



Polymorphism

Important key to the power of O-O programming.

Three Pillars of O-O Programming

Encapsulation:

an object contains both data and the methods that operate on the data. It may **expose** some of these members to the outside and hide others.

This design separates the *public interface* from the *implementation*, and enforces **data integrity**.

Inheritance:

one class can inherit attributes and methods from another class.

Polymorphism:

a behavior can be invoked on different kinds of objects, **without knowing** the type of object that will perform the behavior

Polymorphism

Poly + morph = many + forms


*We can invoke a behavior (method) **without knowing** what kind of object will perform the behavior.*

Many kinds of objects can perform the same behavior.

```
Object x = new Double(3.14);  
x.toString(); // calls toString() of Double class
```

Polymorphism

```
x = new Date( );  
x.toString(); // calls toString() of Date class
```



Binding of Methods to References

For *instance methods*, Java decides at run-time which class's method to invoke.

- Called **dynamic binding** or **dynamic dispatch**.
- It means that you can not tell the **actual method** that will be called from only the **declared type** of a variable.

EXAMPLE: which toString() method will be invoked?

```
Object a = "What am I?"; // a -> String
// Compiler does not know what will happen here:
if (Math.random() > 0.5) a = new Date( );

// which toString will be used?
System.out.print( a.toString( ) );
```

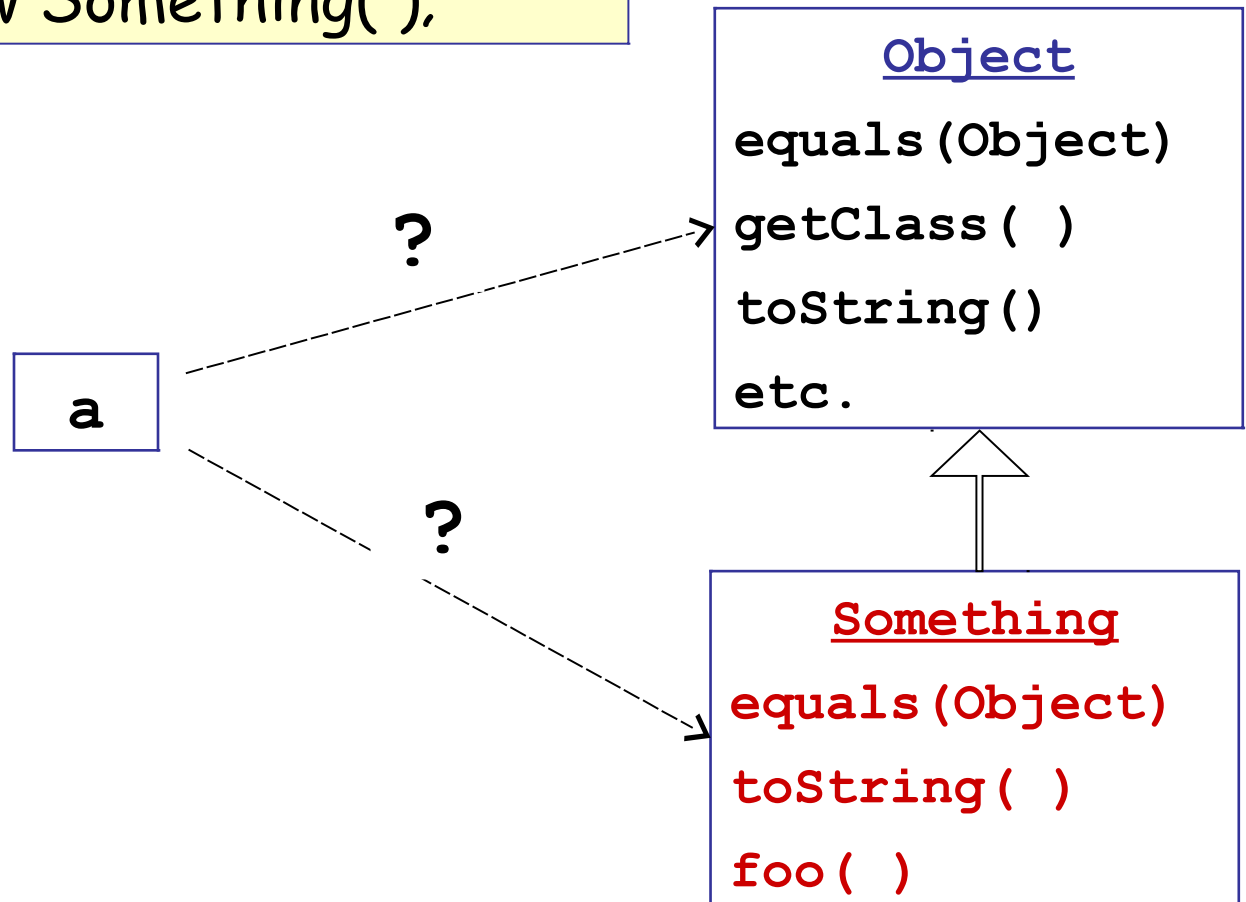
How Does this Work?

Object a = new Something();

a.toString() // what toString() will be invoked?

How Does this Work?

```
Object a = new Something( );
```



Enabling Polymorphism

The *key* to polymorphism invoking a method (asking an object to do something) **without knowing the *kind* of object.**

```
Object a = ? ;  
a.toString( ) ;  
a.run( ) ;
```

a.toString() always works for any kind of object. Why?

This is an **error**. a might not have a "run" method. Why?

- How can we invoke an object's method without knowing what class it belongs to?

Enabling Polymorphism

The *key* to polymorphism is invoking a method (ask an object to do something) **without knowing the *kind* of object.**

```
x.toString() //Guaranteed to have toString!  
x.run( )     // Does x have a run() method?
```

We must **guarantee** that the object referred by **x** will **have the method we want** to invoke.

Two Ways to Enable Polymorphism

In Java there are two ways to "**guarantee**" that a class has some behavior (method):

1. **Inheritance**

If a *superclass* has a method, then all *subclasses* **inherit** that method.

Subclass may also **override** it with their own implementation.

2. **Interface**

An interface **specifies** one or more methods.

A class that **implements an interface** **must** provide that method.

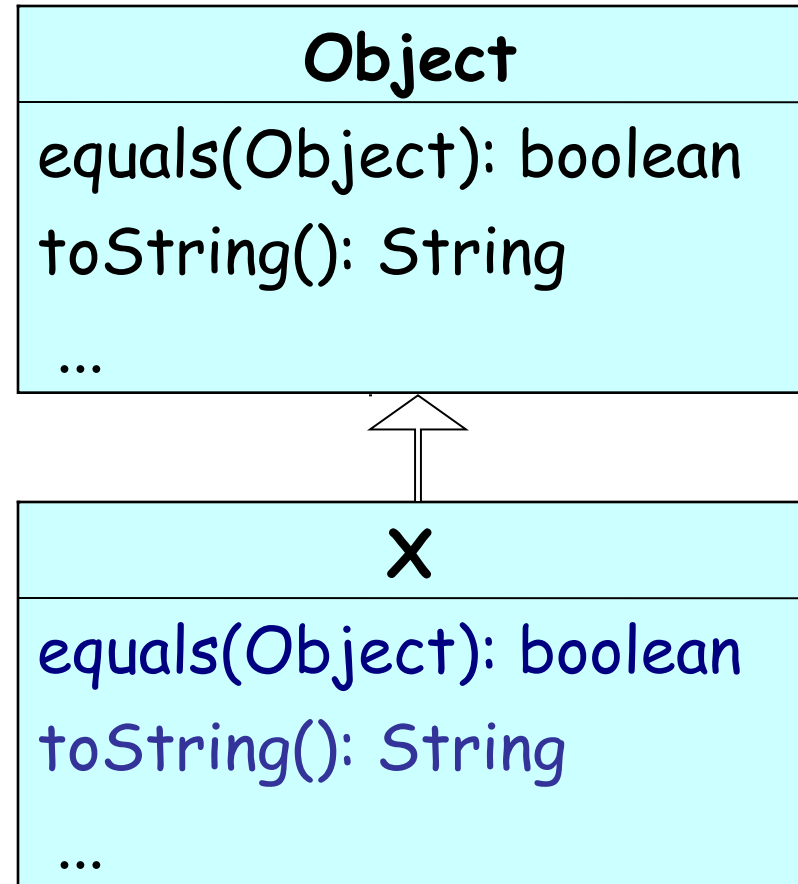
Inheritance and Polymorphism

Every class is a subclass of Object.

Therefore, every object is guaranteed to have all the methods from the Object class.

Every object is guaranteed to have equals(Object) and toString() methods.

We can invoke them *for any kind of object.*



Methods from Object

Every Java class is a **subclass** of the `Object` class.

Therefore, every object has the public methods from `Object`.

Usually, classes will override these methods to provide useful implementations.

Every class inherits these methods automatically.

So, we can *always* use `obj.toString()` or `obj.equals(obj2)` for any kind of object.

Object
<code>equals(Object): boolean</code>
<code>getClass(): Class</code>
<code>hashCode(): int</code>
<code>toString(): String</code>
<code>notify()</code>
<code>wait()</code>
...

Summary

Polymorphism in OOP means that many kinds of objects can provide the same behavior (method), and we can invoke that behavior without knowing which kind of object will perform it.

Don't ask 'what type?'

If you design for polymorphism correctly, you can invoke a method without testing an object's actual type.

So, polymorphism has the nickname:

"Don't ask what type"

Anti-polymorphism example:

```
public void repeat(Object task, int count) {
    if (task instanceof MyTask) {
        MyTask my = (MyTask) task;
        while(count-- > 0) my.run( );
    }
}
```

How does println() work?

`System.out.println(Object x)` can print **any kind** of object.

Even object types we define ourselves (like Student).

How is it possible for **one** method to "print" any kind of object?

any kind of Object

```
Object x = new Something( );  
System.out.println( x ); // prints String form of x
```




Intro to Interfaces

Interfaces are a "pure" way to enable polymorphism in Java.

Not required for Programming 1.

Interface

Interface is a *specification* for some required behavior, *without an implementation*.

A Java *interface* specifies behavior which will be provided by classes that *implement* the interface.

Example: USB interface specifies (a) connector size, (b) electrical properties, (c) communications protocol, ...

Anyone can *implement* the USB interface on their device.

We can use *any* USB port the same way, without knowing the actual type (manufacturer) of the device.

java.lang.Runnable interface

```
public interface Runnable {  
    /**  
     * The method to invoke.  It doesn't  
     * return anything.  
     */  
    public void run( );  
}
```

abstract method = method signature only, no implementation

Runnable example

Declare that this class has the run() behavior.

```
public class MyTask implements Runnable {  
    /** The required method. */  
    public void run() {  
        Implement the required method.  
        System.out.println("I'm running!");  
    }  
}
```

Use the interface in an app

```
public class TaskRunner {  
    /**  
     * Run a task n times.  
     * @param task a Runnable to perform  
     * @param count number of time to do it.  
     */  
    public void repeat(Runnable task, int count)  
    {  
        while(count > 0) {  
            task.run( );  
            count--;  
        }  
    }  
}
```

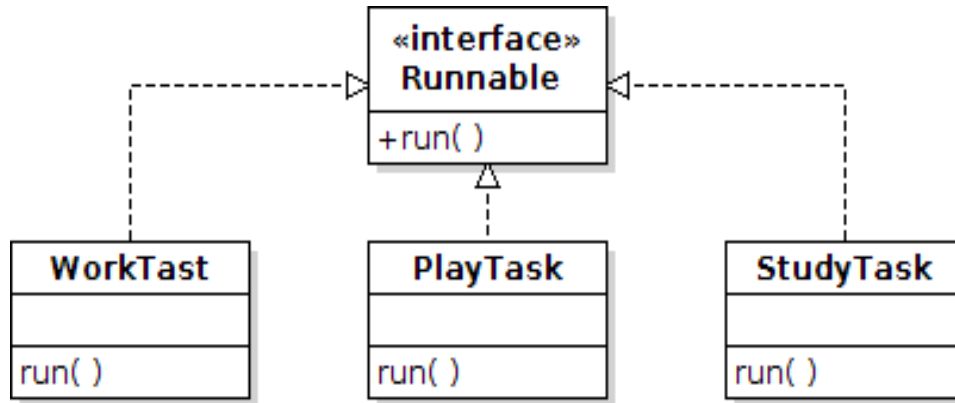
Example: print message 5 times

```
TaskRunner runner = new TaskRunner();  
Runnable mytask = new MyTask( );  
  
runner.repeat(mytask, 5);
```

```
I'm running.  
I'm running.  
I'm running.  
I'm running.  
I'm running.
```

How does Interface enable Polymorphism?

We can define many tasks that implement *Runnable*.



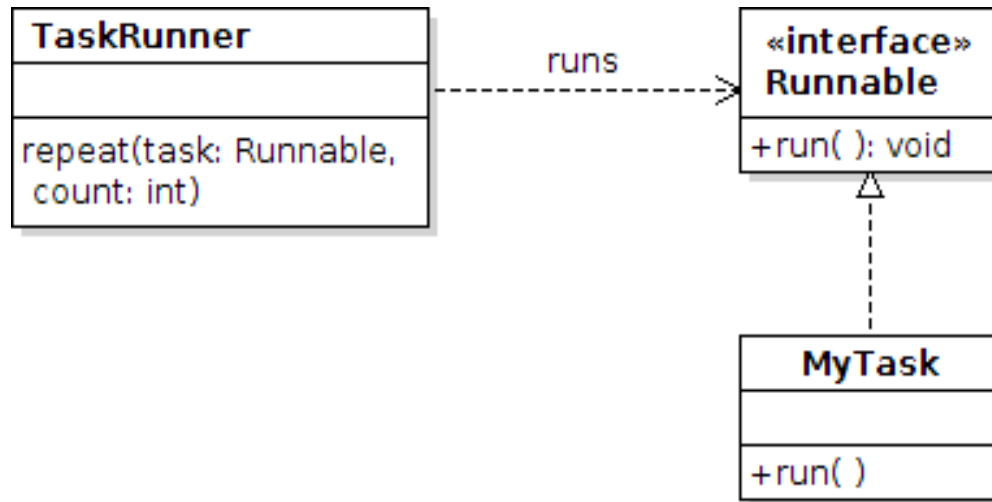
TaskRunner can use *any task* without knowing its type.
Every task is *guaranteed* to have a run() method.

```
Runnable task = null;
if (today()==SUNDAY) task = new PlayTask( );
else task = new StudyTask( );
runner.repeat( task, 3 );
```

UML for interface

UML class diagram for this example.

Notice that TaskRunner does not depend on MyTask.



Make MyTask more *flexible*

Modify **MyTask** so we can use it to print any message.

```
public class MyTask implements Runnable {  
  
    ???  
  
    /** @see java.lang.Runnable#run() */  
    public void run() {  
        System.out.println("_โฆษณาที่นี่: 02-9428555_");  
    }  
}
```


Solution

Modify **MyTask** so we can use it to print any message.

```
public class MyTask implements Runnable {
    private String message;
    /** @param message is the message to print */
    public MyTask(String message) {
        this.message = message;
    }

    /** @see java.lang.Runnable#run() */
    public void run() {
        System.out.println("message");
    }
}
```