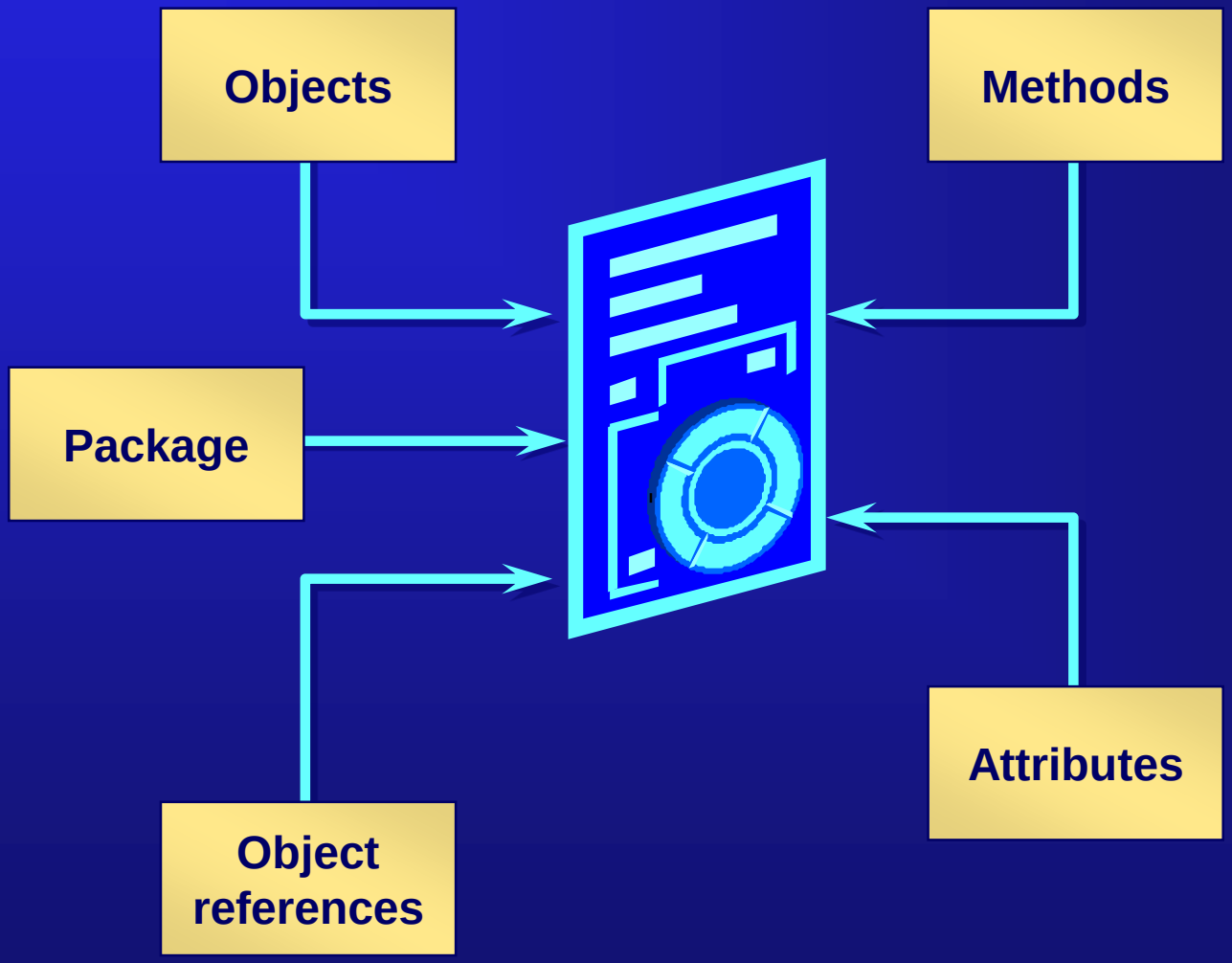
Object Oriented Programming in Java

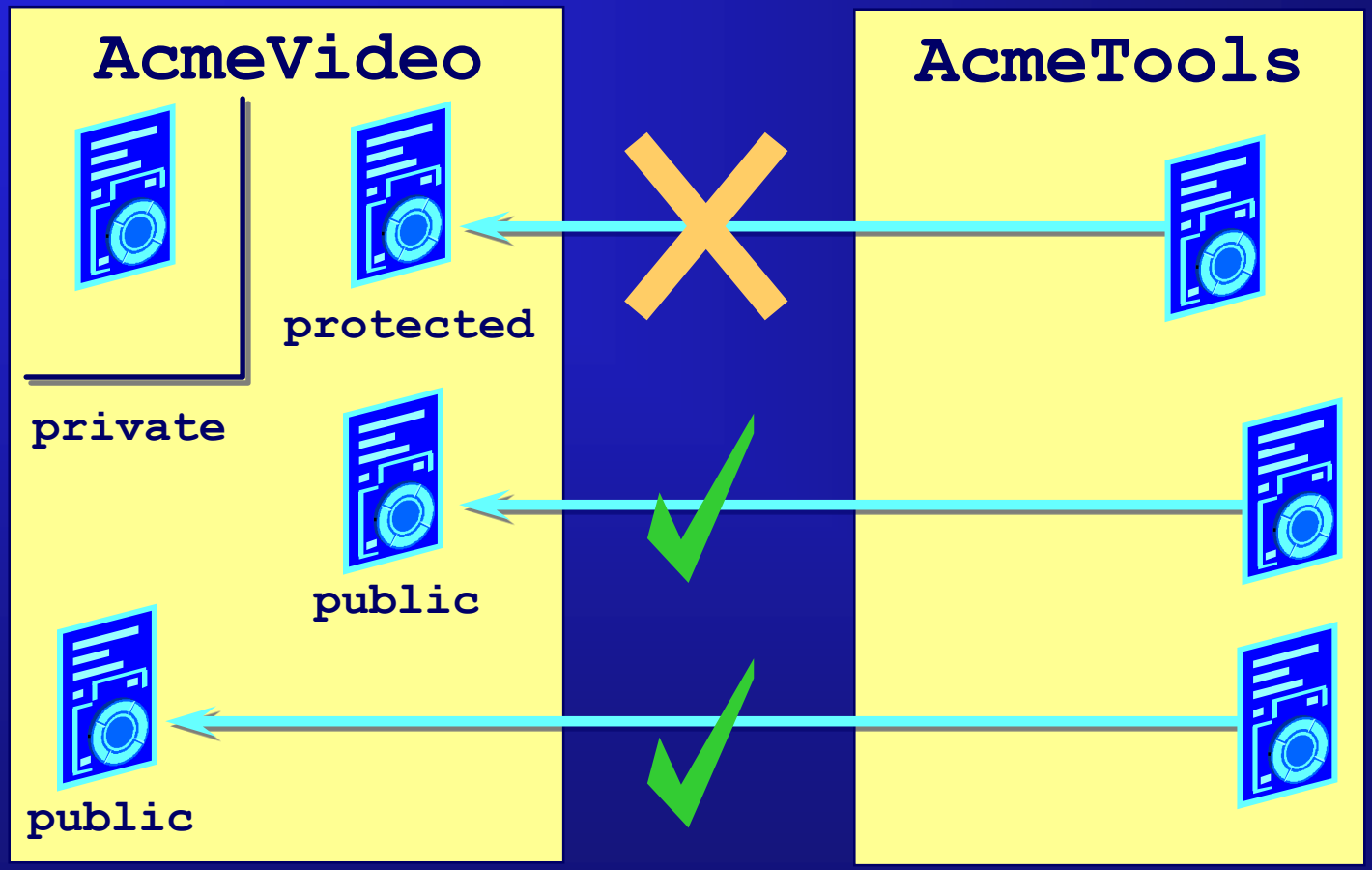
Overview

- Classes define the characteristics, attributes, and behaviors of objects.
- All Java code is held in classes.
- All object data is stored in variables.
- Packages group logically related classes by application and provide access control.
- Java uses packages to control which classes may be seen and accessed by classes outside of the package.

Java Classes

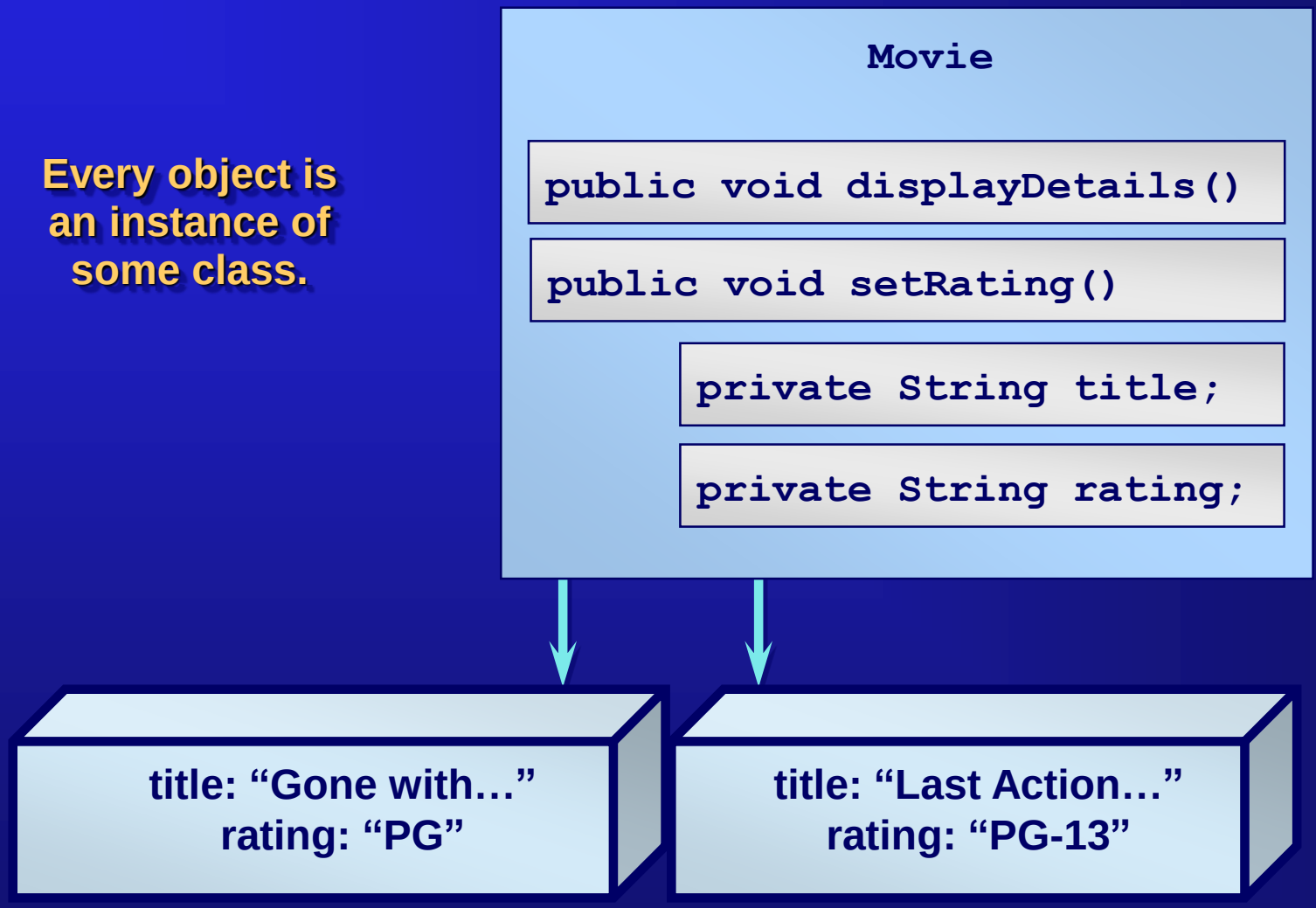


Access Modifiers



Classes and Objects

Every object is an instance of some class.



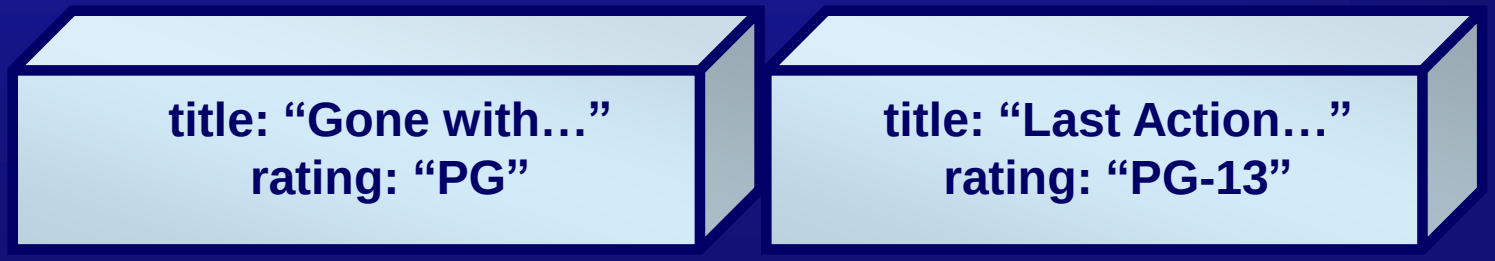
Creating Objects

- Objects are created by using the `new` operator:

```
objectRef = new ClassName();
```

- For example, to create two `Movie` objects:

```
Movie mov1 = new Movie("Gone ...");
Movie mov2 = new Movie("Last ...");
```



A Closer Look at `new`

The `new` operator performs the following actions:

- Allocates memory for the new object
- Calls a special initialization method in the class, called a *constructor*
- Returns a reference to the new object

```
Movie mov1 =
new Movie("Gone...");
```

Primitives and Objects

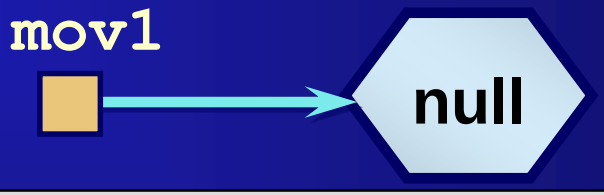
Primitive variables hold a value.

```
int i;
```


```
int j = 3;
```

Object variables hold references.

```
Movie mov1;
```




```
Movie mov1 = new Movie();
```

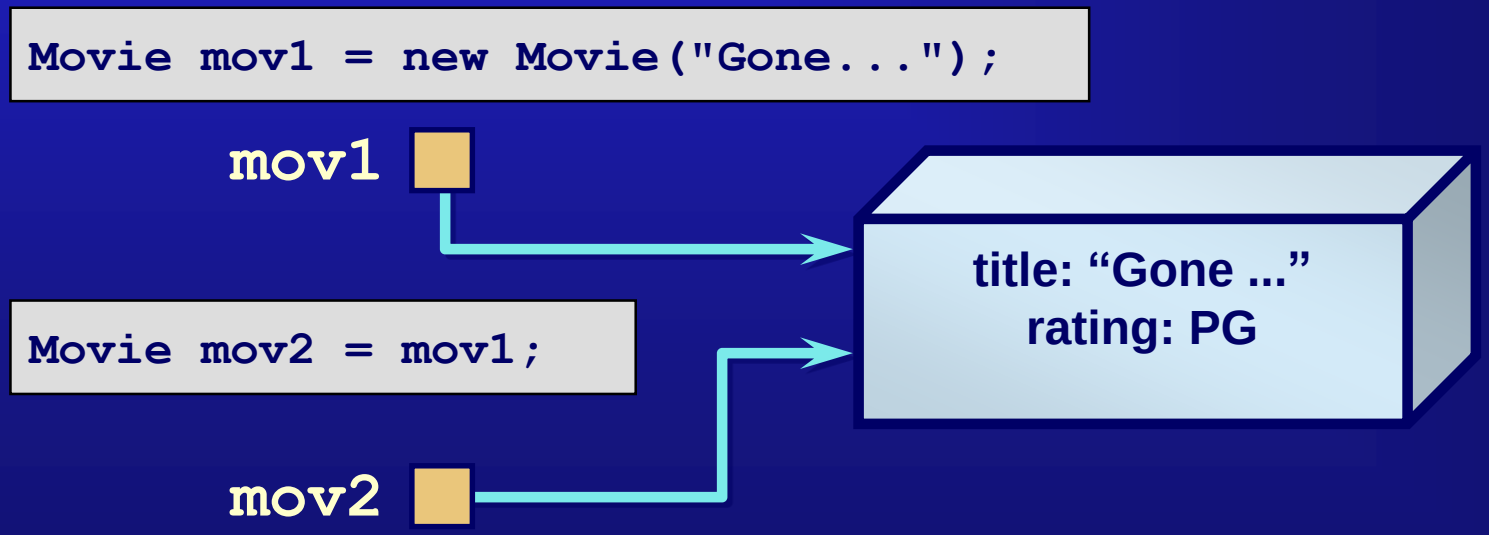
The `null` Reference

- Object references are `null` until initialized.
- You can compare object references to `null`.
- You can "forget" an object by setting the object reference to `null`.

```
Movie mov1 = null;  //Declare object reference
…
if (mov1 == null)     //Ref not initialized?
 mov1 = new Movie(); //Create a Movie object
…
mov1 = null;          //Forget the Movie object
```

Assigning References

Assigning one reference to another results in two references to the same object:



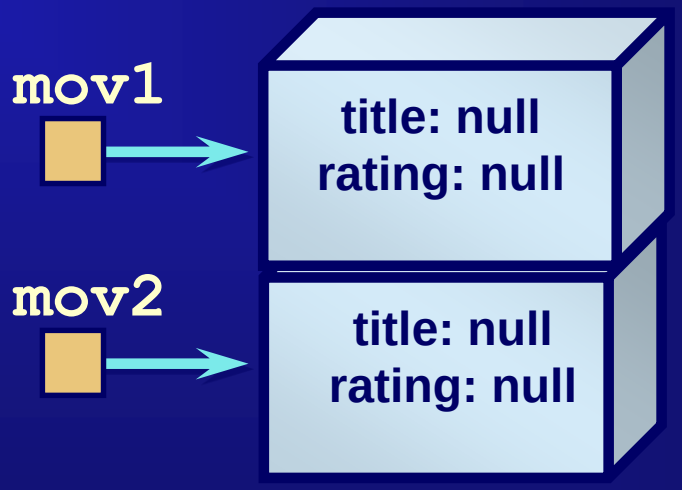
Instance Variables

Instance variables are declared in the class:

```
public class Movie {
  public String title;
  public String rating;
...
}
```

Create `movies`:

```
Movie mov1 = new Movie();
Movie mov2 = new Movie();
```



Accessing Instance Variables

Public instance variables can be accessed by using the *dot* operator:

```
public class Movie {
   public String title;
   public String rating;
   …
}
```

```
Movie mov1 = new Movie();
mov1.title = "Gone ...";
…
if ( mov1.title.equals("Gone ... ") )
     mov1.rating = "PG";
```

Do you really want to do this?

Methods

- A method is equivalent to a function or subroutine in other languages:

```
modifier returnType methodName (argumentList) {
    // method body
    ...
}
```

- A method can only be defined within a class definition.

Specifying Method Arguments: Examples

- Specify the number and type of arguments in the method definition:

```
public void setRating(String newRating) {
    rating = newRating;
  }
```

- If the method takes no arguments, leave the parentheses empty:

```
public void displayDetails() {
  System.out.println("Title is  " + title);
  System.out.println("Rating is " + rating);
}
```

Returning a Value from a Method

- Use a return statement to exit a method and to return a value from a method:

```
public class Movie {
  private String rating;
  …
  public String getRating () {
    return rating;
  }
}
```

- No return needed if the method returns `void`.

Calling Instance Methods

```
public class Movie {
  private String title, rating;
  public  String getRating(){
    return rating;
  }
  public void setRating(String newRating){
    rating = newRating;
  }
}
```

Use the dot operator:

```
Movie mov1 = new Movie();
...
if (mov1.getRating().equals("G"))
  ...
```

Encapsulation

- Instance variables should be declared as `private`.
- Only instance methods can access `private` instance variables.
- `private` decouples the interface of the class from its internal operation.

```
Movie mov1 = new Movie();
...
if ( !mov1.rating.equals("PG") )    // Error
     mov1.setRating("PG");          // OK
```

Passing Primitives to Methods

When a primitive value is passed to a method, a copy of the value is generated:

```
int num = 150;

anObj.aMethod(num);
System.out.println("num: " + num);
```

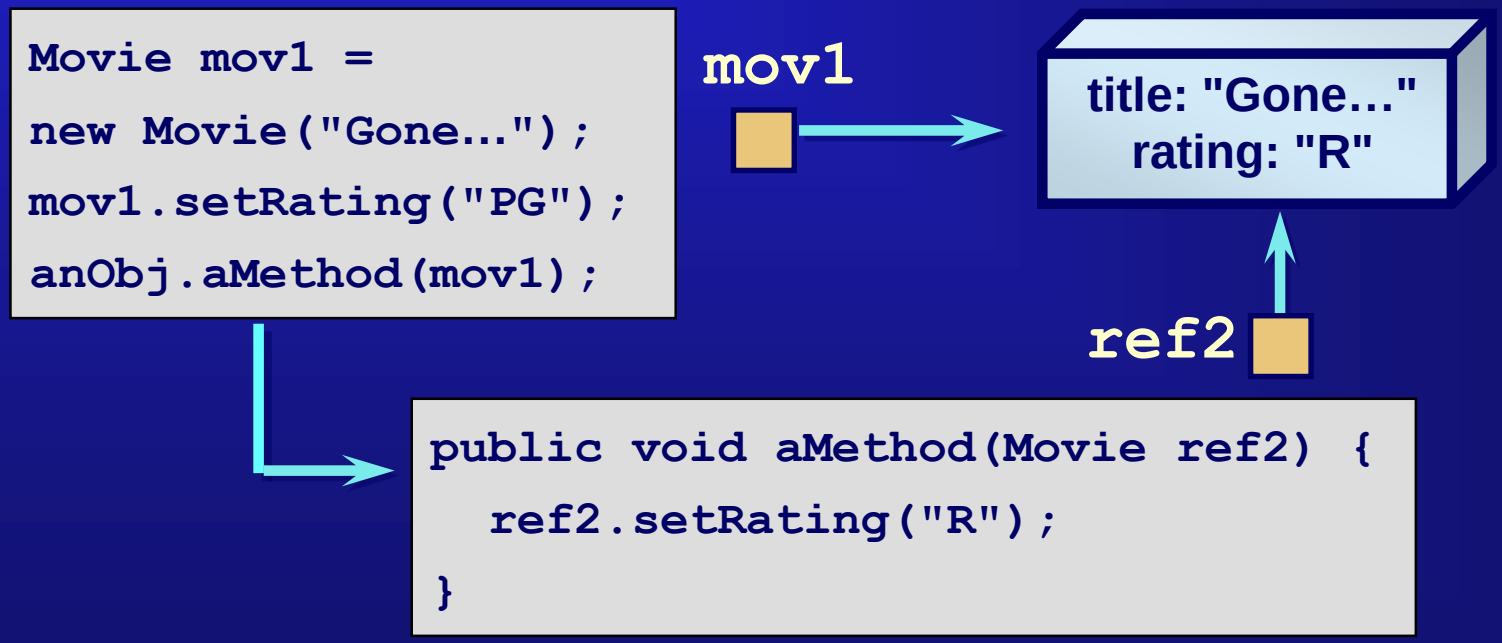
num
150

```
public void aMethod(int arg) {
  if (arg < 0 || arg > 100)
    arg = 0;
  System.out.println("arg: " + arg);
}
```

arg
150

Passing Object References to Methods

When an object reference is passed to a method, the argument refers to the original object:



More on Working with Classes

Overview

- Instance methods are the foundation of encapsulation.
- Overloading provides for a clean interface.
- Constructors ensure consistent object creation.

Method Overloading

- Several methods in a class can have the same name.
- The methods must have different signatures.

```
public class Movie {
  public void setPrice() {
    price = 3.50;
  }
  public void setPrice(float newPrice) {
    price = newPrice;
  } ...
}
```

```
Movie mov1 = new Movie();
mov1.setPrice();
mov1.setPrice(3.25);
```

Simple Initialization of Instance Variables

- Instance variables can be initialized at declaration.
- Initialization happens at object creation.

```
public class Movie {
  private String title;
  private String rating = "G";
  private int numOfOscars = 0;
```

- More complex initialization should be placed in a constructor.

Constructors

- For proper initialization, a class should provide a constructor.
- A constructor is called automatically when an object is created:
  - Usually declared public
  - Has the same name as the class
  - No specified return type
- The compiler supplies an do-nothing no-arg constructor by default.

Defining Constructors

```
public class Movie {
  private String title;
  private String rating = "PG";

  public Movie() {
    title = "Last Action ...";
  }
  public Movie(String newTitle) {
    title = newTitle;
  }
}
```

The `Movie` class now provides two constructors.

```
Movie mov1 = new Movie();
Movie mov2 = new Movie("Gone ...");
Movie mov3 = new Movie("The Good ...");
```

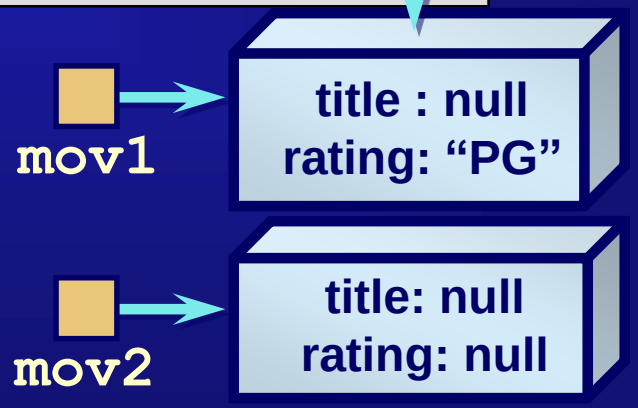
The `this` Reference

Instance methods receive an argument called `this`, which refers to the current object.

```
public class Movie {
  public void setRating(String newRating) {
    this.rating = newRating;
  }
```

this

```
void anyMethod() {
 Movie mov1 = new Movie();
 Movie mov2 = new Movie();
 mov1.setRating("PG"); ...
```



Sharing Code Between Constructors

```
public class Movie {
  private String title;
  private String rating;

  public Movie() {
    this("G");
  }
  public Movie(String newRating) {
    rating = newRating;
  }
}
```

A constructor can call another constructor by using `this()`.

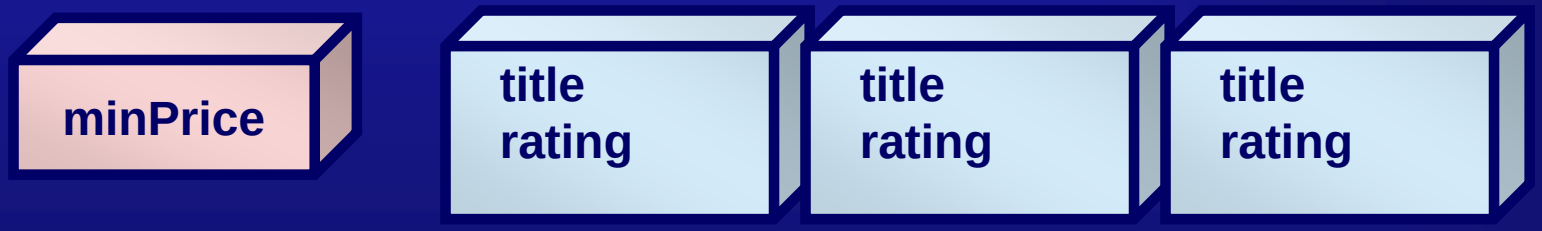
What happens here?

```
Movie mov2 = new Movie();
```

Class Variables

- Class variables belong to a class and are common to all instances of that class.
- Class variables are declared as static in class definitions.

```
public class Movie {
  private static double minPrice;   // class var
  private String title, rating;     // inst vars
```



`Movie` class

`Movie` objects

Initializing Class Variables

- Class variables can be initialized at declaration.
- Initialization takes place when the class is loaded.

```
public class Movie {
  private static double minPrice = 1.29;

  private String title, rating;
  private int length = 0;
```

Class Methods

- Class methods are shared by all instances.
- Useful for manipulating class variables:

```
public static void increaseMinPrice(double inc) {
  minPrice += inc;
}
```

- Call a class method by using the class name or an object reference.

```
Movie.increaseMinPrice(.50);
mov1.increaseMinPrice(.50);
```

Examples in Java

Examples of static methods and variables:

- `main()`
- `Math.sqrt()`
- `System.out.println()`

```
public class MyClass {

  public static void main(String[] args) {
    double num, root;
    …
    root = Math.sqrt(num);
    System.out.println("Root is " + root);
  } …
```

`final` Variables

- A `final` variable is a constant.
- A `final` variable cannot be modified.
- A `final` variable must be initialized.
- A `final` variable is often public to allow external access.

```
public final class Color {

  public final static Color black=new Color(0,0,0);
  ...
}
```

Garbage Collection

- When all references to an object are lost, it is marked for garbage collection.

  Garbage collection reclaims memory used by the object.

- Garbage collection is automatic.

  There is no need for the programmer to do anything, but he or she has no control over when the garbage is collected.

The `finalize()` Method

- If an object holds another resource such as a file, the object should reclaim it.
- You can provide a `finalize()` method in that class.
- The `finalize()` method is called just before garbage collection.

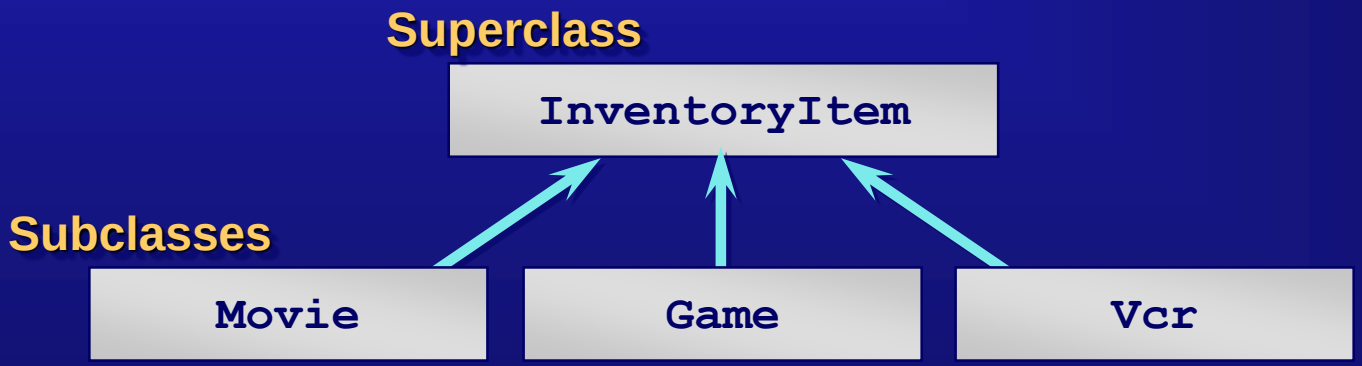
```
public class Movie {
  ...
  public void finalize() {
    System.out.println("Goodbye");
  }
}
```

Any problems?

Reusing Code Through Inheritance

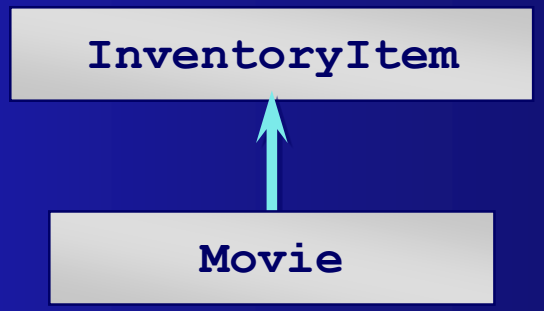
Overview

- Inheritance
- Inheritance and constructors
- Polymorphism
- Inheritance is an OO fundamental



Example of Inheritance

- The `InventoryItem` class defines methods and variables.



- `Movie` extends `InventoryItem`, and can:
  - Add new variables
  - Add new methods
  - Override methods of the `InventoryItem` class

Specifying Inheritance in Java

- Inheritance is achieved by specifying which superclass the subclass extends.

```
public class InventoryItem {
   ...
}
```

```
public class Movie extends InventoryItem {
   ...
}
```

- `Movie` inherits all the variables and methods of `InventoryItem`.

What Does a Subclass Object Look Like?

A subclass inherits all the instance variables of its superclass.

```
public class InventoryItem {
  private float price;
  private String condition; ...
}
```

```
public class Movie extends
InventoryItem{
  private String title;
  private int length; ...
}
```

InventoryItem

Movie

Default Initialization

- What happens when a subclass object is created?

```
Movie movie1 = new Movie();
```

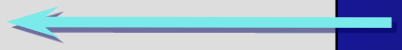
- If no constructors are defined:
  - The default no-arg constructor is called in superclass.
  - The default no-arg constructor is called in subclass.

The `super` Reference

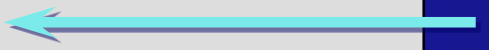
- `super` refers to the base class
- Useful for calling base class constructors
- Must be the first line in the derived class constructor
- May be used to call any base class methods

The super Reference Example

```
public class Title {
  Title(String title) {
    System.out.println("Title");
    ...
  }
}
class Movie extends Title {
  Movie(String title) {
    super(title);
    ...
    System.out.println("Movie");
  }
}
```



Base class constructor



Calls base class constructor

Specifying Additional Methods

- The superclass defines methods for all types of `InventoryItem`.
- The subclass can specify additional methods that are specific to `Movie`.

```
public class InventoryItem {
  public float calcDeposit()...
  public String calcDateDue()...
  ...
```

```
public class Movie extends InventoryItem {
  public void getTitle()...
  public String getLength()...
```

Overriding Superclass Methods

- A subclass inherits all the methods of its superclass.
- The subclass can override a method with its own specialized version.

```
public class InventoryItem {
  public float calcDeposit(int custId) {
  if ...
    retur
  }
```

```
public class Vcr extends InventoryItem {
  public float calcDeposit(int custId) {
   if ...
     return itemDeposit;
  }
```

Invoking Superclass Methods

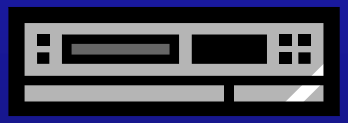
- If a subclass overrides a method, it can still call the original superclass method.
- Use `super.method()` to call a superclass method from the subclass.

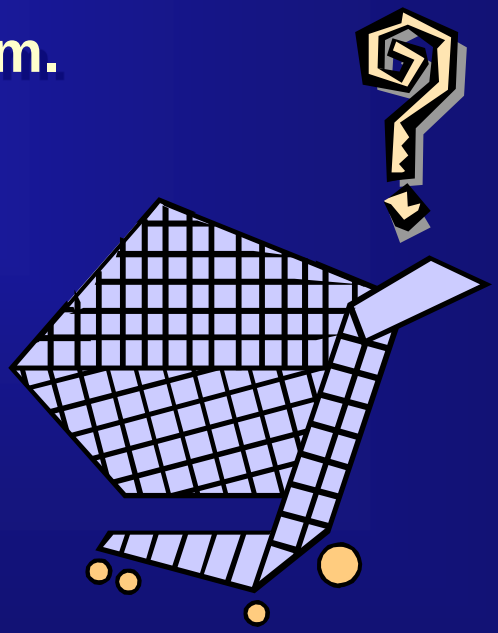
```
public class InventoryItem {
  public float calcDeposit(int custId) {
    if
    re
  }
```

```
public class Vcr extends InventoryItem {
 public float calcDeposit(int custId) {
    itemDeposit = super.calcDeposit(custId);
    return (itemDeposit + vcrDeposit);
}
```

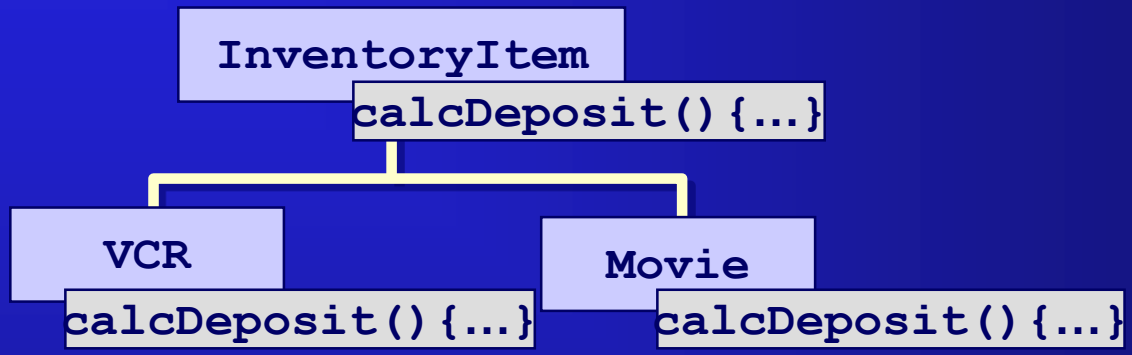
Acme Video and Polymorphism

- Acme Video started renting only videos.
- Acme Video added games and VCRs.
- What's next?
- Polymorphism solves the problem.





How It Works



ShoppingBasket

```
void addItem(InventoryItem item) {
    // this method is called each time
    // the clerk scans in a new item
    float deposit = item.calcDeposit();
    ...
}
```

The `instanceof` Operator

- The true type of an object can be determined by using an `instanceof` operator.
- An object reference can be downcast to the correct type, if needed.

```
public void aMethod(InventoryItem i) {
  ...
  if (i instanceof Vcr)
    ((Vcr)i).playTestTape();
}
```

`final` Methods and Classes

- A method can be marked as `final` to prevent it from being overridden.

```
public final boolean checkPassword(String p) {
   ...
}
```

- A whole class can be marked as `final` to prevent it from being extended.

```
public final class Color {
   ...
}
```

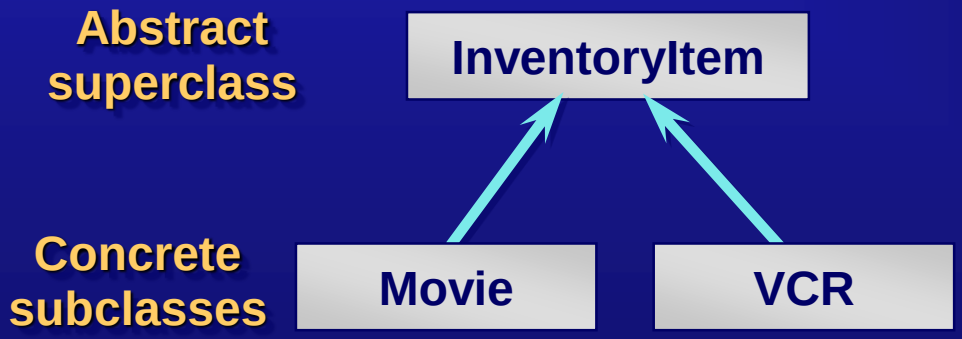
Design Issues

- Inheritance should be used only for genuine “is a kind of” relationships:
  - It must always be possible to substitute a subclass object for a superclass object.
  - All methods in the superclass should make sense in the subclass.
- Inheritance for short-term convenience leads to problems in the future.

Structuring Code Using Abstract Classes and Interfaces

Overview

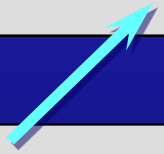
- An abstract class cannot be instantiated.
- Abstract methods must be implemented by subclasses.
- Interfaces support multiple inheritance.

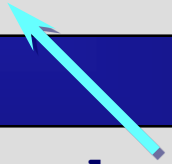


Defining Abstract Classes in Java

Use the `abstract` keyword to declare a class as abstract.

```
public abstract class InventoryItem {
   private float price;
   public boolean isRentable()...
}
```





```
public class Movie
extends InventoryItem {
 private String title;
 public int getLength()...
```

```
public class Vcr
extends InventoryItem {
 private int serialNbr;
 public void setTimer()...
```

Abstract Method

- An abstract method is an implementation placeholder.
- They are part of an abstract class.
- They must be overridden by a concrete subclass.
- Each concrete subclass can implement the method differently.

Defining Abstract Methods in Java

- Use the `abstract` keyword to declare a method as abstract.
  - Provide the method signature only.
  - Class must also be abstract.
- Why is this useful?

```
public abstract class InventoryItem {
  public abstract boolean isRentable();
  ...
```

Example of Interfaces

- Interfaces describe an aspect of behavior that different classes require.
- For example, classes that can be steered support the “steerable” interface.
- Classes can be unrelated.

Non-steerable

Steerable

Defining an Interface

- Use `interface` keyword

```
public interface Steerable {
  int MAXTURN = 45;
  void turnLeft(int deg);
  void turnRight(int deg);
}
```

- All methods `public abstract`
- All variables `public static final`

Implementing an Interface

Use `implements` keyword

```
public class Yacht extends Boat
      implements Steerable
  public void turnLeft(int deg) {...}
  public void turnRight(int deg) {...}
}
```

Sort: A Real-World Example

- Used by a number of unrelated classes
- Contains a known set of methods
- Need it to sort any type of object
- Comparison rules known only to the sortable object
- Good code reuse

Overview of the Classes

- Created by the sort expert

```
public interface
   Sortable
```

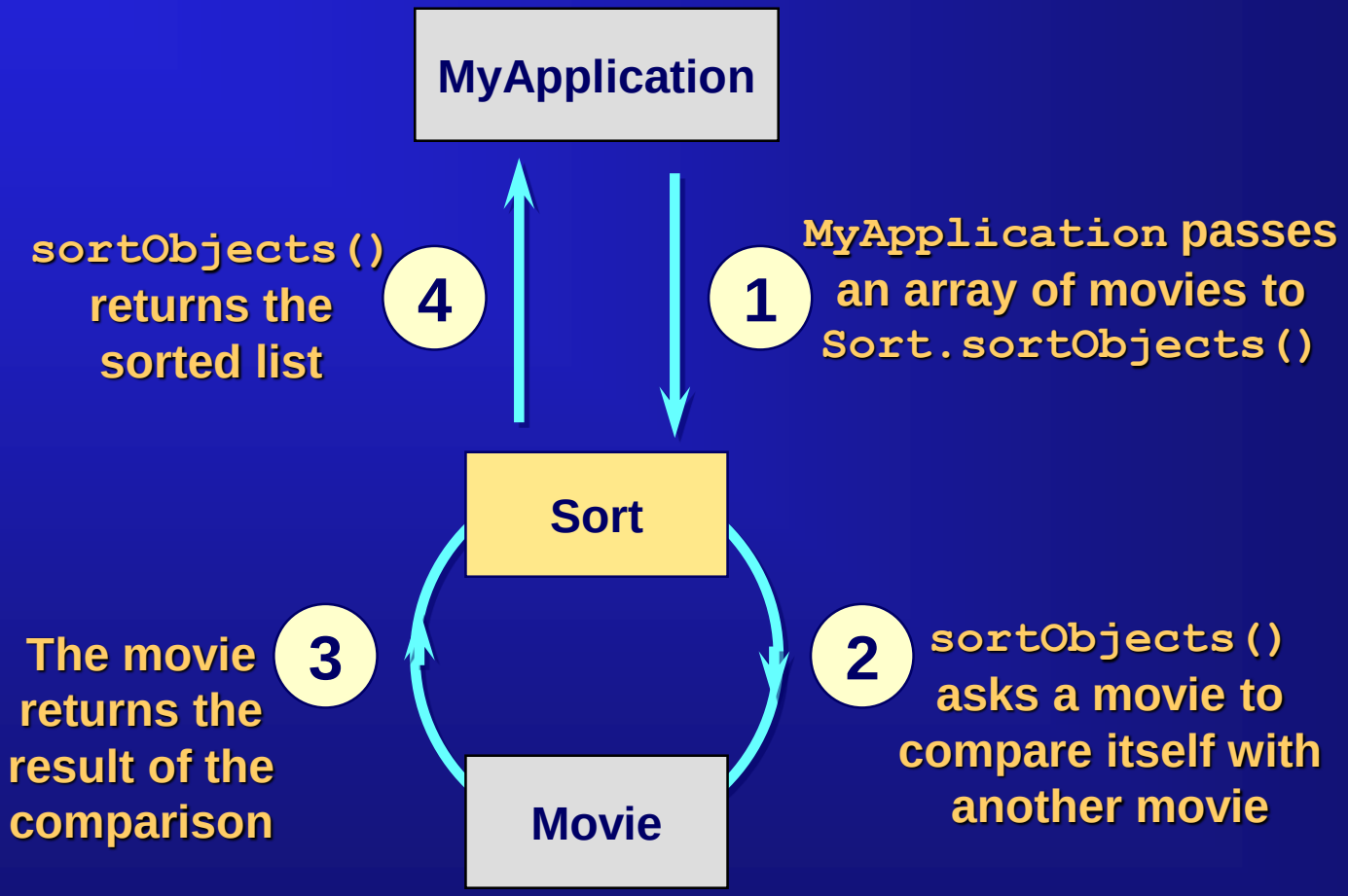
```
public abstract
   class Sort
```

- Created by the movie expert

```
public class Movie
   implements Sortable
```

```
public class
   MyApplication
```

How the Sort Works



The `Sortable` Interface

Specifies the `compare()` method

```
public interface Sortable {
  // compare(): Compare this object to another object
  // Returns:
  //    0 if this object is equal to obj2
  //    a value < 0 if this object < obj2
  //    a value > 0 if this object > obj2
  int compare(Object obj2);
}
```

The `Sort` Class

Holds `sortObjects()`

```
public abstract class Sort {
  public static void sortObjects(Sortable[] items) {
    // Step through the array comparing and swapping;
    // do this length-1 times
    for (int i = 1; i < items.length; i++) {
      for (int j = 0; j < items.length - 1; j++) {
        if (items[j].compare(items[j+1]) > 0) {
          Sortable tempitem = items[j+1];
          items[j+1] = items[j];
          items[j] = tempitem; } } } } }
```

The Movie Class

Implements Sortable

```
public class Movie extends InventoryItem
                    implements Sortable {
  String title;
  public int compare(Object movie2) {
    String title1 = this.title;
    String title2 = ((Movie)movie2).getTitle();
    return(title1.compareTo(title2));
  }
}
```

Using the Sort

Call `Sort.sortObjects(Sortable [])` with an array of `Movie` as the argument

```
class myApplication {
  Movie[] movielist;
  ...      // build the array of Movie
  Sort.sortObjects(movielist);
}
```

Using `instanceof` with Interfaces

- Use the `instanceof` operator to check if an object implements an interface.
- Use downcasting to call methods defined in the interface.

```
public void aMethod(Object obj) {
  ...
  if (obj instanceof Sortable)
    ((Sortable)obj).compare(obj2);
}
```

Generics in Java

Generic Methods and Generic Classes

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Type parameters provide a way for you to re-use the same code with different inputs.

- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

```
public static < E > void printArray( E[] inputArray ) {
    for ( E element : inputArray ){
        System.out.printf( "%s ", element );
    }
    System.out.println();
}
```

Generic Methods

```
public static void main( String args[] ) {
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println( "Array integerArray contains:" );
    printArray( intArray ); // pass an Integer array

    System.out.println( "\nArray doubleArray contains:" );
    printArray( doubleArray ); // pass a Double array

    System.out.println( "\nArray characterArray contains:" );
    printArray( charArray ); // pass a Character array
}
```

Generic Classes

- The class **ArrayList** is designed to take a generics type **<E>** as follows:

```
public class ArrayList<E> implements List<E> .... {
    public ArraList() { ..... }
    public boolean add(E e) { ..... }
    public void add(int index, E element) { ..... }
    public boolean addAll(int index, Collection<? extends E> c)
    public abstract E get(int index) { ..... }
    public E remove(int index) { ..... }
    .....
}
```

Generic Classes

- **<E>** is called the *formal "type" parameter*, which can be used for passing "type" parameters during the actual instantiation.
- The actual type provided will then substitute all references to **E** inside the class. For example,

```
ArrayList<Integer> lst1 = new ArrayList<Integer>(); // E substituted with Integer
lst1.add(0, new Integer(88));
lst1.get(0);

ArrayList<String> lst2 = new ArrayList<String>(); // E substituted with String
lst2.add(0, "Hello");
lst2.get(0);
```

Generic Interfaces

- The following is the design of the interface **java.util.List<E>**:

```
public interface List<E> extends Collection<E> {
      boolean add(E o);
      void add(int index, E element);
      boolean addAll(Collection<? extends E> c);
      boolean containsAll(Collection<?> c);
      .....
 }
```

Formal Type Parameter Naming Convention

- Use an uppercase single-character for formal type parameter. For example,
  - <E> for an element of a collection;
  - <T> for type;
  - <K, V> for key and value.
  - <N> for number
  - S,U,V, etc. for 2nd, 3rd, 4th type parameters

Bounded Generics

- A bounded parameter type is a generic type that specifies a bound for the generic, in the form of **<T extends *ClassUpperBound*>**

- For example, **<T extends Number>** accepts **Number** and its subclasses (such as**Integer** and **Double**).

Bounded Generics

- The following is the design of the interface **java.util.List<E>**:

```
public class MyMath {
    public static <T extends Number> double add(T first, T second) {
        return first.doubleValue() + second.doubleValue();
    }

    public static void main(String[] args) {
        System.out.println(add(55, 66));        // int -> Integer
        System.out.println(add(5.5f, 6.6f));    // float -> Float
        System.out.println(add(5.5, 6.6));      // double -> Double
    }
}
```