Inheritance

AP Computer Science

Can you see any similarities in the following examples?

Circle Class

```
public class Circle {
   private int x, y;
   private double radius;
   // constructors not shown
   // modifiers and accessors for y not shown
   public void setX(int xPos){
      x = xPos;
   }
   public int getX(){
      return x;
   }
   public double getArea(){
      return Math.PI * radius * radius;
   }
}
```

Rectangle Class

```
public class Rectangle {
   private int x, y;
   private double width, height;
   // constructors not shown
   // modifiers and accessors for y not shown
   public void setX(int xPos){
      x = xPos;
   }
   public int getX(){
      return x;
   }
   public double getArea(){
      return width * height;
   }
}
```

Triangle Class

```
public class Triangle {
   private int x, y;
   private double base, height;

   // constructors not shown

   // modifiers and accessors for y not shown

   public void setX(int xPos){
      x = xPos;
   }

   public int getX(){
      return x;
   }

   public double getArea(){
      return 0.5 * base * height;
   }
}
```

Code Duplication

Circle Class	Rectangle Class	Triangle Class
Methods Inside	**Methods Inside**	**Methods Inside**

getX()
setX()
getY()
setY()
getArea()
getRadius()
setRadius()
	getX()
	setX()
	getY()
	setY()
	getArea()
	getWidth()
	setWidth()
	getHeight()
	setHeight()
		getX()
		setX()
		getY()
		setY()
		getArea()
		getBase()
		setBase()
		setHeight()
		getHeight()

Instance Variables Inside	**Instance Variables Inside**	**Instance Variables Inside**
x **y** radius	**x** **y** width height	**x** **y** base height

Code Duplication

- There are duplicates of the instances variables for the x and y positions
- There are duplicates of the modifier and accessor methods for the instance variables x and y
- Every class contains a getArea() method
  - However the implementation is different

- What if there was an easy way to allow us to reuse code instead of duplicating it in every class?

Inheritance

- Inheritance allows us to inherit (reuse) most of the code from one class and use it in another class
- The new class is similar to the original, but has a few differences

- There are multiple ways to think about this relationship:
  - Parent class/child class
  - Superclass/subclass
  - Base class/derived class

Inheritance

- There are two things not inherited by the subclass:
  - Constructors are not inherited - they need to be called from the subclass
  - Instance variables are not inherited - they are accessed through the accessor and modifier methods

Inheritance

- Inheritance is defined as an **is-a relationship**
  - You should always be able to say the child is-a parent:
    - Camry is-a Car
    - Dog is-a Mammal
    - Student is-a Person
- Java uses single inheritance
  - This means a child can only have one parent class
- A parent class can have multiple child classes

Examples Revisited

- We will revisit our previous examples to take advantage of inheritance
- We will use Shape as our parent class:

```
public class Shape {
   private int x, y;
   // constructors not shown
   // modifiers and accessors for y not shown
   public void setX(int xPos){
      x = xPos;
   }
   public int getX(){
      return x;
   }
}
```

Circle Class

- To use inheritance we use the keyword **extends**
- Notice how we are now only worried about what is unique about a circle in relation to a shape

```
public class Circle1 extends Shape {
   private double radius;
   // constructors not shown
   public double getArea(){
      return Math.PI * radius * radius;
   }
}
```

Rectangle & Triangle Classes

```
public class Rectangle1 extends Shape{
   private double width, height;

   // constructors not shown

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

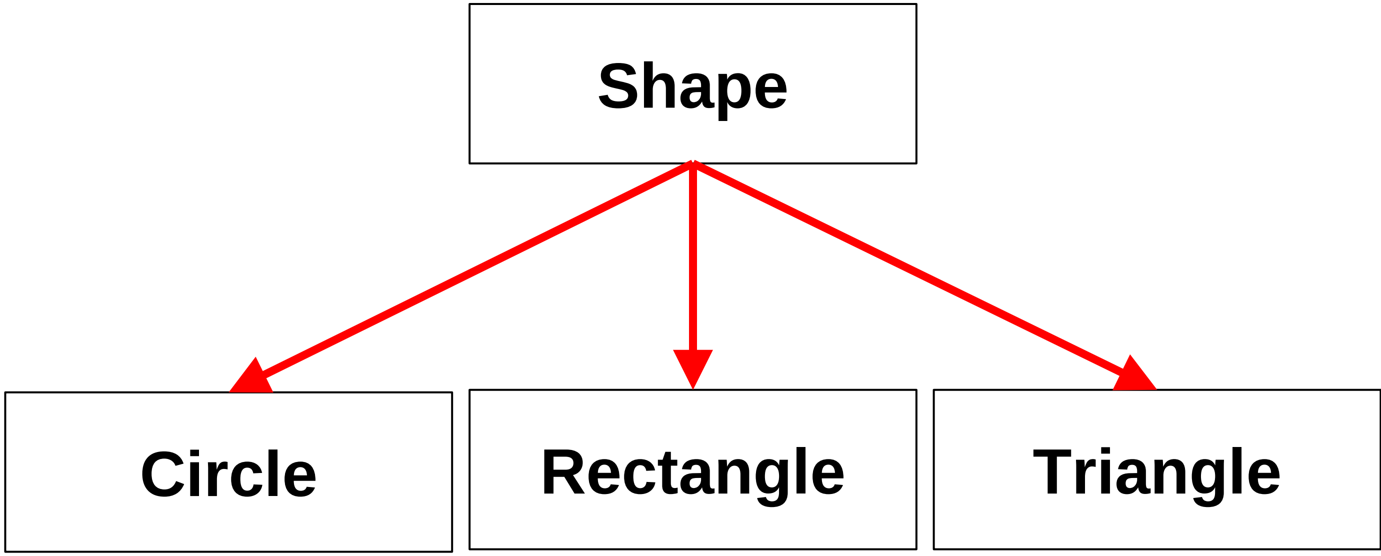
```
public class Triangle1 extends Shape{
   private double base, height;

   // constructors not shown

   public double getArea(){
      return 0.5 * base * height;
   }
}
```

Hierarchy for Shapes Example

Shape Class
Methods Inside

setX()
getX()
setY()
getY()

Instance Variables Inside
x y



Shape

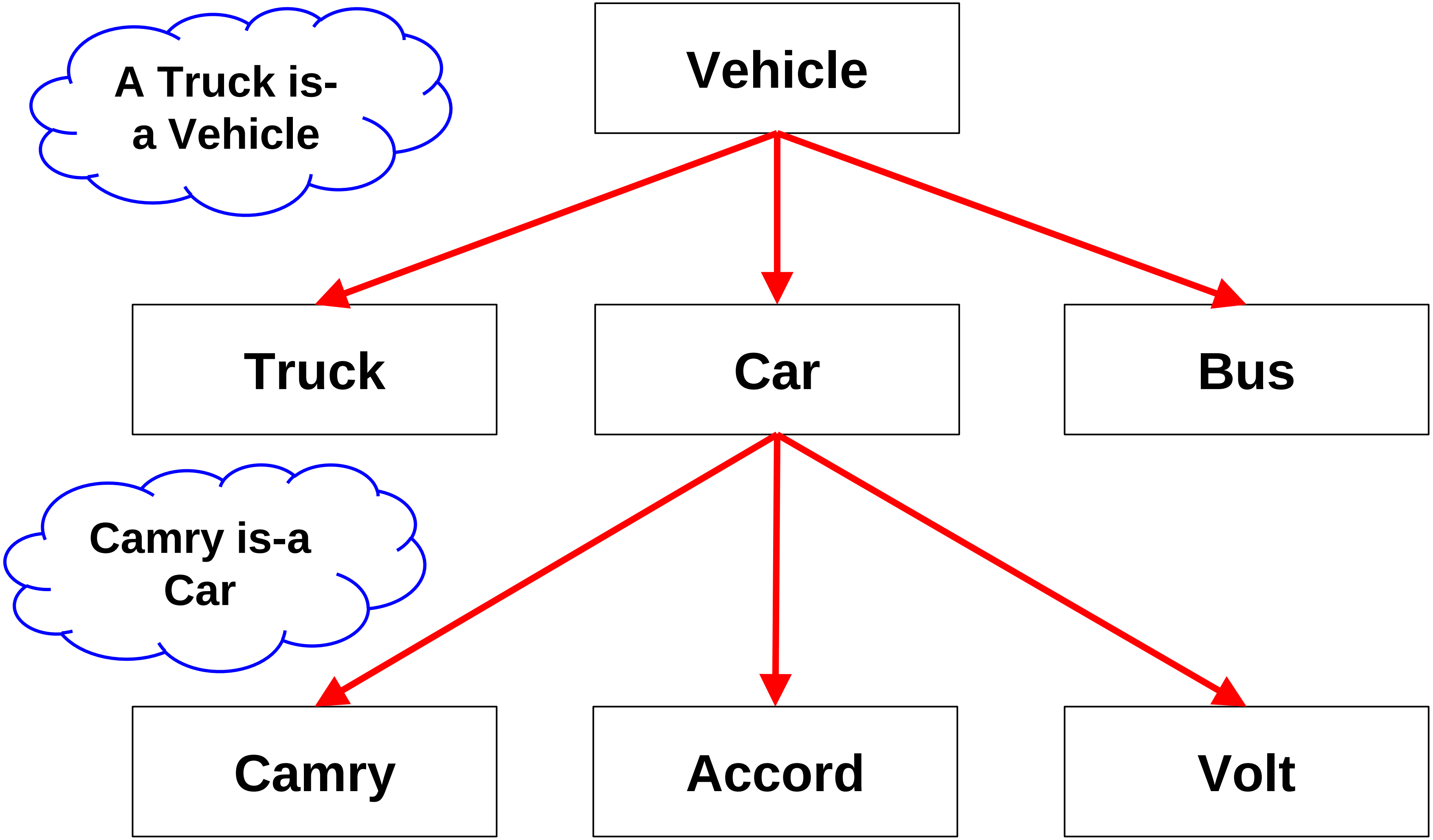
Circle

Rectangle

Triangle

Circle Class	Rectangle Class	Triangle Class
Methods Inside	**Methods Inside**	**Methods Inside**
getArea()	getArea()	getArea()
Instance Variables Inside	**Instance Variables Inside**	**Instance Variables Inside**
radius	width height	base height

Another Hierarchy Example



Inheritance - Behind the Scenes

Instantiating an Object

- In the examples so far we have excluded the constructors
- We will add the constructors to a couple of the classes to demonstrate how objects of the subclass are instantiated

Instantiating an Object

```
public class Shape2 {
   private int x, y;
   public Shape2(){
      x = y = 0;
   }
   public Shape2(int xPos, int yPos){
      x = xPos;
      y = yPos;
   }}
```

```
public class Circle2 extends Shape2 {
   private double radius;
   public Circle2(){
      radius = 0.0;
   }
   public Circle2(double r){
      radius = r;
   }}
```

Instantiating an Object

- Here is a constructor call:

```
// main of another class
Circle3 cir1 = new Circle3();
```

```
public class Circle3 extends Shape2 {
   private double radius;

   public Circle3(){
      super();
      radius = 0.0;
   }

   public Circle3(double r){
      super();
      radius = r;
   }}
```

- Everything should look great except this **super()** call?

Calling the Parent Constructor

- Remember in our Shape class we have two instance variables we are inheriting
- How do these two variables get instantiated?
- This is what the super() call does - it calls the constructor in the parent class

Calling the Parent Constructor

- If you do not provide a super() call Java will automatically call the default constructor in the parent class
- The super() call **must** be the first statement inside the child constructor!

Default Constructor

- Can anyone think of a potential problem with calling the default constructor?
- What happens if the parent class does not have a default constructor, but does have an initialization constructor?
- This is something to keep in mind, and a good reason to always provide a default constructor

Super Constructor Call

- We could also do something like this:

```
// main of another class
Circle4 cir1 = new Circle4(8.0, 50, 50);
```

```
public class Circle4 extends Shape2 {
   private double radius;

   public Circle4(){
      super();
      radius = 0.0;
   }

   public Circle4(double r, int xPos, int yPos){
      super(xPos, yPos);
      radius = r;
   }
}
```

Super Constructor Call

- Would this work?

```
// main of another class
Circle5 cir1 = new Circle5(8.0, 50, 50);
```

```
public class Circle5 extends Shape2 {
   private double radius;

   public Circle5(){
      super();
      radius = 0.0;
   }

   public Circle5(double r, int xPos, int yPos){
      radius = r;
      super(xPos, yPos);
   }
}
```

- No, the super() call must happen first

Super Constructor Call

- Would this work?

```
// main of another class
Circle6 cir1 = new Circle6();
```

```
public class Shape3 {
   private int x, y;
   public Shape3(int xPos, int yPos){
      x = xPos;
      y = yPos;
   }}
```

```
public class Circle6 extends Shape3 {
   private double radius;
   public Circle6(){
      radius = 0.0;
   }}
```

- No, there is no default constructor in Shape

toString()

- What is the output?

```
// main of another class
Circle7 cir1 = new Circle7();
System.out.println(cir1);
```

```
public class Shape4 {
   private int x, y;
   public String toString(){
      return "Shape toString()";
   }}
```

```
public class Circle7 extends Shape4 {
   private double radius;
   public String toString(){
      return "Circle toString()";
   }}
```

Output
Circle toString()

Super

- What is the output?

```
// main of another class
Circle7 cir1 = new Circle7();
System.out.println(cir1);
```

```
public class Shape4 {
   private int x, y;
   public String toString(){
      return "Shape toString()";
   }}
```

Output

```
Circle toString()
Shape toString()
```

```
public class Circle7 extends Shape4 {
   private double radius;
   public String toString(){
      return "Circle toString()\n" + super.toString();
   }}
```

Super

- super can be used to call any method or constructor in the parent class
    - super.toString();
    - super.setX(5);
    - super.getX();

This

- Here is a constructor call:

```
// main of another class
Circle8 cir1 = new Circle8();
```

```
public class Circle8 extends Shape {
   private double radius;
   public Circle8(){
      this(0.0);
   }
   public Circle8(double r){
      radius = r;
   }
}
```

- Any guesses on what **this(0.0)** does?

This

- What is the output?

```
// main of another class
Shape5 shape1 = new Shape5(5, 8);
System.out.println(shape1);
```

```
public class Shape5 {
   private int x, y;
   public Shape5(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public String toString(){
      return "" + "x: " + x + "\n" + "y: " + y;
   }
}
```

Output

```
x: 5
y: 8
```

This

- What is the output?

```
// main of another class
Shape6 shape1 = new Shape6();
System.out.println(shape1);
```

```
public class Shape6 {
   private int x, y;
   public Shape6() {
      this(0, 0);
   }
   public Shape6(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public String toString(){
      return "" + "x: " + x + "\n" + "y: " + y;
   }}
```

Output

```
x: 0
y: 0
```

This

- this can be used to call any method or constructor for the current object.
  - this.toString();
  - this.setX(5);
  - this.getX();
  - this();

Files from this presentation

- You can find any of the Java classes used as examples in this presentation **here**