Polymorphism

Polymorphism

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.

Poly - morph = many forms

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

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

Polymorphism

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
```

println( ) uses Polymorphism

`println(x)` calls `x.toString()`.

println(x) doesn't know the type of object that x refers to.

Using *polymorphism*, x.toString() always invokes toString() of the correct class.

```
Object x = new Date( );
System.out.println( x ); // calls x.toString()
     // invokes toString of Date class

x = new Student("Bill Gates");
System.out.println( x );
     // invokes toString of Student class
```

Enabling Polymorphism

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

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

a.toString() <u>always</u> 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 <u>class</u> it belongs to?

Enabling Polymorphism

The compiler has to "know" that x will always have the requested method... regardless of the actual type of x.

```
x.run( ) // huh? Does x have a run()?
```

We must guarantee that different kinds of objects 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 its *subclasses* inherit that method.

   Subclasses can use the parent's method, or ***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.

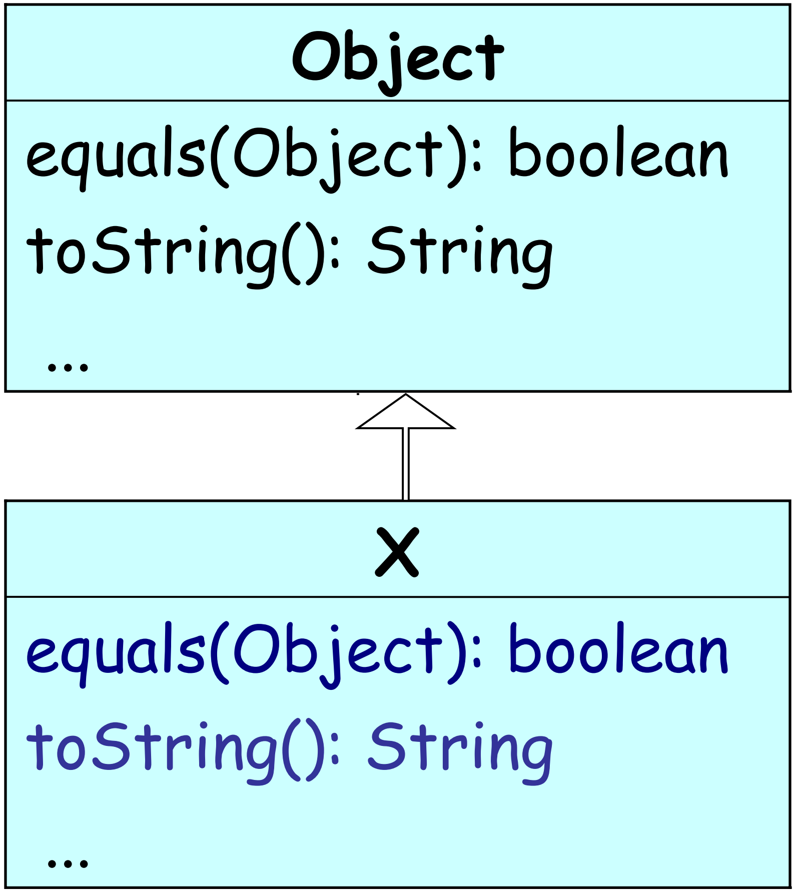
Inheritance and Polymorphism

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.*



Object

equals(Object): boolean

toString(): String

...

X

equals(Object): boolean

toString(): String

...

Methods from Object

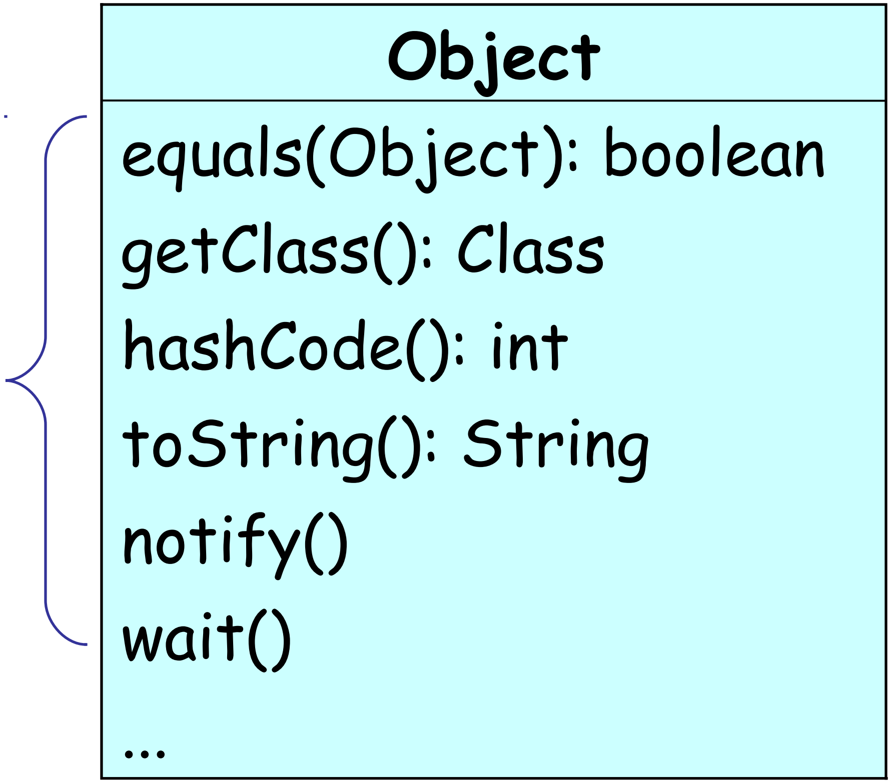
Every Java class is a subclass of the `Object` class.

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

equals(Object): boolean

getClass(): Class

hashCode(): int

toString(): String

notify()

wait()

...

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.
   * @see java.lang.Runnable#run()
   */
  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() {

    System.out.println("I'm running!");
  }
}
```

Implement the required method.

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

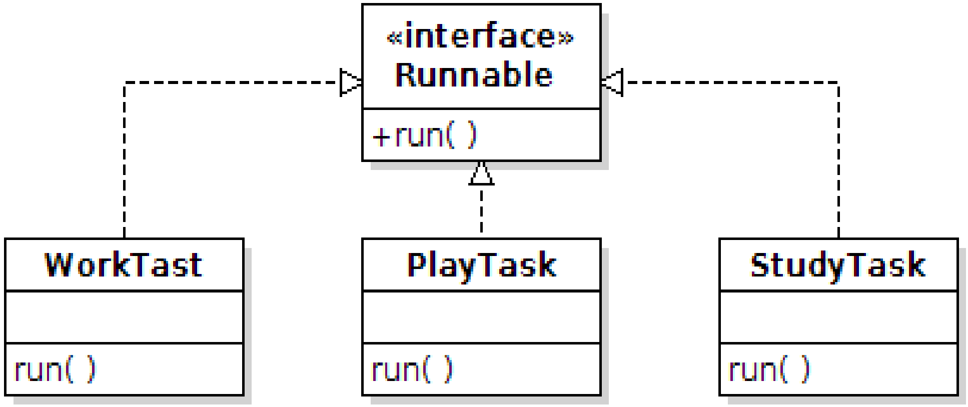
```
TaskRunner runner = new TaskRunner();
Runnable mytask = new MyTask( );

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*.



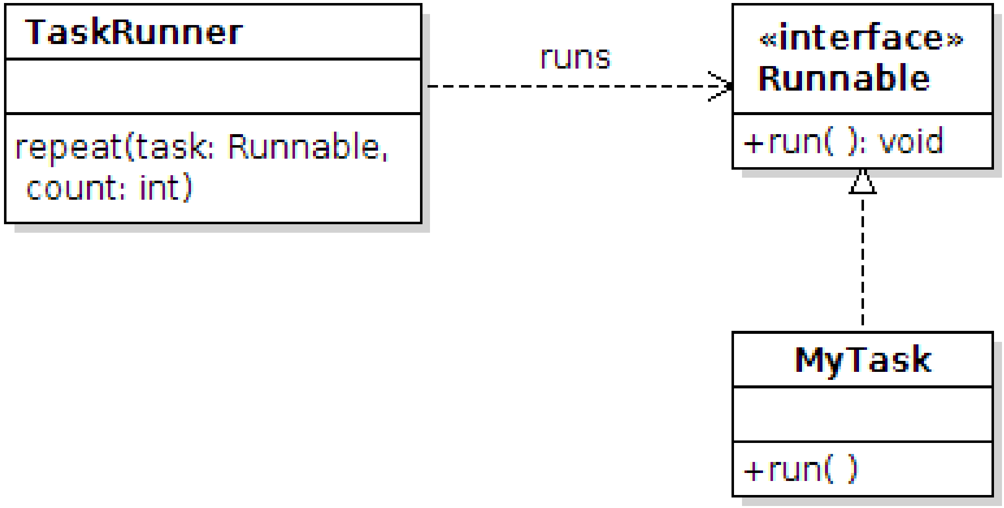
TaskRunner can use *any task* without knowing its type.
Every task is guaranteed to have a run( ) method.

```
Runnable task = null;
if (time < 1700) task = new StudyTask( );
else task = new PlayTask( );
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 );
  }
}
```

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.

Enabling Polymorphism

To use polymorphism, we must *guarantee* that the object **`x`** refers to has the method we want to invoke.

`x.toString( )` // x must have a toString method

- How can we guarantee this??

1. Use inheritance: a subclass has all the methods of its superclass.

2. Use an Interface: an interface specifies a required behavior without implementing it.

Every class that implements the interface is promising that it provides the interface's behavior.

Don't ask "what type?"

With polymorphism, we can invoke a behavior without knowing the type (class) of object that will perform the behavior. We don't test for the type of object.

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( );
    }
```