Another polymorphic example

```java
Vehicle[] mover = new Vehicle[5];
mover[0] = new Vehicle(...);
mover[1] = new Plane(...);
mover[2] = new Plane(...);
mover[3] = mover[1];
```

The reference type may not be the same as the object type!

Accessing methods/variables through a polymorphic reference

```java
Dice d = new TrickDice(...);
```

Consider the reference type and object type:

1. Which type determines whether a method/variable can be accessed?
2. For an overridden method, which type determines which version gets invoked?

Client code:

```java
V x = new W();
System.out.println(x.num1); //?
System.out.println(x.num2); //?
x.vmethod(); //?
x.wmethod(); //?
```

Another polymorphic example

```java
Vehicle[] mover = new Vehicle[5];
mover[0] = new Vehicle(...);
mover[1] = new Plane(...);
mover[2] = new Plane(...);
mover[3] = mover[1];
```

The reference type may not be the same as the object type!

Accessing methods/variables through polymorphic references

The type of the reference determines the methods and fields that can be accessed

```java
class V {
    public int num1;
    public void vmethod() { num1++; }
}
class W extends V {
    public int num2;
    public void wmethod() { num2++; }
}
```

Client code:

```java
V x; // x references type V or its subtype
String s = "Which type, V or W? ";
System.out.print(s);
char input = Keyboard.readChar();
if (input == 'V')
    x = new V();
else
    x = new W();
System.out.println(x.num1); //?
System.out.println(x.num2); //?
x.vmethod(); //?
x.wmethod(); //?
```
Accessing *overridden* methods through polymorphic references

- The *type of the object* determines which version of the method gets invoked
- Class `Vehicle` has method `toString` that class `Plane` overrides:

```java
Vehicle v1 = new Vehicle(...);
Vehicle v2 = new Plane(...);
System.out.println(v1); //Vehicle's version
System.out.println(v2); //Plane's version
```

**instanceof**

- `instanceof` is an *operator* for determining when an instance is of (from) a particular class
- See example in class `House`

**The Object class**

If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the **`Object` class**

- ⇒ All classes are derived from the `Object` class

```java
class Room  
    // is the same as  
class Room extends Object  
```

**abstract class**

- A placeholder in a class hierarchy that represents a generic concept
- Cannot be instantiated
- Modifier: `abstract`
- Can contain abstract methods
  ```java
public abstract class Geometry  
public abstract double Area();  
```
- Subclasses of abstract classes will “fill out” these abstract methods

**The Object class**

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

- ⇒ All classes are derived from the `Object` class

```java
Object  
    // Room  
    // House  
    // Bathroom  
```

**The Object class**

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

- ⇒ All classes are derived from the `Object` class

```java
toString: "default" instance method defined in the `Object` class
Arrays are `Objects`, literally!
```