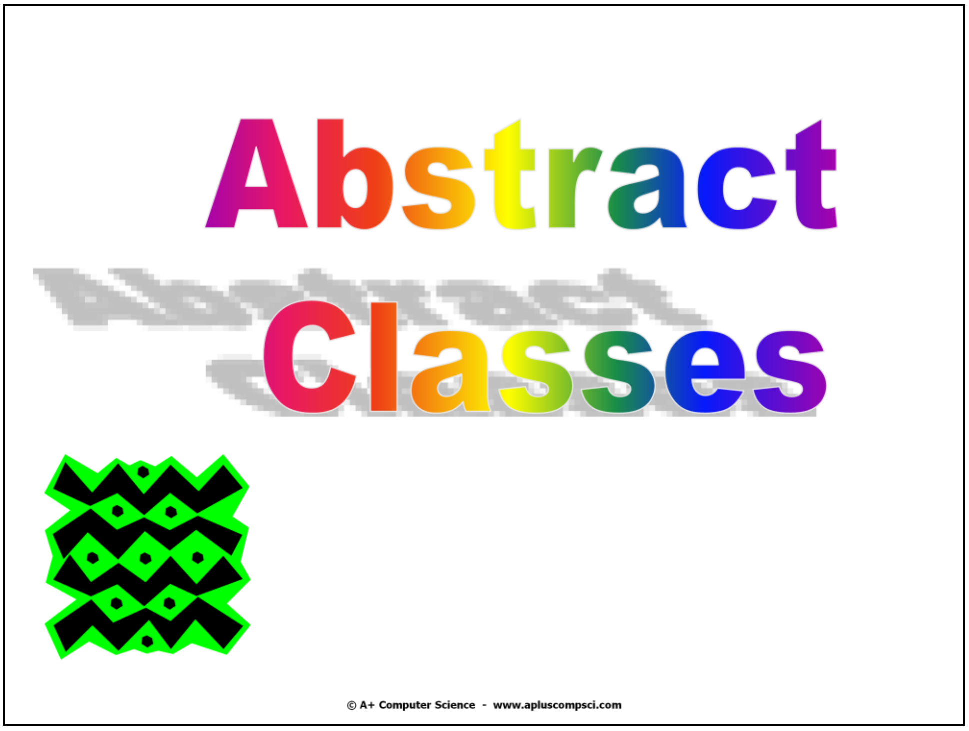
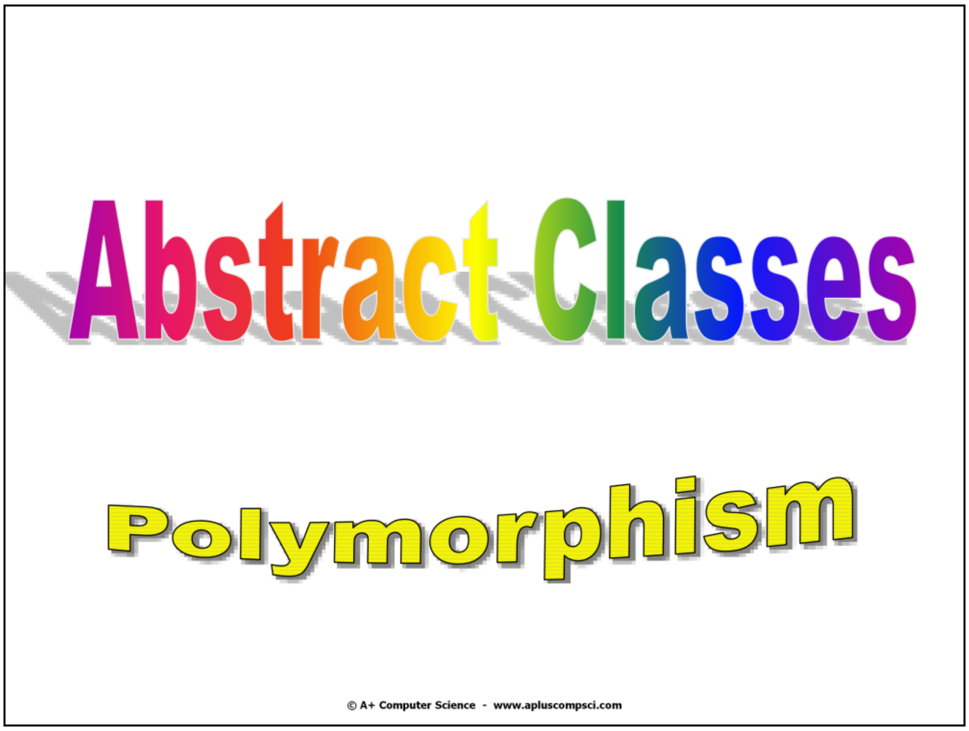
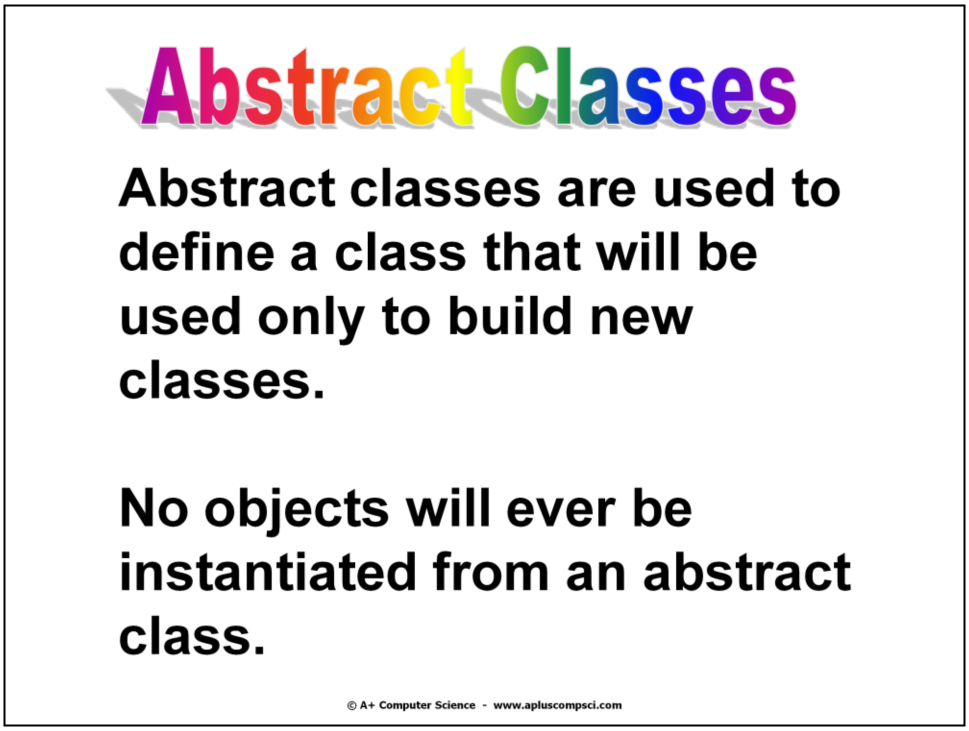


Abstract Classes



Abstract Classes

Polymorphism



Abstract Classes

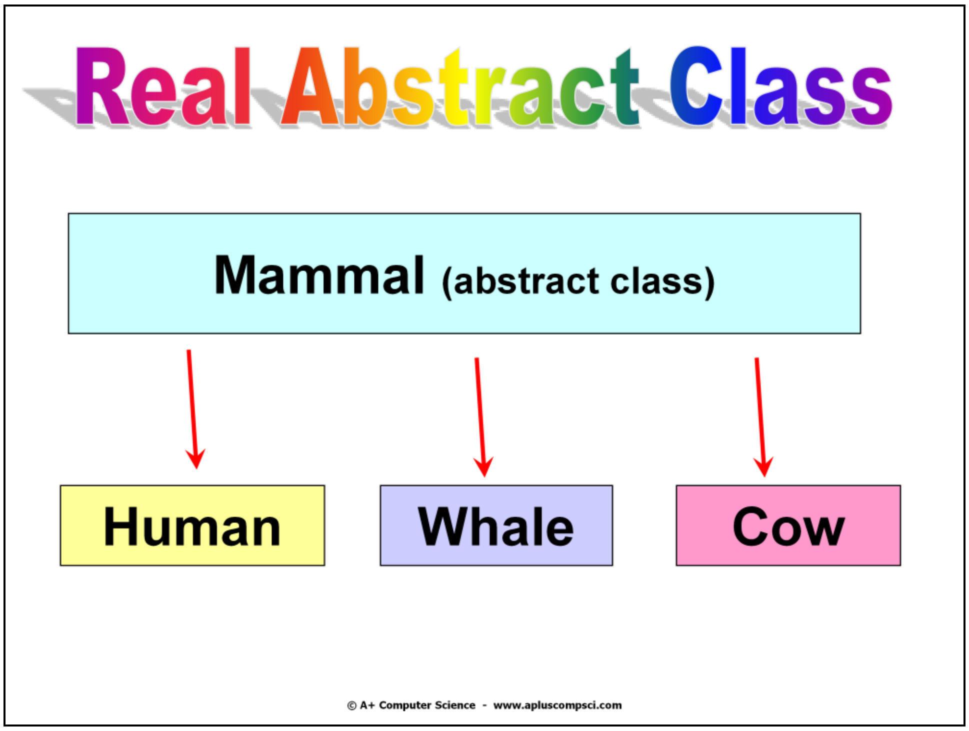
Abstract classes are used to define a class that will be used only to build new classes.

No objects will ever be instantiated from an abstract class.

© A+ Computer Science - www.apluscompsci.com

Abstract Classes are used to create hierarchies of classes.
Abstract classes are used to setup future classes.

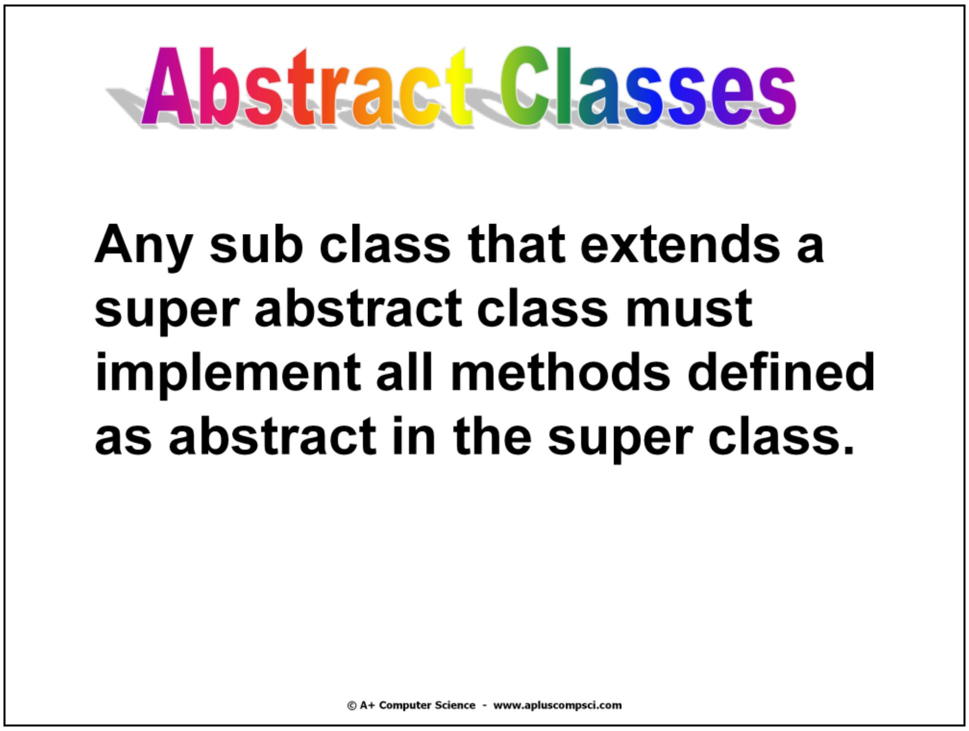
An abstract class can not be instantiated.



Real Abstract Class

© A+ Computer Science - www.apluscompsci.com

In this example, Mammal is the abstract class. You would never have just a Mammal. For instance, you would not walk outside and go "Hey, look at that Mammal!" Mammal would be used to create something more specific, like a Dog, Human, or Whale.



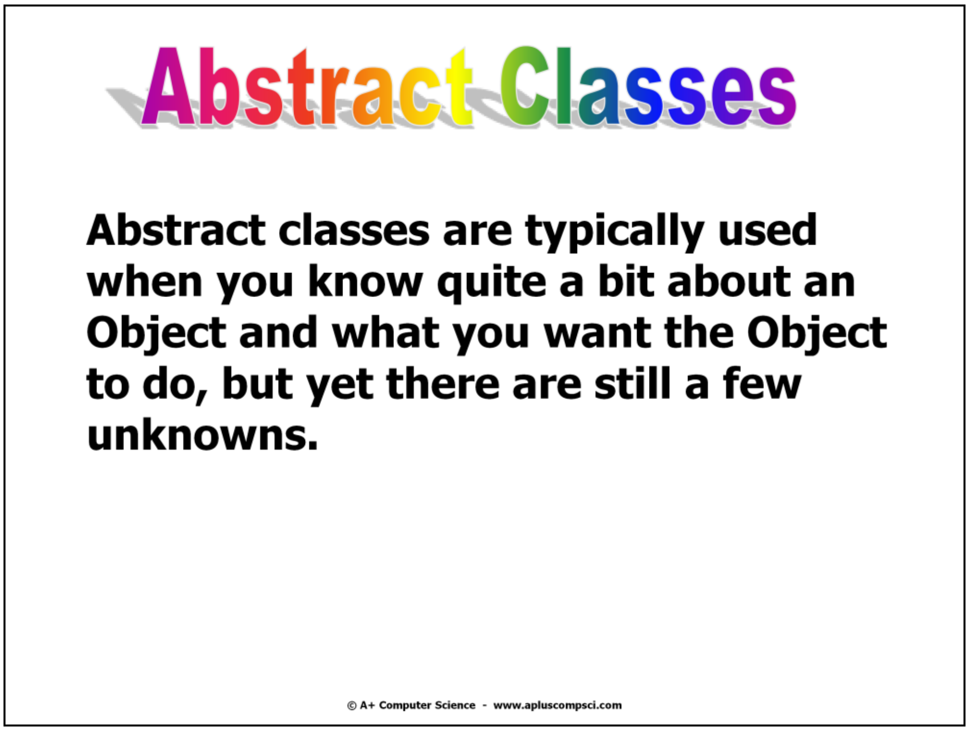
Abstract Classes

Any sub class that extends a super abstract class must implement all methods defined as abstract in the super class.

© A+ Computer Science - www.apluscompsci.com

All abstract methods in the Abstract class must be implemented by the sub class extending the abstract class.

This process is very similar to implementing an interface. When implementing an interface, all abstract methods in the interface must be implemented in the class. All methods in an interface are Abstract.



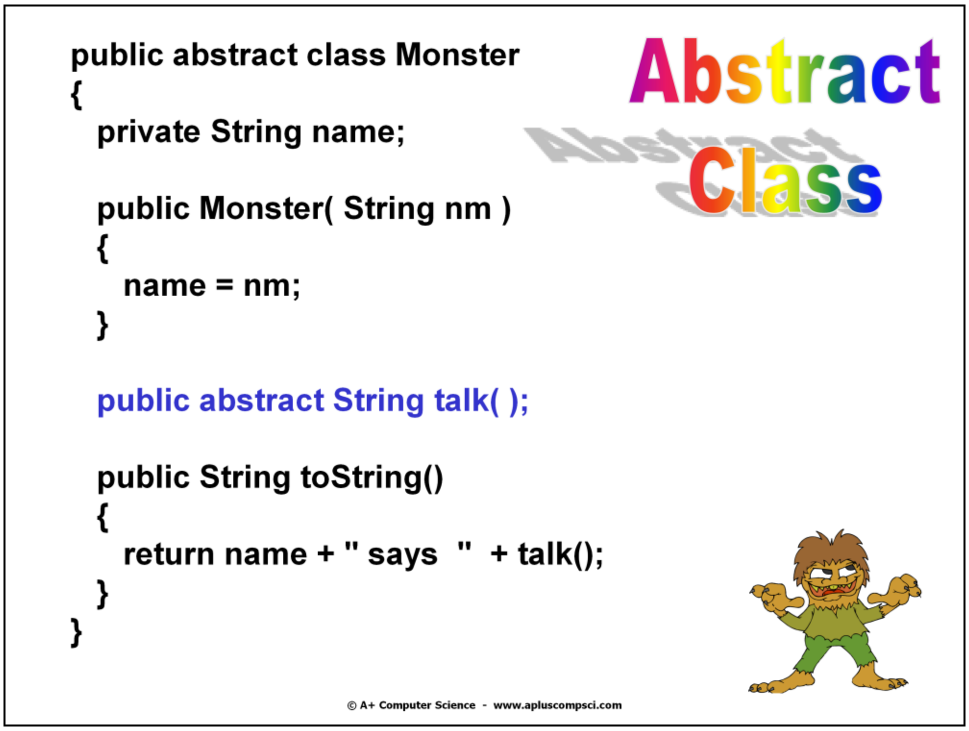
Abstract Classes

Abstract classes are typically used when you know quite a bit about an Object and what you want the Object to do, but yet there are still a few unknowns.

Abstract classes are great when you know quite a bit about an Object, but not everything.

For instance, say you are writing a program to simulate flying objects and you know that all of the objects will have width, height, and speed, but you do not know how each one will fly.

The fly properties and behavior will differ for each type of flying object which means that all of the common properties and behaviors that are already known can be placed in an abstract class. The specific fly methods will be implemented in each of the specific flying objects.



```
public abstract class Monster
{
   private String name;

   public Monster( String nm )
   {
      name = nm;
   }

   public abstract String talk( );

   public String toString()
   {
      return name + " says  " + talk();
   }
}
```

Abstract
Class
© A+ Computer Science - www.apluscompsci.com

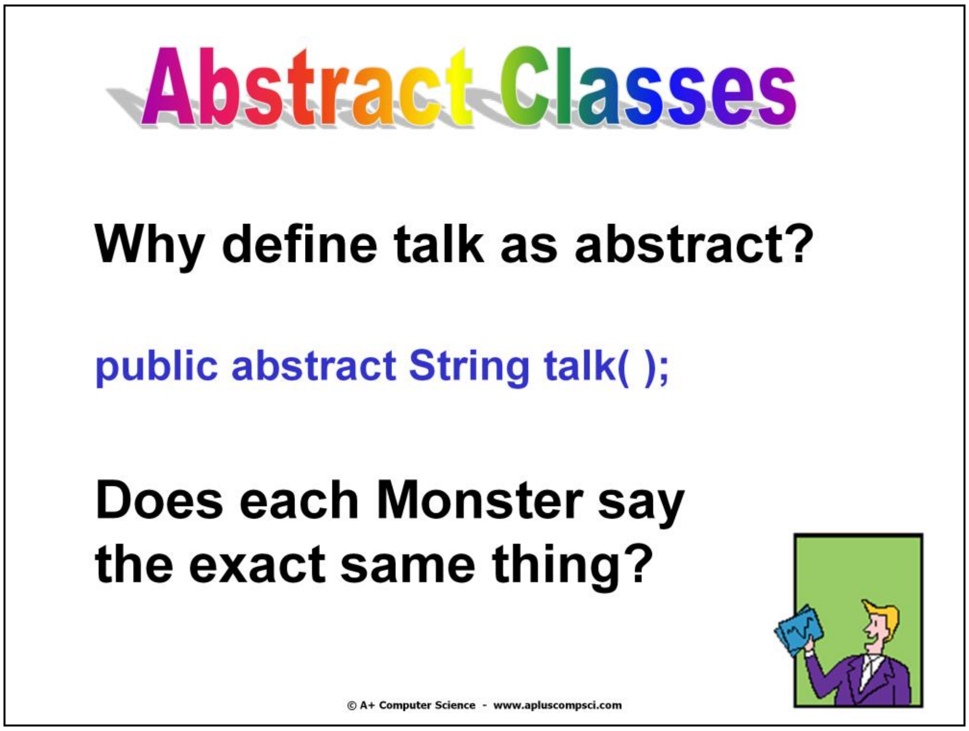
Monster is an abstract class.

```
Monster x = new Monster();      //illegal
Monster y = new Ghost();        //legal
```

Monster cannot be instantiated.

Monster contains one abstract method, method `talk();`

The assumption is that all Monsters will have a name and name related methods. Also, it is assumed that all Monsters will talk. There is no way to know at the Monster level what exactly a particular type of Monster will say. The `talk()` method is abstract because it makes no sense it implement `talk()` at the Monster level. It makes more sense to write `talk()` for each specific Monster as each specific Monster will say something specific.



Abstract Classes

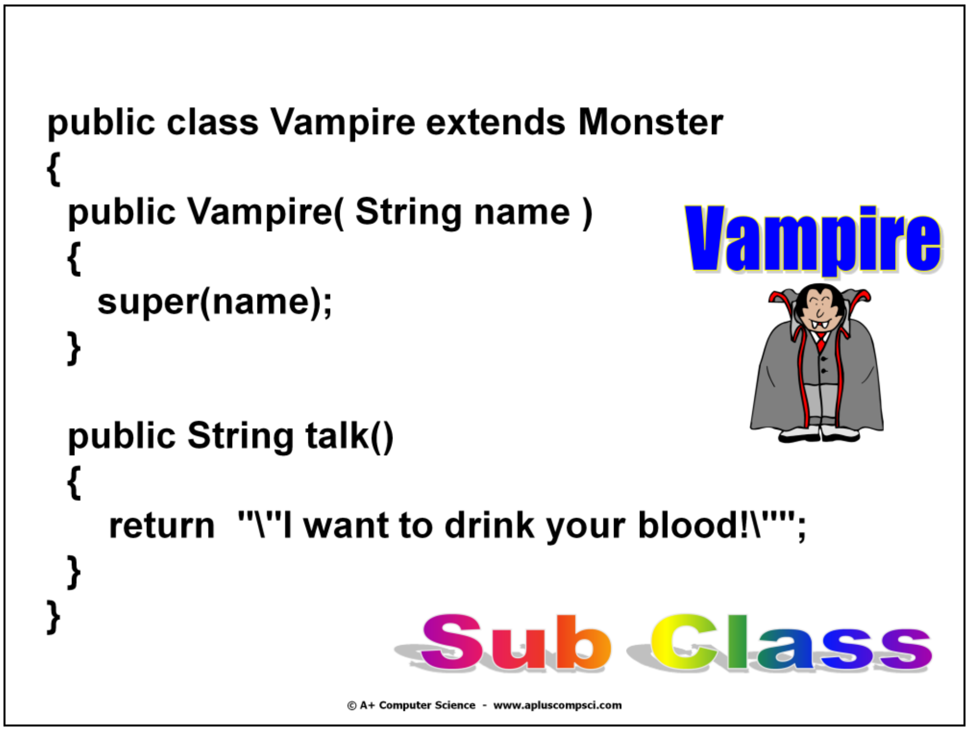
Why define talk as abstract?

```
public abstract String talk( );
```

Does each Monster say the exact same thing?

Each Monster will say something specific. A Ghost will say something different than a Werewolf and a Yetti will say something different from a Vampire.

Thus, the `talk()` method is abstract as it will be written in each of the specific Monsters in a way specific to each Monster.



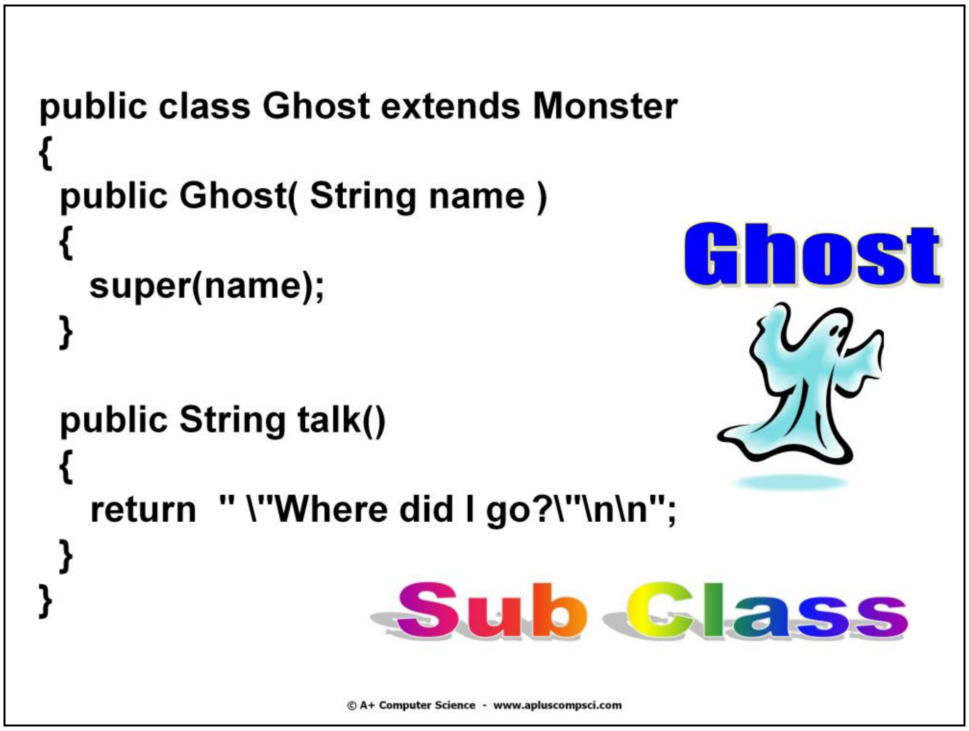
```
public class Vampire extends Monster
{
  public Vampire( String name )
  {
     super(name);
  }

  public String talk()
  {
     return  "\"I want to drink your blood!\"";
  }
}
```

Vampire
Sub Class
© A+ Computer Science - www.apluscompsci.com

Vampire is a Monster, but it is a Monster that talks in its own specific way.

A Vampire says “I want to drink your blood!”.

Notice that the talk() method was called in the Monster class before it was even implemented.
This is a very good example of polymorphism.



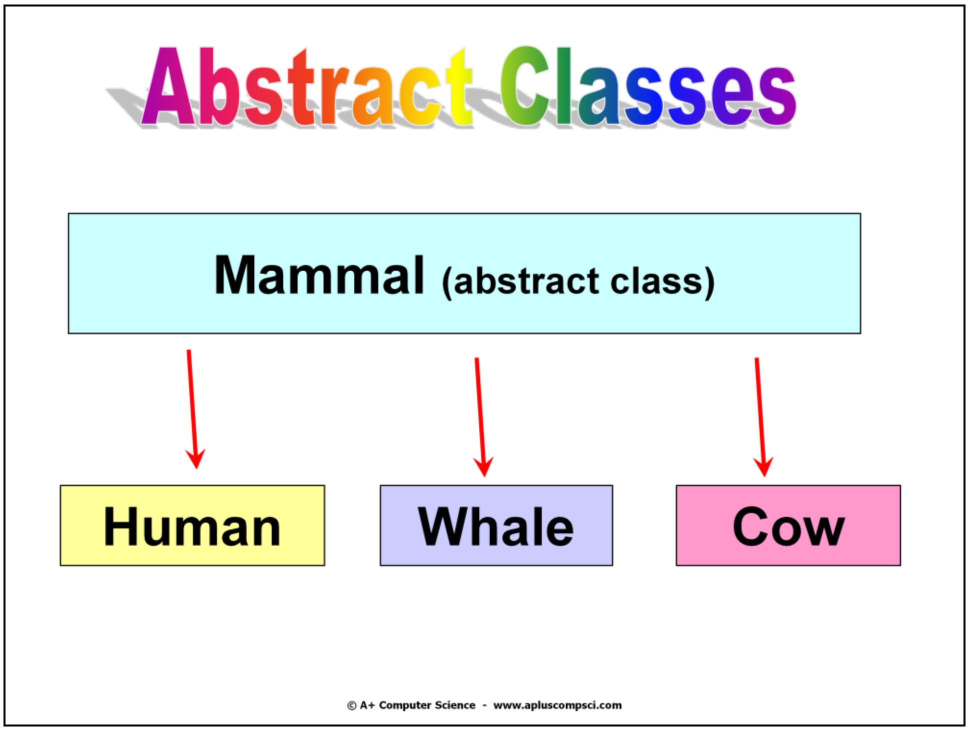
```
public class Ghost extends Monster
{
  public Ghost( String name )
  {
    super(name);
  }

  public String talk()
  {
    return  " \"Where did I go?\"\n\n";
  }
}
```

Ghost
Sub Class
© A+ Computer Science - www.apluscompsci.com

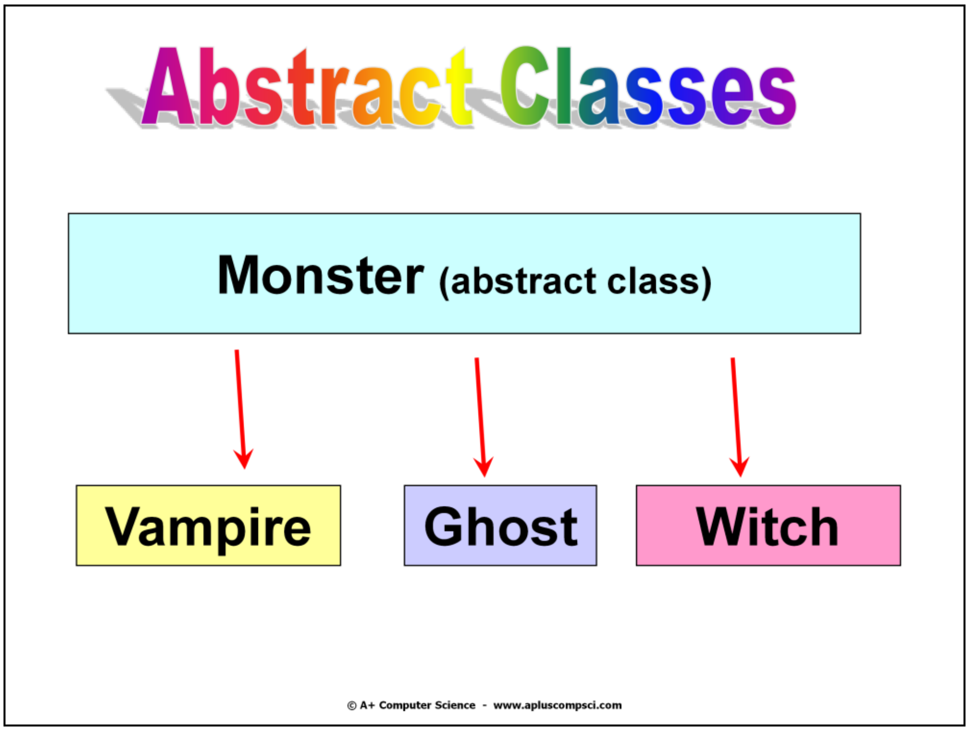
Ghost is a Monster, but it is a Monster that talks in its own specific way.

A Ghost says “Where did I go?”.

Notice that the talk() method was called in the Monster class before it was even implemented.
This is a very good example of polymorphism.


Abstract Classes

In this example, Mammal is the abstract class. You would never have just a Mammal. For instance, you would not walk outside and go “Hey, look at that Mammal!” Mammal would be used to create something more specific, like a Dog, Human, or Whale.



In this example, Monster is the abstract class. You would never have just a Monster. Monster would be used to create something more specific.

A Vampire is a specific type of Monster.

A Ghost is a specific type of Monster.

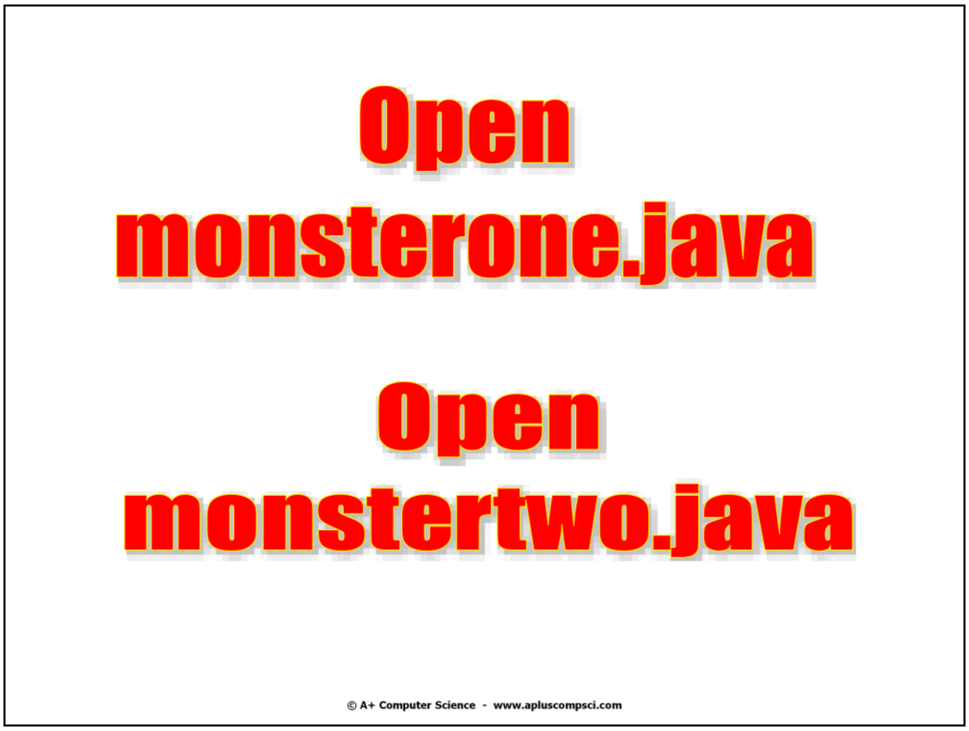
A Witch is a specific type of Monster.

A Skeleton is a specific type of Monster.

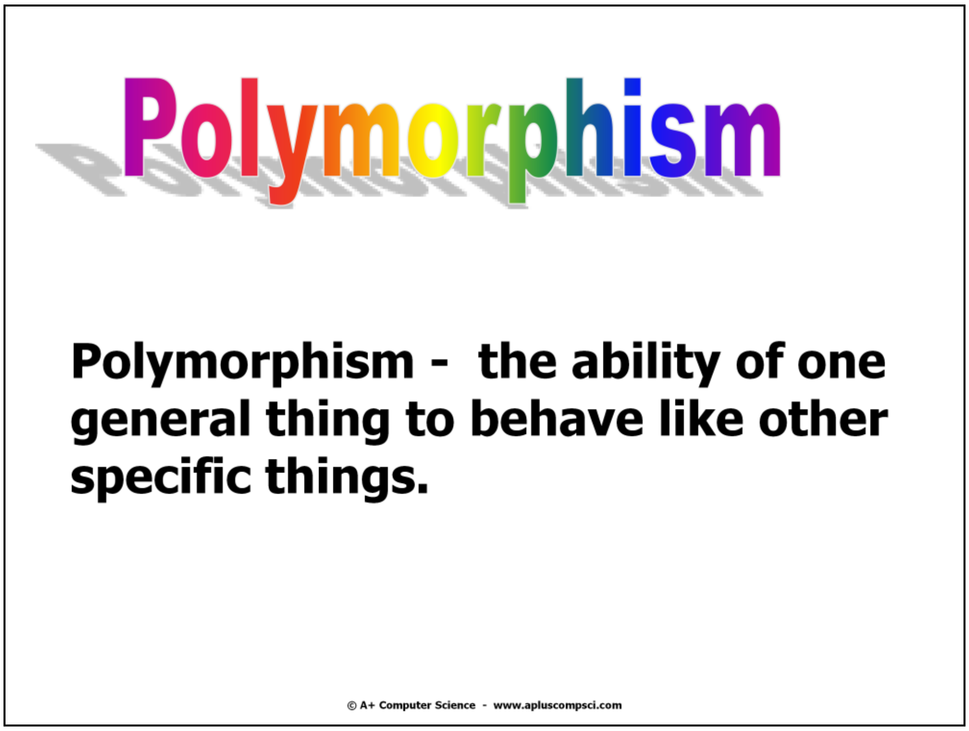
A Sasquatch is a specific type of Monster.

A Student is a specific type of Monster.

Open monster.java ghost.java


Open
monsterone.java

Open
monstertwo.java



Polymorphism

Polymorphism - the ability of one general thing to behave like other specific things.

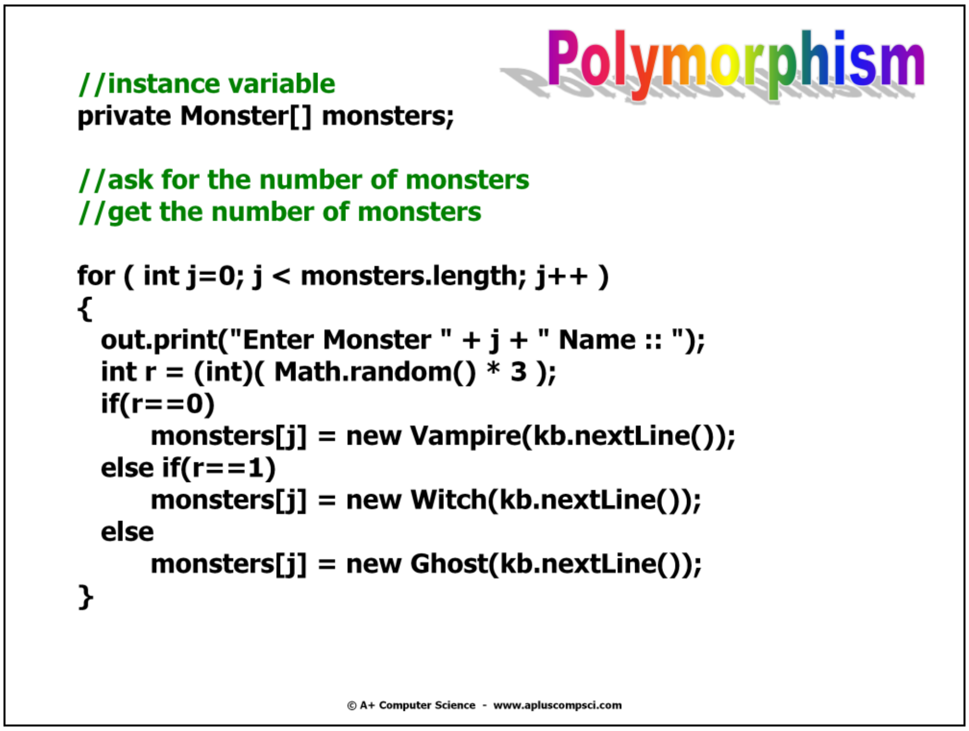
© A+ Computer Science - www.apluscompsci.com

Polymorphism is seen in Java when a general reference is used to refer to a more specific child.

Actor a = new Critter();

a = new Flower();

Having a defined as an Actor allows a to refer to any of its children.



Polymorphism

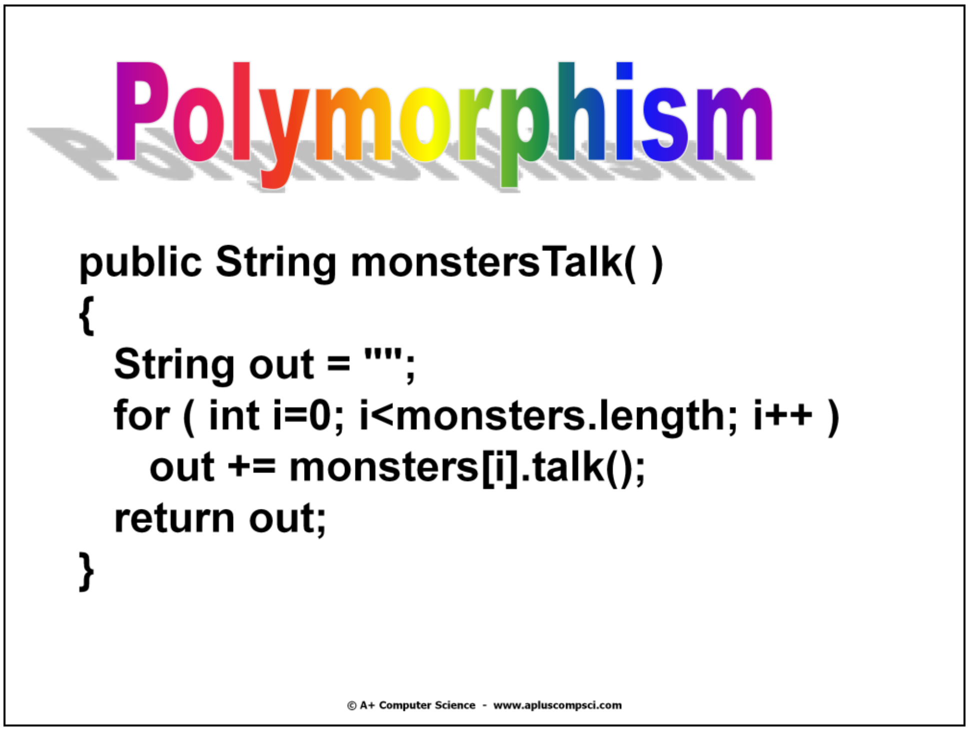
```
//instance variable
private Monster[] monsters;

//ask for the number of monsters
//get the number of monsters

for ( int j=0; j < monsters.length; j++ )
{
  out.print("Enter Monster " + j + " Name :: ");
  int r = (int)( Math.random() * 3 );
  if(r==0)
      monsters[j] = new Vampire(kb.nextLine());
  else if(r==1)
      monsters[j] = new Witch(kb.nextLine());
  else
      monsters[j] = new Ghost(kb.nextLine());
}
```

In this section of code, random Monsters are loaded into the monsters array.

Monsters is an array of Monster references.

Monsters can refer to any type of Monster.

Vampire, Witch, and Ghost extend Monster; thus, they are Monster Objects.
The monsters array can refer to any of these as they all extend Monster.

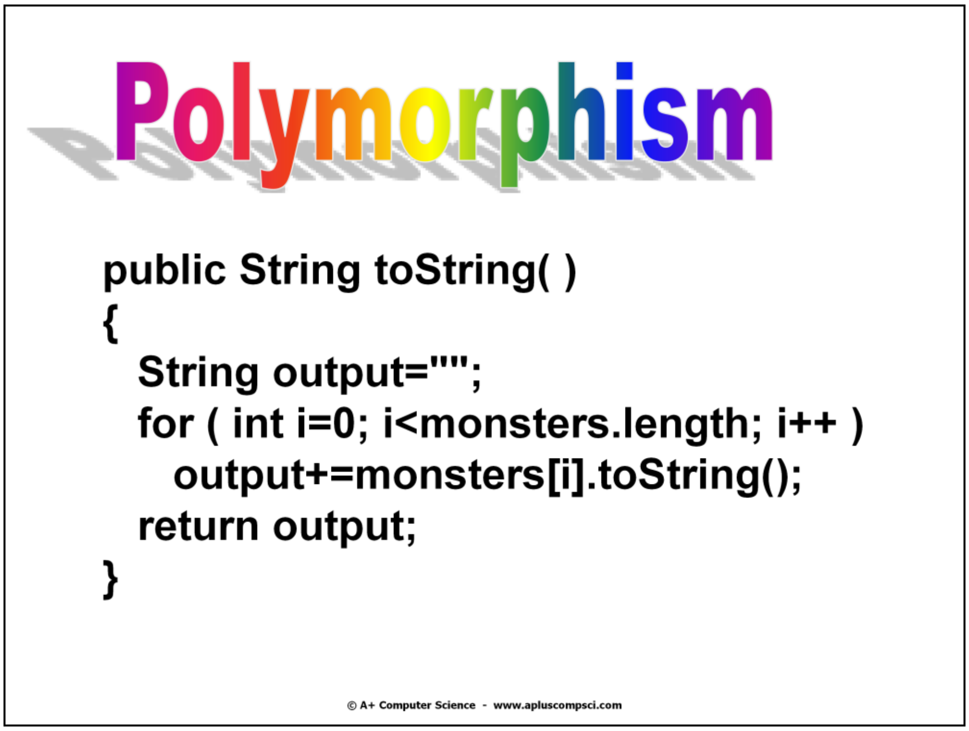


Polymorphism

```
public String monstersTalk( )
{
   String out = "";
   for ( int i=0; i<monsters.length; i++ )
      out += monsters[i].talk();
   return out;
}
```

© A+ Computer Science - www.apluscompsci.com

The `monstersTalk()` method will make each Monster talk. Java does not need to know which type of Monster each reference refers to. Java knows that each reference is a Monster and that Monster has a `talk()` method. Java will dynamically call the appropriate Monster `talk()` for each specific type of Monster at run-time. This is a very good example of dynamic binding.

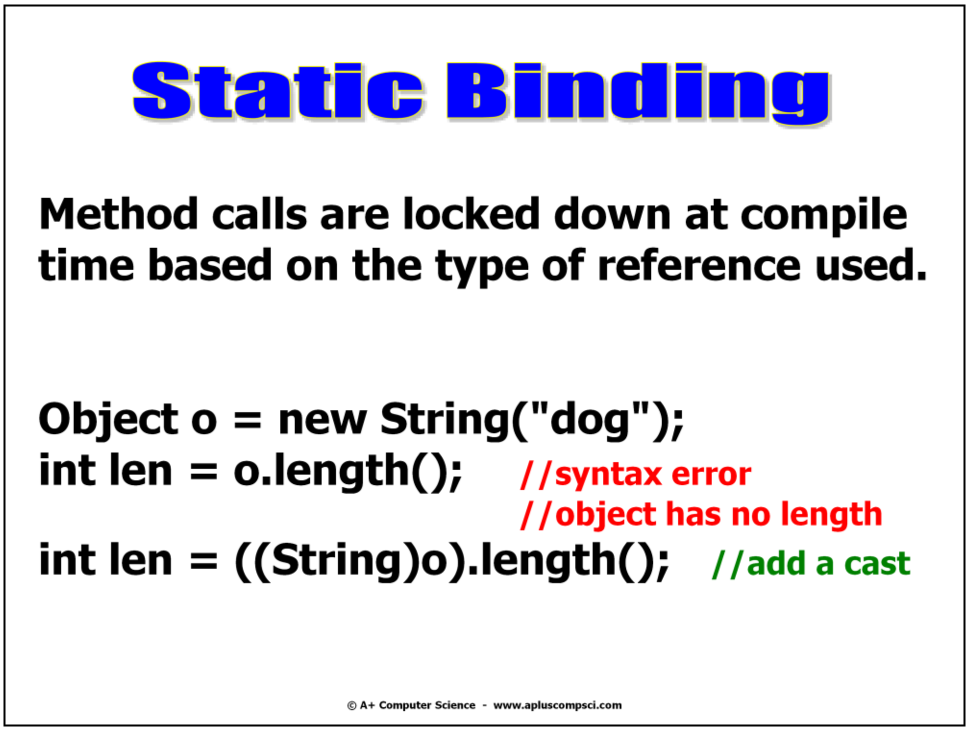


Polymorphism

```
public String toString( )
{
   String output="";
   for ( int i=0; i<monsters.length; i++ )
      output+=monsters[i].toString();
   return output;
}
```

© A+ Computer Science - www.apluscompsci.com

The `toString()` method will call each Monsters `toString()`. Java does not need to know which type of Monster each reference refers to. Java knows that each reference is a Monster and that Monster has a `toString()` method because all Objects automatically extend class Object which has a `toString()`. Java will dynamically call the appropriate Monster `toString()` for each Monster at run-time. This is another very good example of dynamic binding.



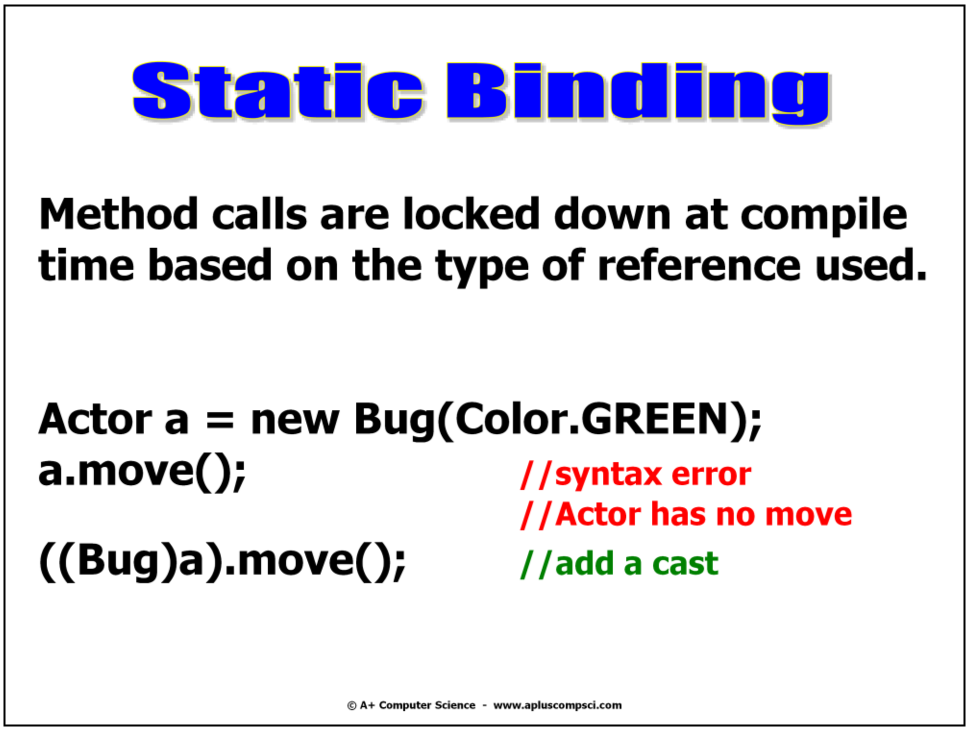
Static Binding

Method calls are locked down at compile time based on the type of reference used.

```
Object o = new String("dog");
int len = o.length();      //syntax error
                           //object has no length
int len = ((String)o).length();   //add a cast
```

Static binding is basically just matching up a class with its list of methods at compile time.   Java makes sure that class x has method y.

Dynamic binding occurs dynamically at run-time. Java will call the appropriate method for a particular class on the fly. The same line of code will be used to call each different Objects method.   The method names are the same, but the Objects will differ.



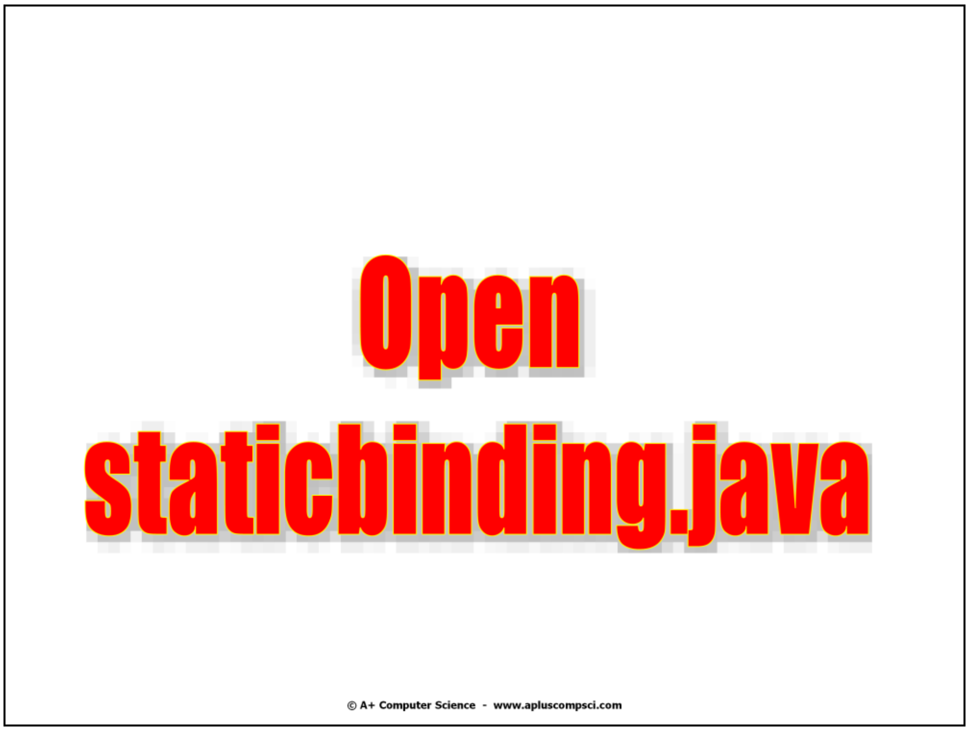
Static Binding

Method calls are locked down at compile time based on the type of reference used.

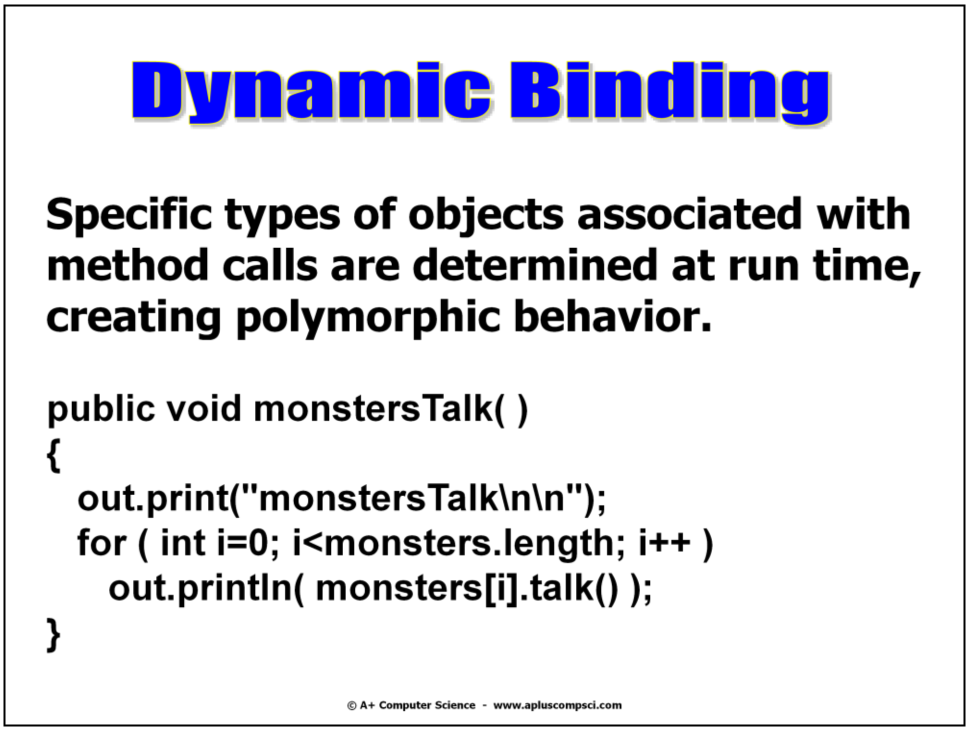
```
Actor a = new Bug(Color.GREEN);
a.move();                    //syntax error
                             //Actor has no move
((Bug)a).move();             //add a cast
```

Static binding is basically just matching up a class with its list of methods at compile time. Java makes sure that class x has method y.

Dynamic binding occurs dynamically at run-time. Java will call the appropriate method for a particular class on the fly. The same line of code will be used to call each different Objects method. The method names are the same, but the Objects will differ.



Open staticbinding.java



Dynamic Binding

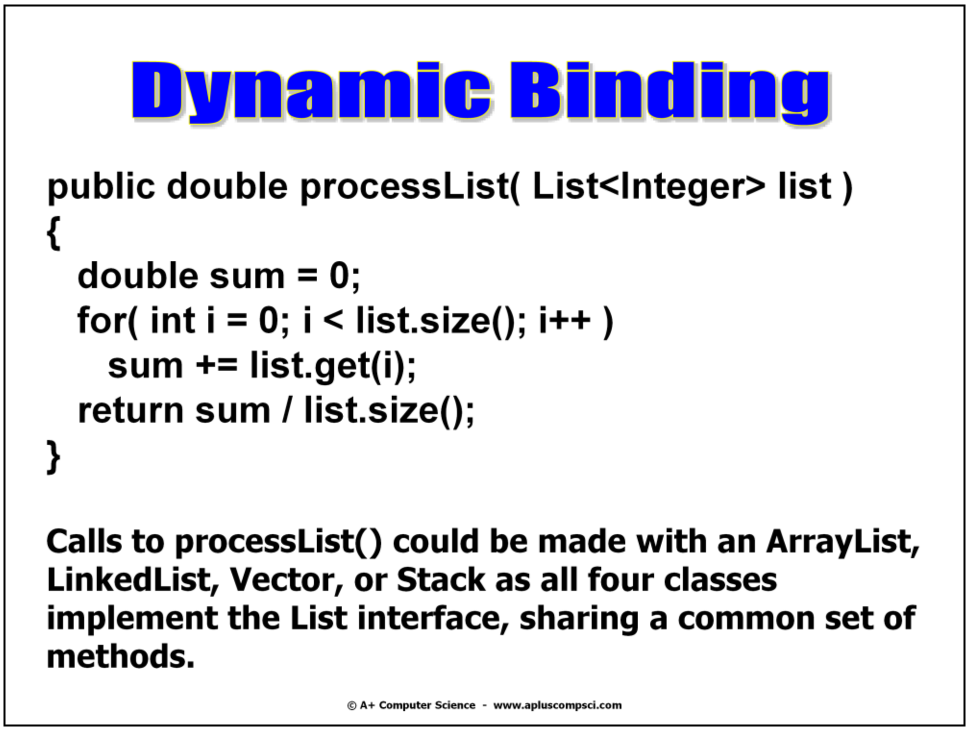
Specific types of objects associated with method calls are determined at run time, creating polymorphic behavior.

```
public void monstersTalk( )
{
  out.print("monstersTalk\n\n");
  for ( int i=0; i<monsters.length; i++ )
    out.println( monsters[i].talk() );
}
```

© A+ Computer Science - www.apluscompsci.com

Static binding is basically just matching up a class with its list of methods at compile time. Java makes sure that class x has method y.

Dynamic binding occurs dynamically at run-time. Java will call the appropriate method for a particular class on the fly. The same line of code will be used to call each different Objects method. The method names are the same, but the Objects will differ.



Dynamic Binding

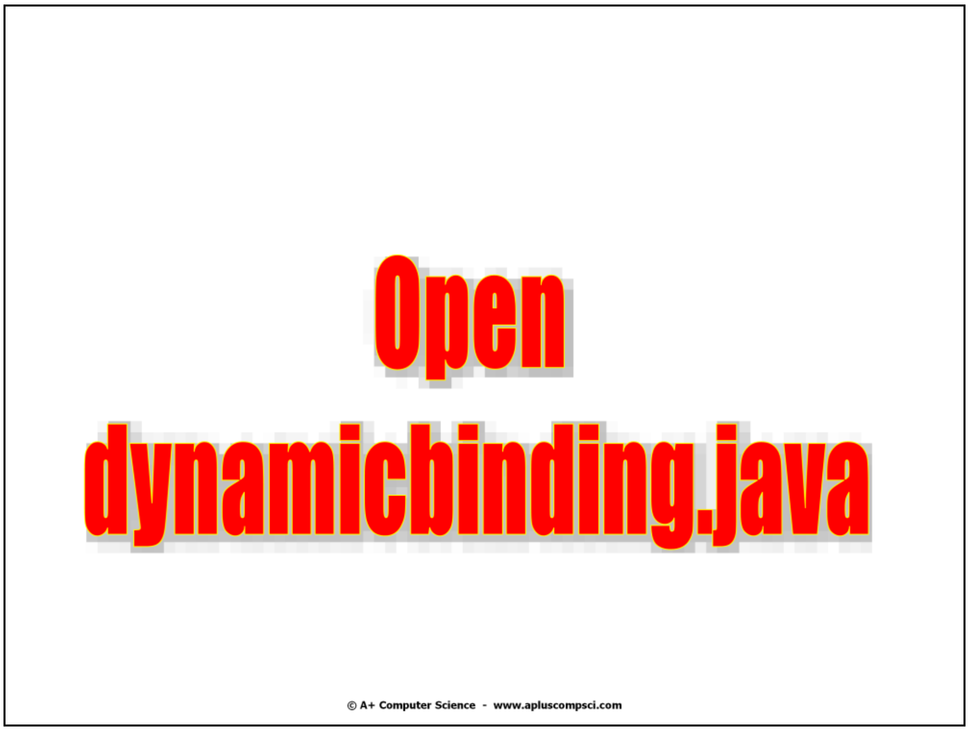
```
public double processList( List<Integer> list )
{
   double sum = 0;
   for( int i = 0; i < list.size(); i++ )
      sum += list.get(i);
   return sum / list.size();
}
```

Calls to processList() could be made with an ArrayList, LinkedList, Vector, or Stack as all four classes implement the List interface, sharing a common set of methods.

© A+ Computer Science - www.apluscompsci.com

Static binding is basically just matching up a class with its list of methods at compile time. Java makes sure that class x has method y.

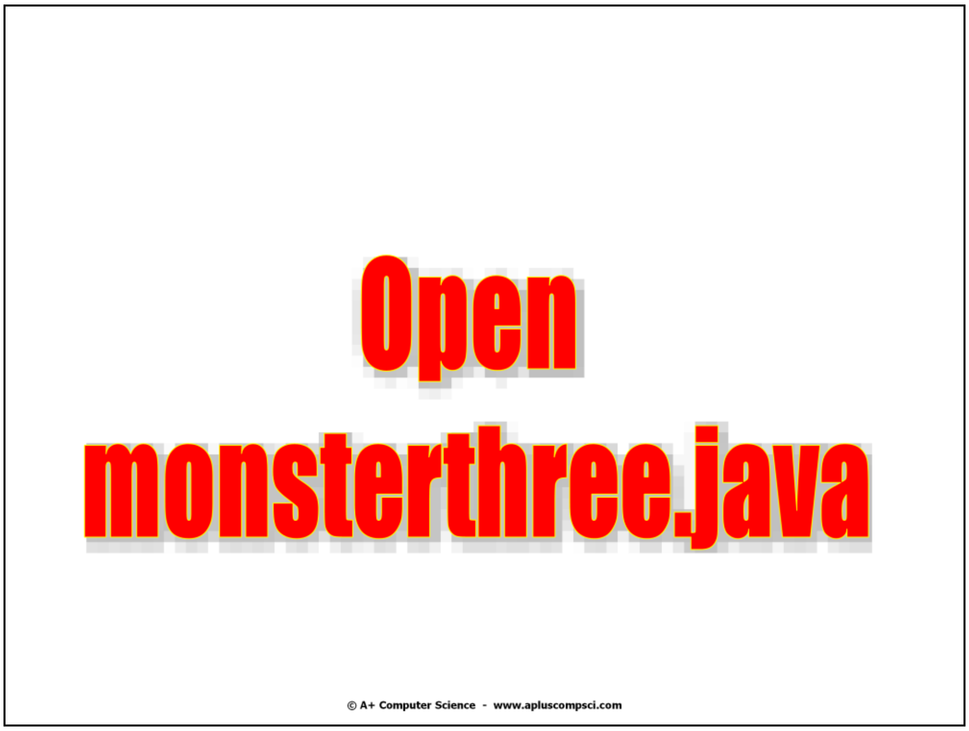
Dynamic binding occurs dynamically at run-time. Java will call the appropriate method for a particular class on the fly. The same line of code will be used to call each different Objects method. The method names are the same, but the Objects will differ.



Open dynamicbinding.java

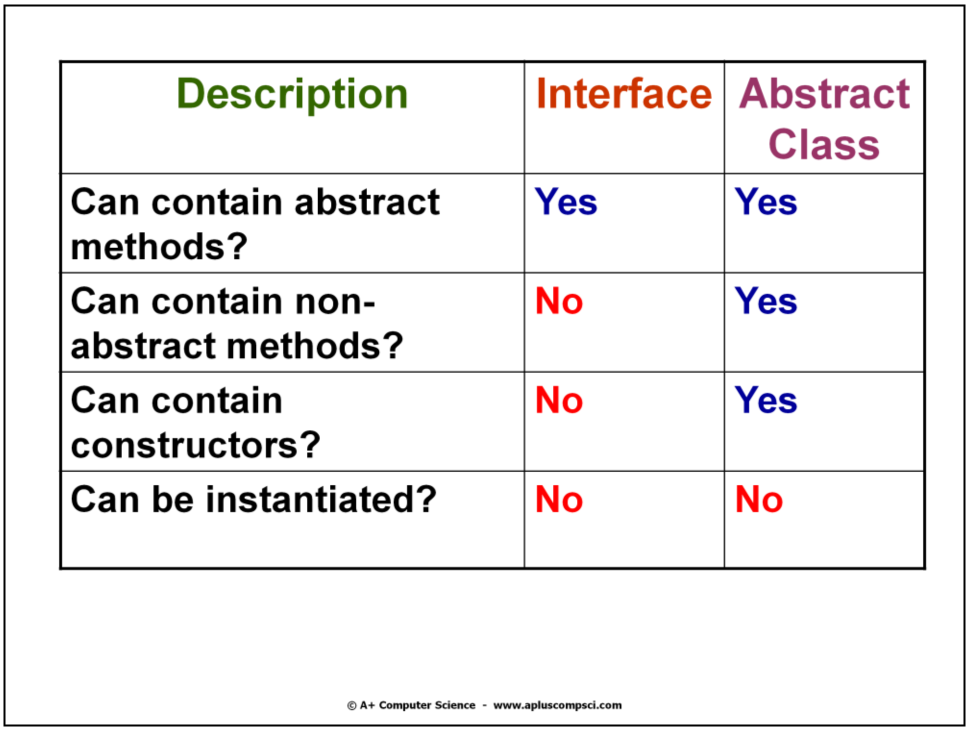
© A+ Computer Science - www.apluscompsci.com

Open monsterpack.java packrunner.java

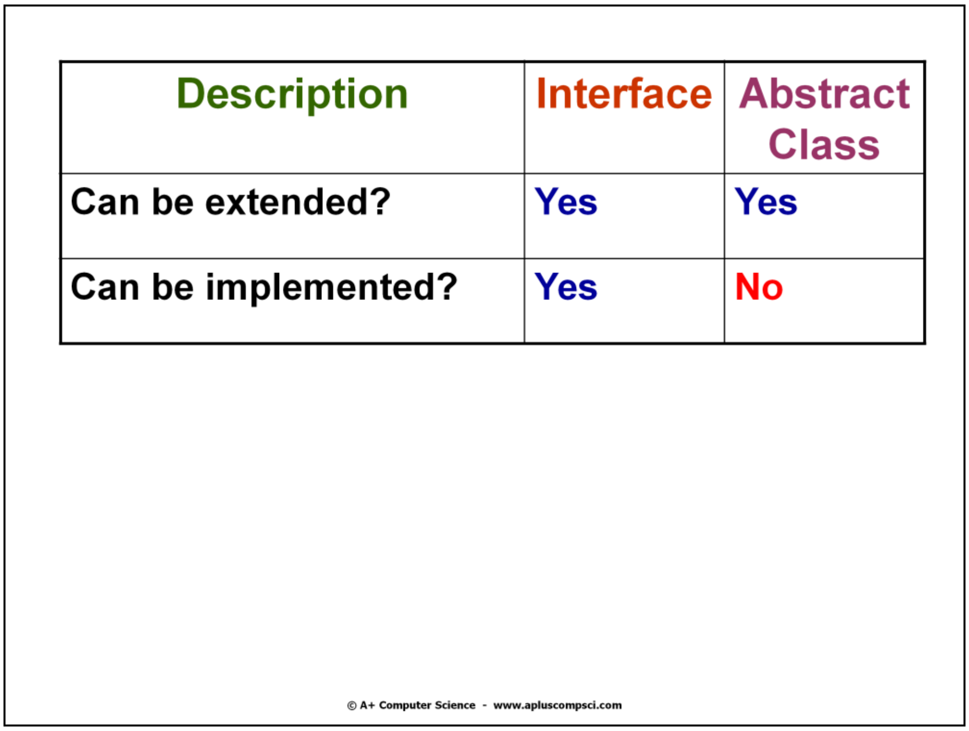


Open monsterthree.java

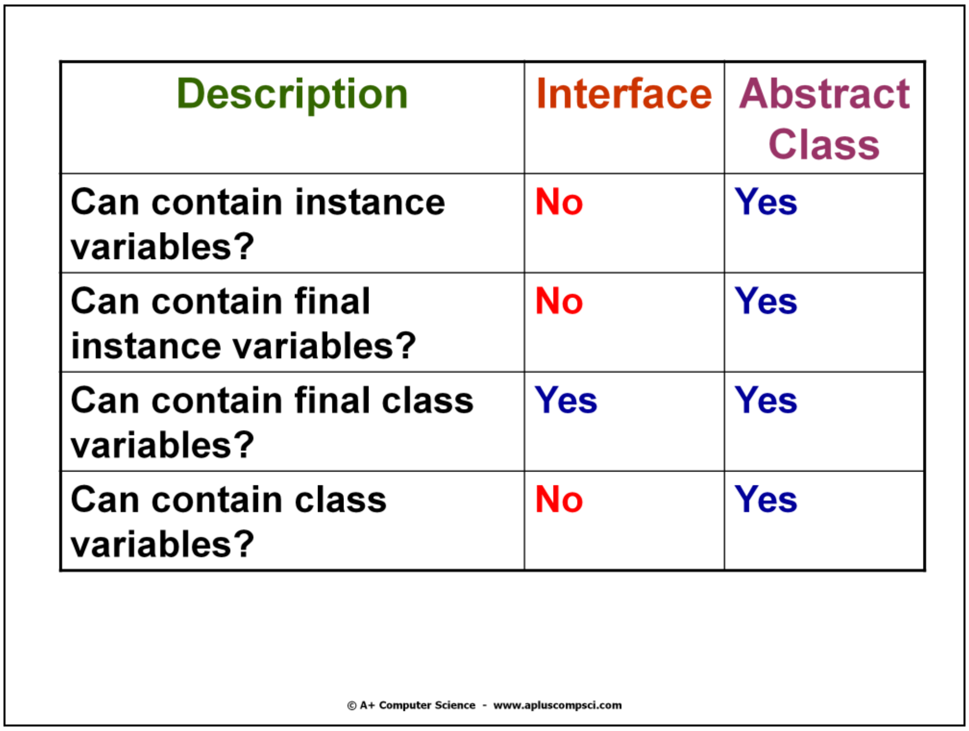
© A+ Computer Science - www.apluscompsci.com



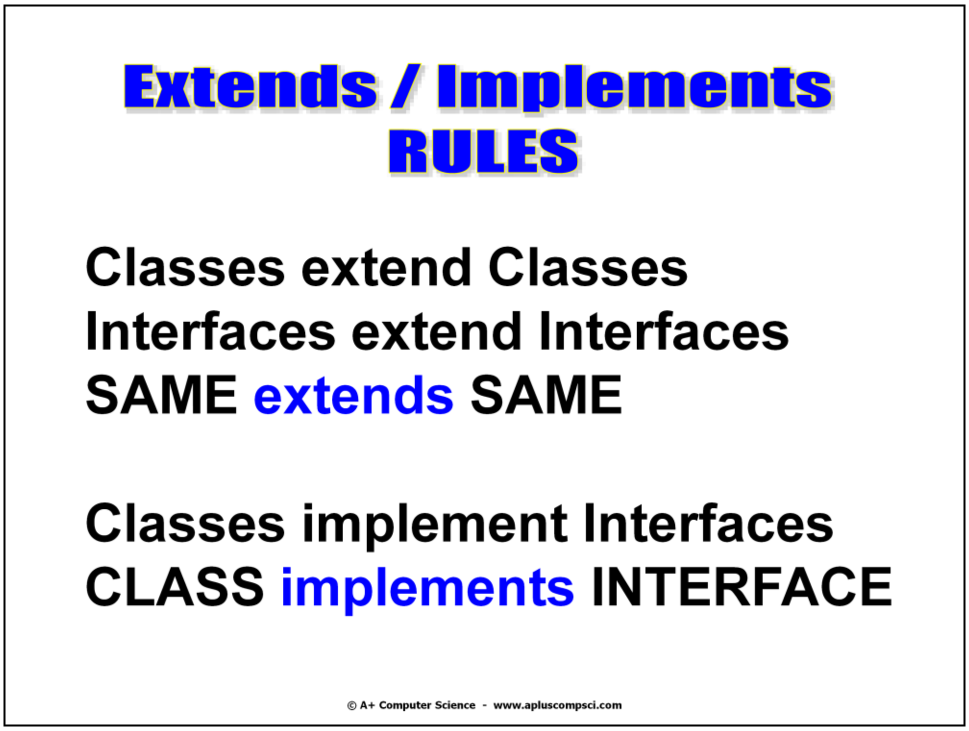
Description	Interface	Abstract Class
Can contain abstract methods?	Yes	Yes
Can contain non-abstract methods?	No	Yes
Can contain constructors?	No	Yes
Can be instantiated?	No	No



Description	Interface	Abstract Class
Can be extended?	Yes	Yes
Can be implemented?	Yes	No



Description	Interface	Abstract Class
Can contain instance variables?	No	Yes
Can contain final instance variables?	No	Yes
Can contain final class variables?	Yes	Yes
Can contain class variables?	No	Yes



Extends / Implements RULES

Classes extend Classes
Interfaces extend Interfaces
SAME extends SAME

Classes implement Interfaces
CLASS implements INTERFACE

© A+ Computer Science - www.apluscompsci.com

A class can extend another class.

An abstract class can extend another abstract class.

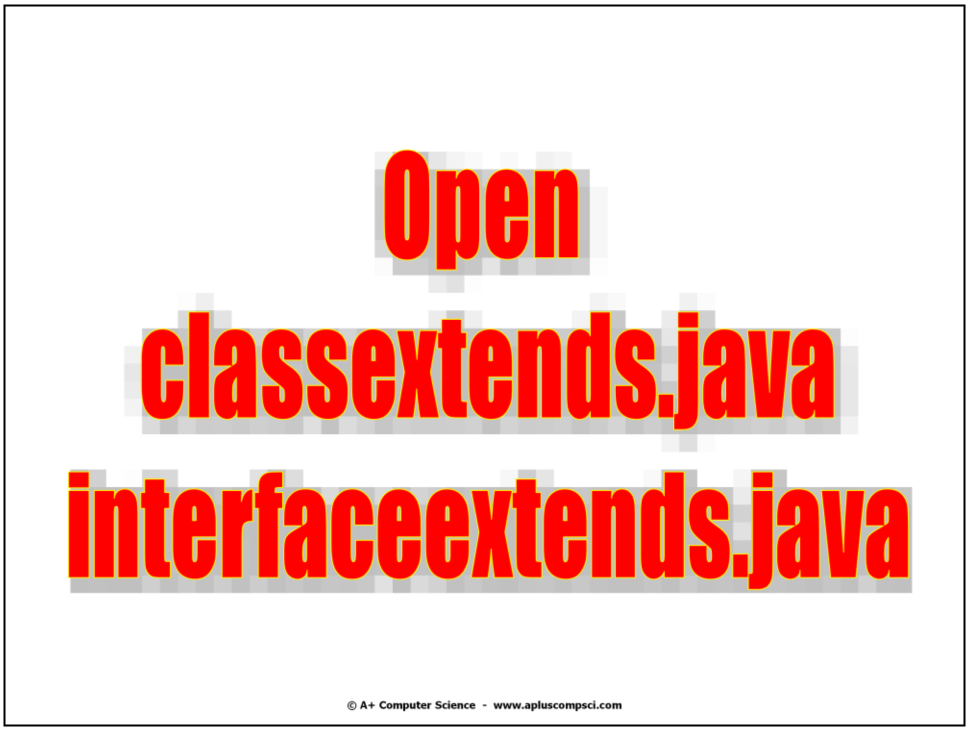
A class can extend an abstract class.

An abstract class can extend a class.

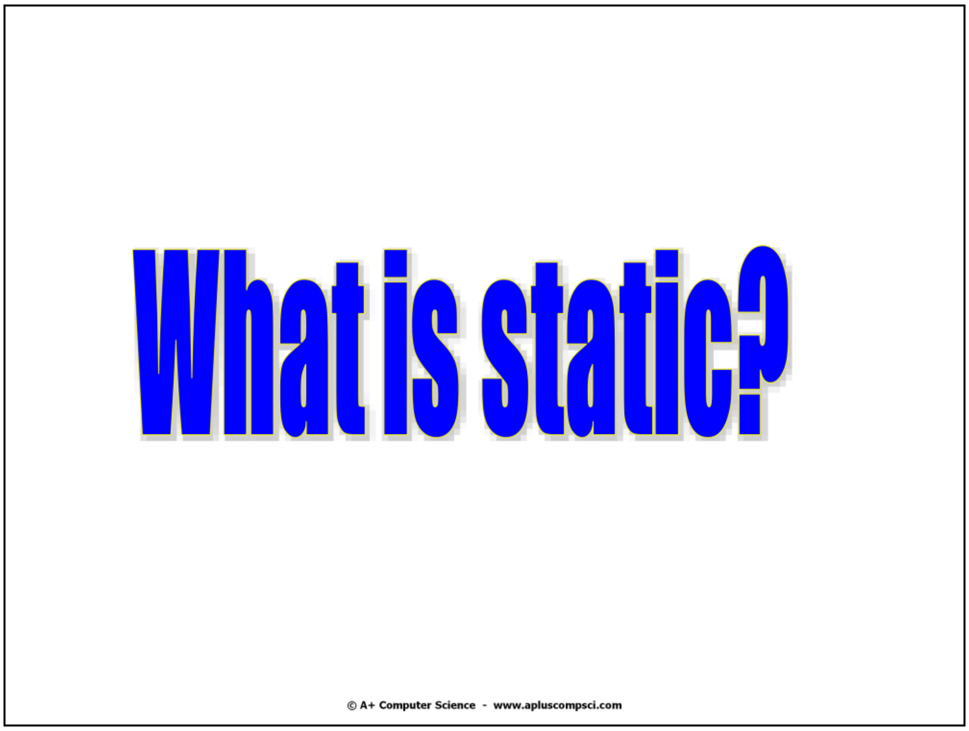
An interface can extend another interface.

A class can implement an interface.

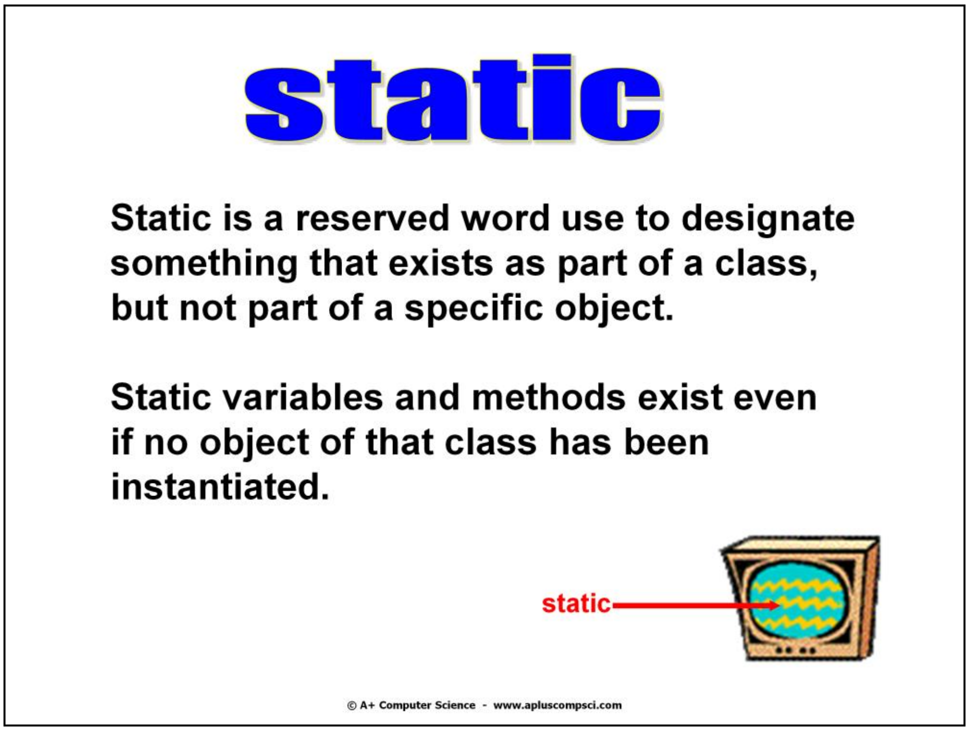
An abstract class can implement an interface.



Open classextends.java interfaceextends.java



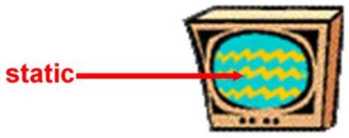
What is static?

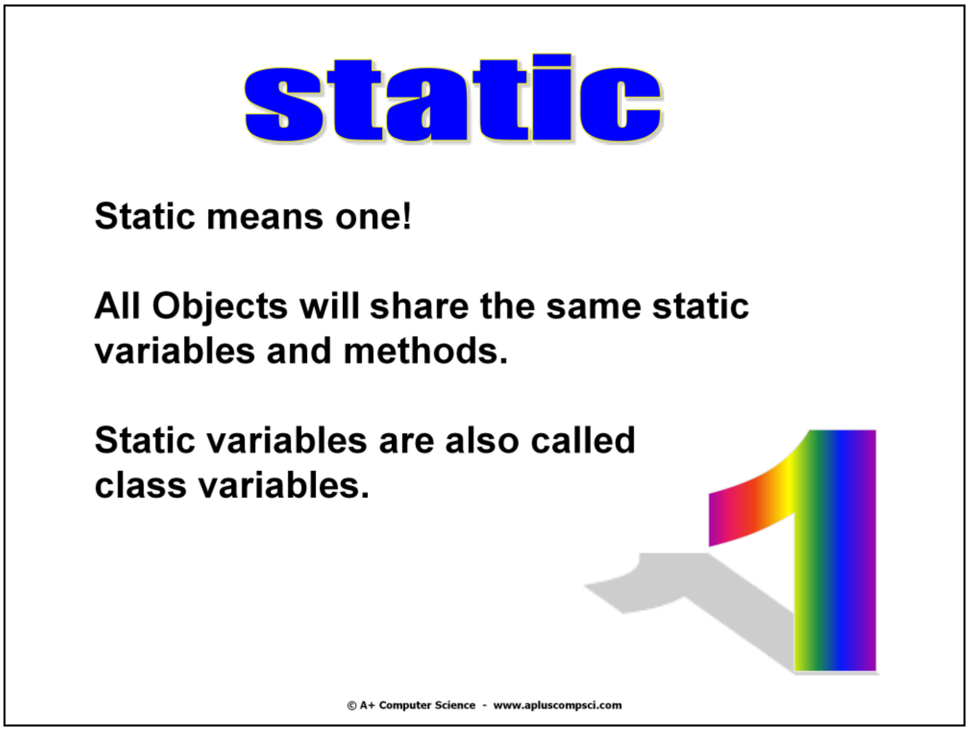


static

Static is a reserved word use to designate something that exists as part of a class, but not part of a specific object.

Static variables and methods exist even if no object of that class has been instantiated.



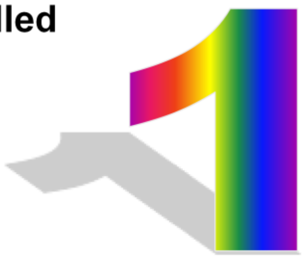


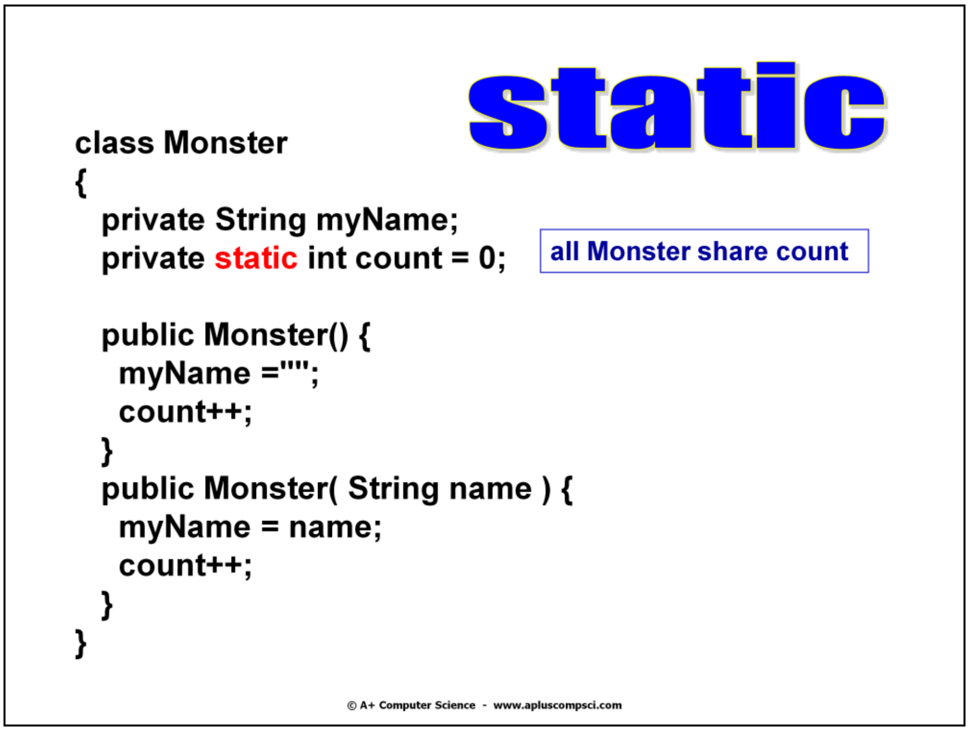
static

Static means one!

All Objects will share the same static variables and methods.

Static variables are also called class variables.

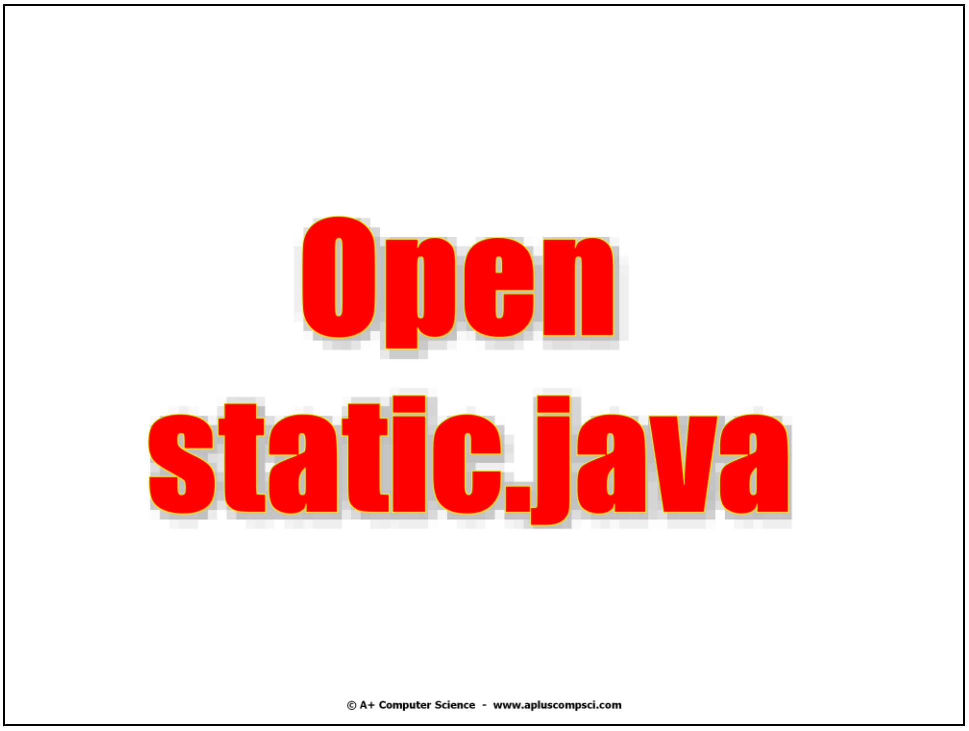




static

```
class Monster
{
  private String myName;
  private static int count = 0;

  public Monster() {
    myName ="";
    count++;
  }
  public Monster( String name ) {
    myName = name;
    count++;
  }
}
```

all Monster share count



Open static.java

Start work on the Labs