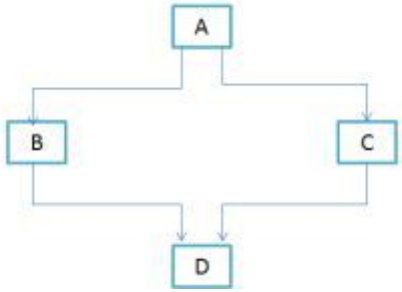


Types of Inheritance

- Multiple inheritance
  - Multiple parents -> one child



- Hybrid inheritance
  - Is a combination of Single and Multiple inheritance



- Java does not support multiple inheritance

Inheritance – super keyword

- The **super** keyword in java is a reference variable that is used to refer immediate parent class object.
- Whenever you **create the instance of subclass, an instance of parent class is created implicitly** i.e. referred by super reference variable.
- Usage of java super Keyword
  - super is used to refer immediate parent class member(instance variable & method).
  - super() is used to invoke immediate parent class constructor.

SUPER – PARENT'S INSTANCE VARIABLE/METHOD

- The super keyword can also be used to
  - Access parent class's instance variable if child has an instance variable with same name.
  - invoke parent class's method. It should be used in case subclass contains the same method as parent class.

SUPER – PARENT'S INSTANCE VARIABLE/METHOD

```
class Parent {
   String name;
   void message(){
       System.out.println("Welcome to Parent class."); }
}
// Create a subclass by extending class parent.
class Child extends Parent {
   String name; // this name hides the name in Parent

   public Child(String a, String b) {
         super.name = a; // name in A
         name = b; // name in B
   }
   void show() {
   System.out.println("superclass's name: " + super.name + "; subclass's name: " + name); }

   //method overriding
   void message(){
       System.out.println("Welcome to Child class."); }

   void display(){
         message();//will invoke current class message() method
         super.message();//will invoke parent class message() method
   }
}
```

SUPER – PARENT'S INSTANCE VARIABLE/METHOD

```
public class TestSuper {
    public static void main(String[] args) {
        Child s=new Child("Parent", "Child");
        s.show();
        s.display();
    }
}
```

- Output:

superclass's name: Parent; subclass's name: Child

Welcome to Child class.

Welcome to Parent class.

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- The super keyword can also be used to invoke the parent class constructor as given below:

```
class Vehicle{
    Vehicle(){
        System.out.println("Vehicle is created.");
    }
}

class Bike extends Vehicle{
    Bike(){
        super(); //will invoke Parent class's constructor
        System.out.println("Bike is created.");
    }
    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

Output:
Vehicle is created
Bike is created

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- What would be the output of the program below:

```
class Vehicle{
    Vehicle(){
        System.out.println("Vehicle is created.");
    }
}

class Bike extends Vehicle{
    Bike(){
        System.out.println("Bike is created.");
    }
    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- What would be the output of the program below:

```
class Vehicle{
    Vehicle(){
        System.out.println("Vehicle is created.");
    }
}

class Bike extends Vehicle{
    Bike(){
        System.out.println("Bike is created.");
    }
    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

Output:
Vehicle is created
Bike is created

If the constructor doesn't have super(), compiler will provide super() as the first statement of the constructor.

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- **Note:**
- super() should be the first statement of your constructor
- If the constructor doesn't have super(), compiler will provide super() as the first statement of the constructor.
  - **It will only work when Parent has no constructor or a parameter less constructor. See the next example.**

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- What would be the output of the program below:

```
class Vehicle{
    Vehicle(String name){
        System.out.printf("Vehicle %s is created.\n", name);
    }
}

class Bike extends Vehicle{
    Bike(){
        System.out.println("Bike is created.");
    }

    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- What would be the output of the program below:

```
class Vehicle{
    Vehicle(String name){
        System.out.printf("Vehicle %s is created.\n", name);
    }
}

class Bike extends Vehicle{
    Bike(){ // Implicit super constructor Vehicle() is undefined. Must explicitly
    invoke another constructor
        System.out.println("Bike is created.");
    }

    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

SUPER-TO INVOKE PARENT CONSTRUCTOR.

- So the fix is:

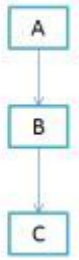
```
class Vehicle{
    Vehicle(String name){
        System.out.printf("Vehicle %s is created.\n", name);
    }
}

class Bike extends Vehicle{
    Bike(){
        super("Bike");
        System.out.println("Bike is created.");
    }

    public static void main(String[] args){
        Bike b = new Bike();
    }
}
```

When Constructors Are Executed

- When a class hierarchy is created, the constructors are executed in the order of the hierarchy.



- For example: for the hierarchy above, constructor of A will be executed first, then B and then C.

When Constructors Are Executed

```
// Create a super class.
class A {
  A() {
      System.out.println("Inside A's constructor."); }
}
// Create a subclass by extending class A.
class B extends A {
  B() {
      System.out.println("Inside B's constructor."); }
}
// Create another subclass by extending B.
class C extends B {
  C() {
      System.out.println("Inside C's constructor."); }
}
class CallingCons {
  public static void main(String args[]) {
      C c = new C();
  }
}
```

Output:
Inside A's constructor
Inside B's constructor
Inside C's constructor

Object Class

- All classes inherit directly or indirectly from java.lang.Object
  - class Parent { int size; }
  - class Parent extends Object { int size; }
- The child class below is a grandchild of Object
  - class Child extends Parent{ }
- Having a common ancestor class allows java to provide standard members on all objects, like toString()

```
class TestObject {
    public static void main( String args[] ) {
        Parent watchThis = new Parent();
        int myHash = watchThis.hashCode();
        System.out.println( myHash );
        // Where does hashCode come from?
    }
}
```

Object Class - methods

- clone() - Creates a clone of the object.
- equals(Object) - Compares two Objects for equality.
  - Uses "==" to test for equality
  - Note: If a class needs an implementation of equals, which differs from the default "equals", the class should override the method.
- toString() - Returns a String that represents the value of this Object.

Encapsulation

Encapsulation

- *Encapsulation* is the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse.
- One way to think about encapsulation is as a protective wrapper
  - that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
- the basis of encapsulation is the class.
  - Use access modifier to provide encapsulation

Encapsulation

- Since the **purpose** of a class/encapsulation is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class.
  - Use **access modifier** to specify which members should/ shouldn't be accessed by outside world.
  - the **public interface** should be carefully designed not to expose too much of the inner workings of a class

Getter/Setter Method

- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.
- To achieve that in Java –
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.
  - Example:

```
private double balance;
public void setBalance(double b) {
        balance = b;
}
public double getBalance() {
        return balance;
}
```

Benefit of Encapsulation

- Benefit
  - Hide complexity from user.
  - Implement logic while updating field
  - Easy to maintain
  - Make variable read-only or write only

Polymorphism

Polymorphism

- **Poly** means **many** and **morph** means **form**. Thus, polymorphism refers to being able to use many forms of a type without regard to the details.

- More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”

- *polymorphism* refers to a programming language's ability to process objects differently depending on their data type or class.

Polymorphism

- 3 types
  - Method overriding
  - Method overloading
  - Subclass polymorphism

Method overriding

- Means a **child class** is **re-implementing** a **method of its super class**. Or
- A class replacing an ancestor's implementation of a method with an implementation of it own.
- When overriding a method in child class
  - Method Signature(name and argument list) and return type must be the same as the parent method.
  - Child method could be equal or more accessible.

Method overriding

- When an **overridden method is called** from within its **subclass**, it will always refer to the version of **that method defined by the subclass**.
  - The version of the method defined by the superclass will be hidden.
- To invoke parent method in child class, need to use "**super**" keyword.
- Static methods can not be overridden
- Dynamic binding

METHOD OVERRIDING - EXAMPLE

```
public class TestOverridding {
    public static void main(String[] args) {
        System.out.println("------Parent-----");
        Parent p = new Parent();
        p.display();

        System.out.println("\n------Child-----");
        Child c = new Child();
        c.display();
    }
}

class Parent {
    public void display() {
        System.out.println( "Display in Parent" );
    }
}
class Child extends Parent {
    public void display() {
        System.out.println( "Display in Child" );
    }
}
```

Output:

------Parent-----
Display in Parent

------Child-----
Display in Child

Method overriding - Example(super keyword)

```
public class TestOverridding {
    public static void main(String[] args) {
        System.out.println("------Parent-----");
        Parent p = new Parent();
        p.display();

        System.out.println("\n------Child-----");
        Child c = new Child();
        c.display();
    }
}

class Parent {
    public void display() {
        System.out.println( "Display in Parent" );
    }
}
class Child extends Parent {
    public void display() {
        super.display(); // invoke the parent class's method
        System.out.println( "Display in Child" );
    }
}
```

Output:

------Parent-----
Display in Parent

------Child-----
Display in Parent
Display in Child

Method overloading

- Two methods in the **same class** can have the **same name** with different **signature**. This is called method overloading and the methods are called overloaded method.
  - The signature of a method is its **name** with **number**, **type** and **order** of its parameters.
  - The return type is not part of the signature of a method.
- Condition:
  - Methods should be in the same class
  - Same name
  - Must have different argument.
  - Return type could be same or different.

Method overloading

- This called static binding because, which method to be invoked will be decided at the time of compilation

Method overloading - Example

```
class OverloadDemo {
      void test() {
          System.out.println("No parameters"); }

      // Overload test for one integer parameter.
      void test(int a) {
          System.out.println("a: " + a); }

      // Overload test for a double parameter
      double test(double a) {
          System.out.println("double a: " + a);
          return a*a; }
}
class Overload {
      public static void main(String args[]) {
            OverloadDemo ob = new OverloadDemo();
            // call all versions of test()
            ob.test();
            ob.test(10);
            double result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
      }
}
```

Output:

```
No parameters
a: 10
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Constructor Overloading

- In addition to overloading normal methods, you can also overload constructor methods.
- In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

Constructor Overloading - Example

```
public class Box {
    double width, height, depth;
    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
    public static void main(String[] args){
        Box b = new Box(10, 8, 5);
        System.out.println("Volume of Box: " + b.volume());
        Box b1 = new Box(5);
        System.out.println("Volume of Cube: " + b1.volume());
    }
}
```

Output:
Volume of Box: 400.0
Volume of Cube: 125.0

Invoking Overloaded Constructors -this( )

- Sometimes it is useful for one constructor to invoke another.
  - this is accomplished by using the **this keyword.**
  - The general form is
    - this(*arg-list)*
- When **this( ) is executed, the overloaded constructor that matches the parameter list** specified by *arg-list is executed.*
- The call to **this( ) must be the first statement within** the constructor.
  - General rule: Constructor call must be the first statement in a constructor.

Constructor Overloading - Example

```
public class Box {
    double width, height, depth;
    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
    public static void main(String[] args){
        Box b = new Box(10, 8, 5);
        System.out.println("Volume of Box: " + b.volume());
        Box b1 = new Box(5);
        System.out.println("Volume of Cube: " + b1.volume());
    }
}
```

Output:
Volume of Box: 400.0
Volume of Cube: 125.0

Constructor Overloading – Example with this()

```
public class Box {
    double width, height, depth;
    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // constructor used when cube is created
    Box(double len) {
        this(len, len, len);
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
    public static void main(String[] args){
        Box b = new Box(10, 8, 5);
        System.out.println("Volume of Box: " + b.volume());
        Box b1 = new Box(5);
        System.out.println("Volume of Cube: " + b1.volume());
    }
}
```

Output:
Volume of Box: 400.0
Volume of Cube: 125.0

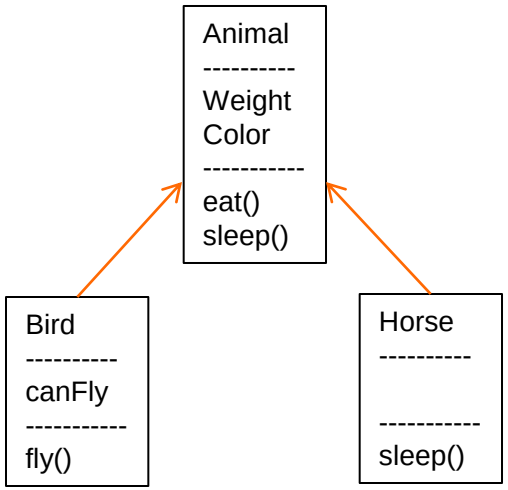
Subclass Polymorphism

- A **parent class reference** is used to **refer** to a **child class object**.
- Couple things to remember:
  - The only possible way to access an object is through a reference variable.
  - The type of the reference variable would determine the methods that it can invoke on the object.
- Using subclass polymorphism we can call or execute the child-class overriding method by the parent-class object.

Subclass Polymorphism - Example

```
public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a = new Bird();
        a.sleep(); // sleep() method of animal class will be executed

        Animal h = new Horse();
        h.sleep(); // sleep() method of Horse class will be executed
    }
}
```



Subclass Polymorphism –Compile/run time

- 2 types of check
  - reference variable would determine the methods that it can invoke on the object. **Compile time check.**
  - object type (NOT reference variable type) determines which overridden method will be used at **runtime.**
  - Consider the code below

```
Animal a = new Bird();
a.sleep();
```

    - Can't call a.sleep() if the sleep() method is not available in Animal class. Will produce compile error.
    - During runtime the Horse's sleep() method will be executed not Animal's.

Subclass Polymorphism – access method not available in parent

```
public class TestPolymorphism {
	public static void main(String[] args) {
		Animal a = new Bird();
		a.fly(); // Compile error: The  method fly() is undefined for the type Animal
	}
}
```

- So what to do if need to call subclass method that is not available in Parent class?
  - Casting

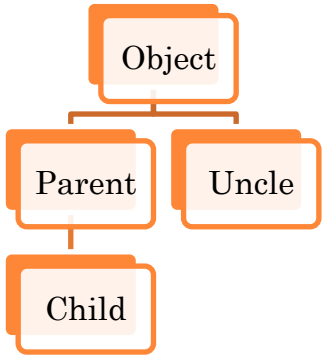
```
public class TestPolymorphism {
	public static void main(String[] args) {
		Animal a = new Bird();
		((Bird)a).fly(); // OK
	}
}
```

Casting and Classes

- An instance of a child class can be assigned to a variable (field) of the parent class.
- variable references an subtype object can be cast down to its subclass type with an **explicit** cast.
- A runtime error occurs when explicitly casting an object to a type that it is not.
  - An object of type Parent cannot be cast to type Child.
- In the code shown in next page, the cast "(Uncle) object" is a runtime error
  - because at that time object holds an instance of the Child class, which is not of type (or subclass) of Uncle.

Casting and Classes

```
class Casting {
      public static void main( String args[] ) {
            Object object;
            Parent parent;
            Child child = new Child();
            Uncle uncle;
            parent = child;
            object = child;
            parent = (Parent) object; // explicit cast down
            child = (Child) object; // explicit cast down
            uncle = (Uncle) object; //Runtime exception
      }
}
```



- **Output**

  java.lang.ClassCastException: Child: cannot cast to Uncle

- the cast "(Uncle) object" is a runtime error - because at that time object holds an instance of the Child class, which is not of type (or subclass) of Uncle.

Precaution while down casting

- We need to first check if the object is of that type.
- How?
  - The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
  - getClass() method return the “class [ClassName]”
  - getClass().getName() return the class name.

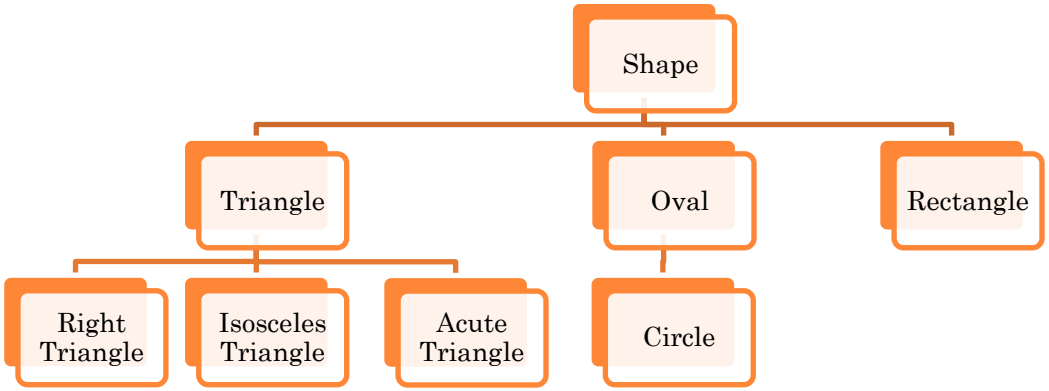
- Let’s revisit example of polymorphism.

```
public class TestPolymorphism {
      public static void main(String[] args) {
            Animal a = new Bird();
            if (a instanceof Bird) // can also use if (a.getClass().getName().equals("Bird"))
               ((Bird)a).fly(); // OK
      }
}
```

Subclass Polymorphism – Benefits – Need to Update

- Process objects differently based on their data type.
  - The same invocation can produce "many forms" of results
  - We can override method in subclasses and which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the real object)
- Can pass parent ref as method argument and handle all subclasses
- Can return parent type and handle all subclasses.

Polymorphism – Benefits



- Assume the above hierarchy.
- Consider each class has a draw() method to draw the specific shape.
- Also consider a method that will take a Shape reference variable (which can hold a shape object or any of its subclass's object) as argument and call the draw method. E.g.

```
static void drawShape(Shape shape){
    shape.draw();
}
```

Polymorphism – Benefits

- What will happen if you call the method using a Triangle object or a Circle object.

  drawShape(new Triangle()); // will call the draw method of Triangle class
  drawShape(new Circle()); // will call the draw method of Circle class

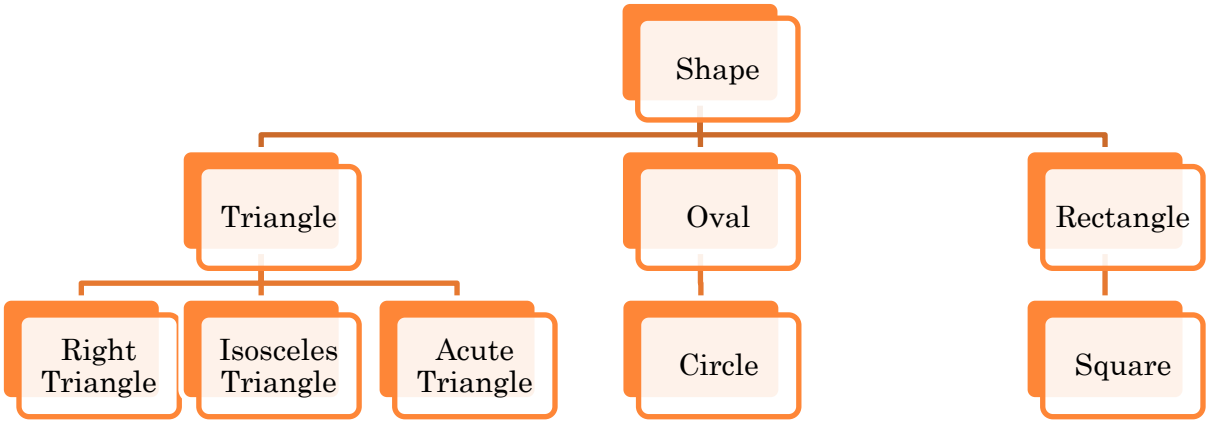
  - The same invocation can produce "many forms" of results
  - We can override method in subclasses and which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the real object)

Polymorphism - Benefits

- Polymorphism enables programmers to deal with generalities and
  - let the execution-time environment handle the specifics.
- Programmers can command objects to behave in manners appropriate to those objects,
  - without knowing the types of the objects
  - (as long as the objects belong to the same inheritance hierarchy).

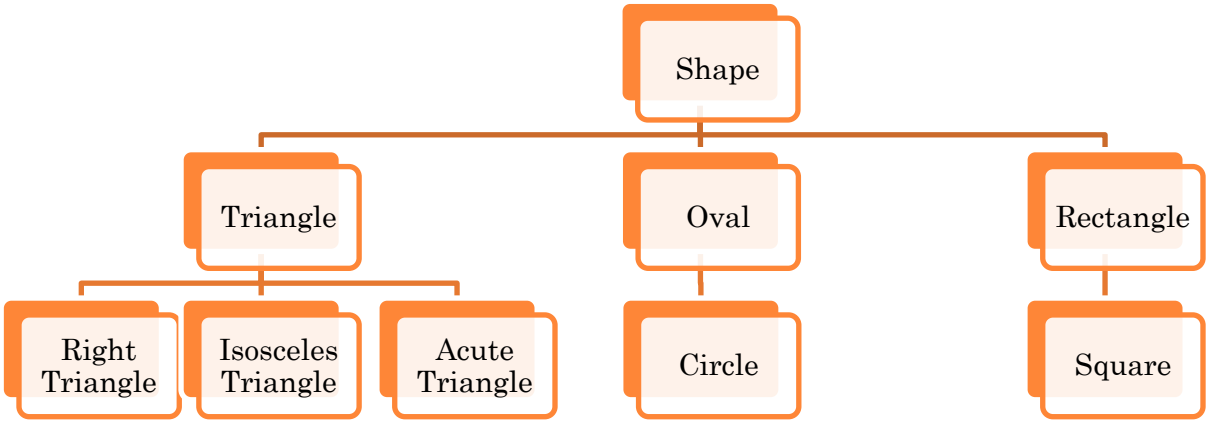
Polymorphism - Benefits

- Now think : - What need to change in the old hierarchy or ***drawShap(Shape shape)*** method, if you add a new subtype "Square" as shown below.



Polymorphism - Benefits

- Now think : - What need to change in the old hierarchy or ***drawShap(Shape shape)*** method, if you add a new subtype "Square" as shown below.



- Polymorphism Promotes Extensibility
  - New object types that can respond to existing method calls can be
    - incorporated into a system **without requiring** modification of the base system.
    - Only client code **that instantiates new objects must be modified** to accommodate new types.

Static & Dynamic Binding

- Association of method definition to the method call is known as binding.
- 2 types:
  - static binding (also known as early binding).
  - dynamic binding (also known as late binding).

Static Binding

- The binding which can be resolved at compile time by compiler is known as static. Or
- When type of the object is determined at compiled time(by the compiler), it is known as static binding.
- All the static, private and final methods have always been bonded at **compile-time** .
  - Why?
    - Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class.
    - Hence compiler doesn't have any difficulty to determine object of class (local class for sure).
    - That's the reason binding for such methods is static.

Dynamic Binding

- When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding.
- Example: Overriding
  - in overriding both parent and child classes have same method.
  - Thus while calling the overridden method, the compiler gets confused between parent and child class method.
  - The method is decided during runtime.

Reference

- Java:Complete Reference Chapter 7,8
- Online Reference:
  - http://www.javatpoint.com/super-keyword
  - https://www.tutorialspoint.com/java/
  - https://docs.oracle.com/javase/tutorial/