Creating Classes and Objects

```java
// Fig. 2.1: Welcome1.java
// Text-printing program.

public class Welcome1
{
    // main method begins execution of Java application
    public static void main( String[] args )
    {
        System.out.println( "Welcome to Java Programming!" );
    } // end method main
} // end class Welcome1
```

/* For several lines */

Scope

Starting point

- Every Java program consists of at least one class that you define.
- Java is case sensitive—uppercase.
- `javac Welcome1.java`
- `javadoc Welcome1.java`
- Class Welcome1 is public and should be declared in file named Welcome1.java

Declaring more than one `public` *class in the same file is a compilation error.*

Class Definition

A class is an encapsulated collection of data and methods to operate on the data. A class definition, data and methods, serves as a blueprint that is used in the creation of new objects of that class.

A class definition typically consists of:

Access modifier: Specifies the availability of the class from other classes

Class keyword: Indicates to Java that the following code block defines a class

Instance fields: Contain variables and constants that are used by objects of the class

Constructors: Are methods having the same name as the class, which are used to control the initial state of any class object that is created

Instance methods: Define the functions that can act upon data in this class

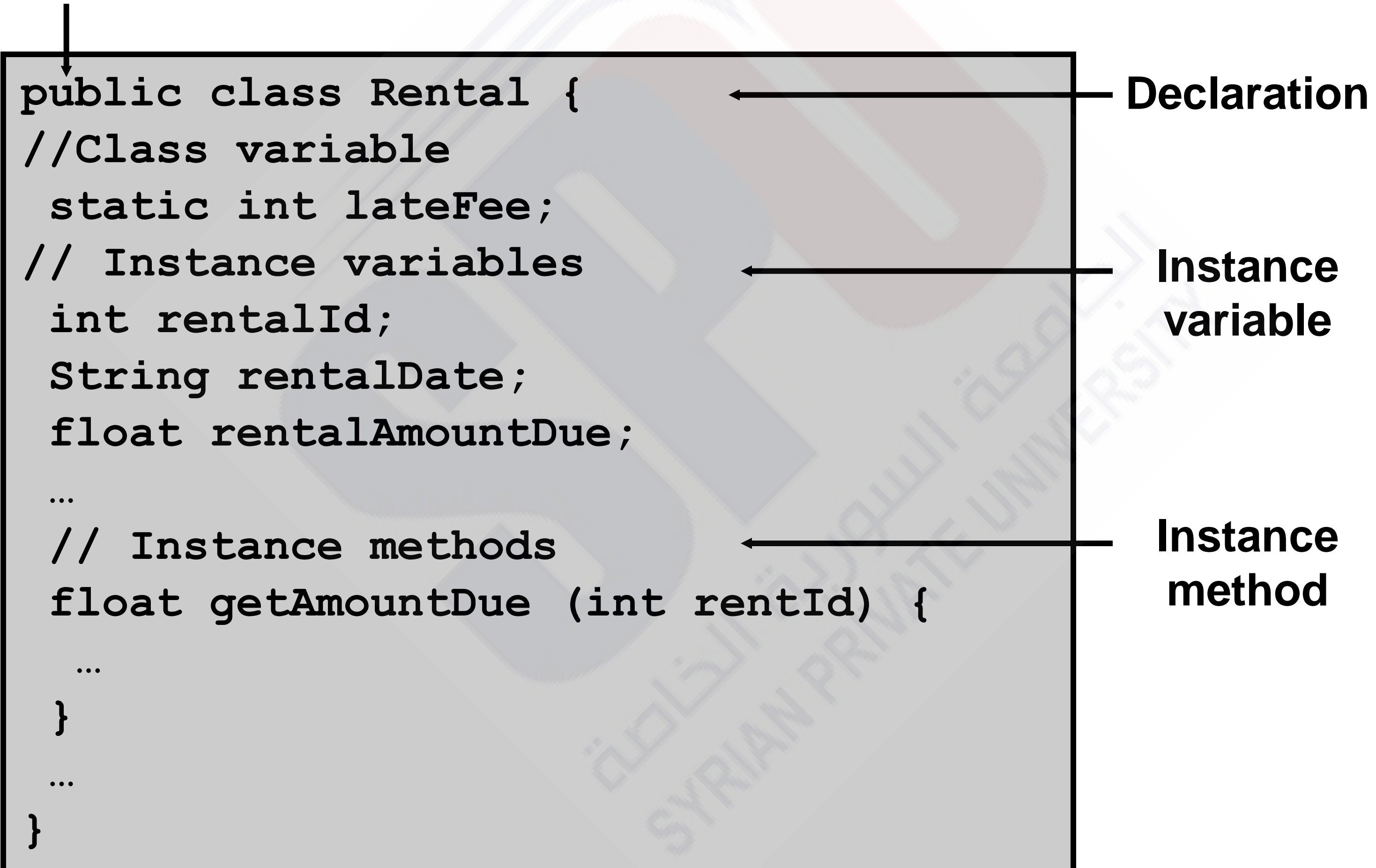
static
Class fields: Contain variables and constants that belong to the class and are shared by all objects of that class

static
Class methods: Are methods that are used to control the values of class fields

The order of the fields, constructors, and methods does not matter in Java. Ordering the parts of a Java program consistently will, however, make your code easier to use, debug, and share. The order listed in the slide is generally accepted.

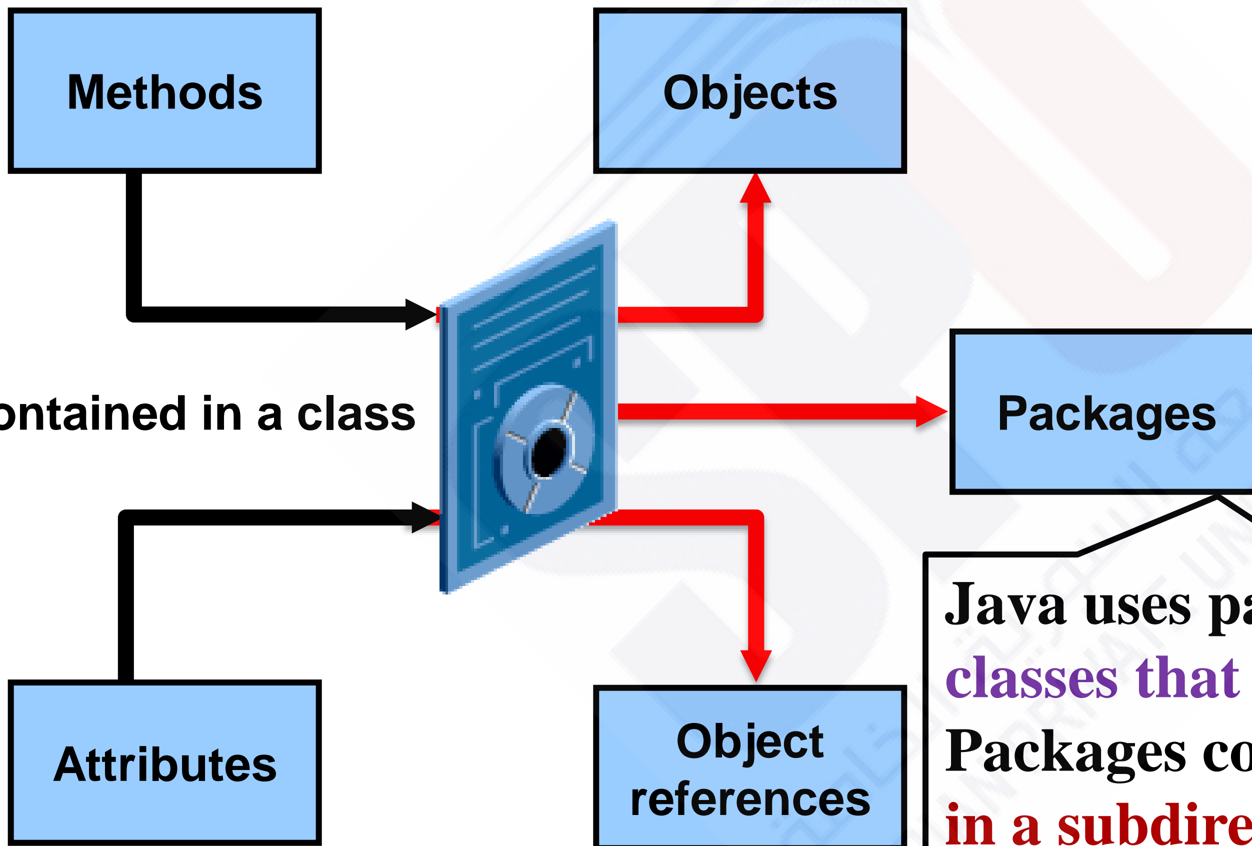
Rental Class: Example

Access modifier




```
public class Rental {
//Class variable
 static int lateFee;
// Instance variables
 int rentalId;
 String rentalDate;
 float rentalAmountDue;
 …
 // Instance methods
 float getAmountDue (int rentId) {
  …
 }
 …
}
```

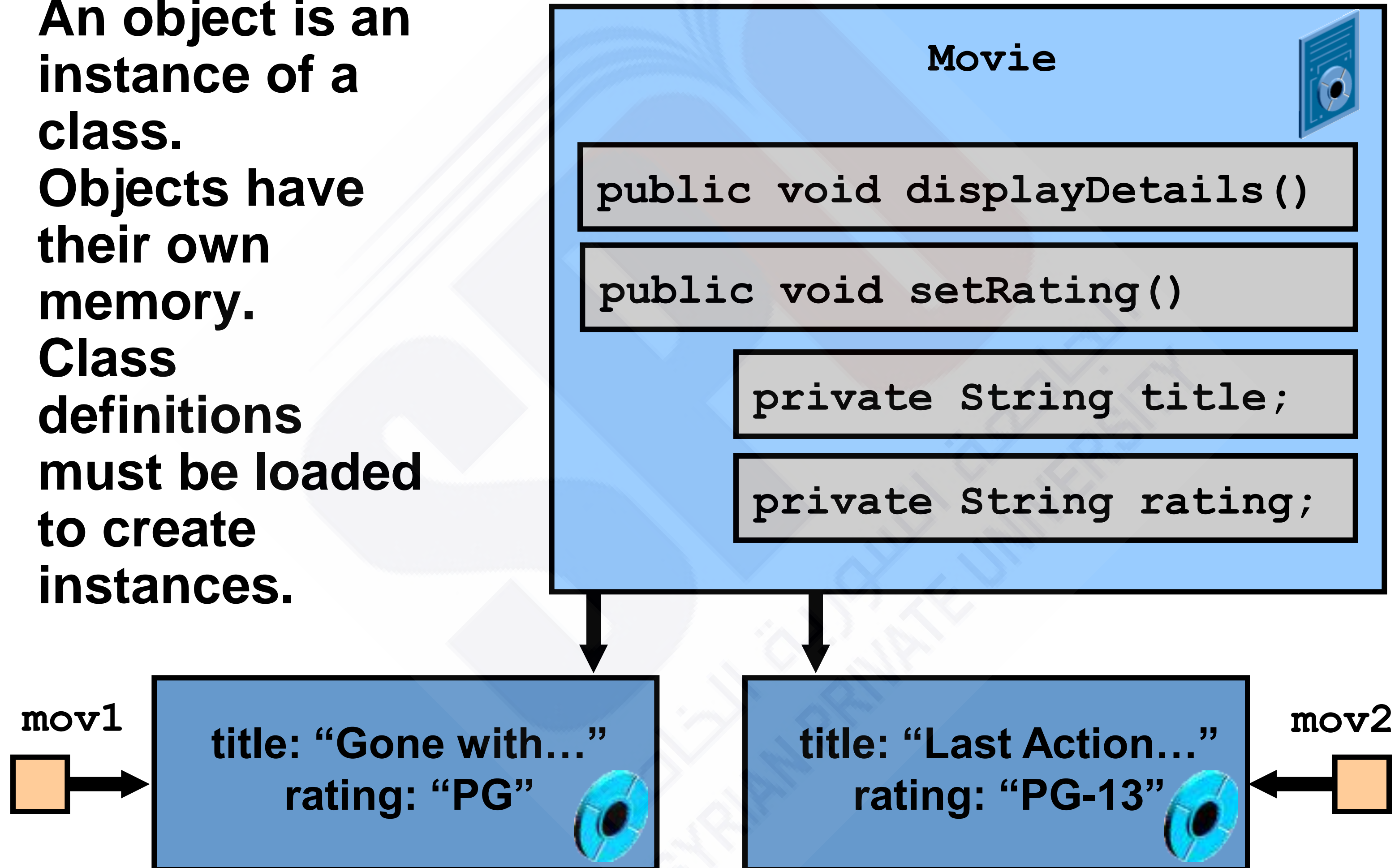
Using Java Classes



Java uses packages to group classes that are logically related. Packages consist of all the classes in a subdirectory. They are also used to control access from programs outside of the package.

Comparing Classes and Objects

- An object is an instance of a class.
- Objects have their own memory.
- Class definitions must be loaded to create instances.

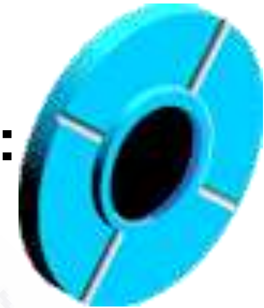


Creating Objects

- Objects are typically created by using the new operator:

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

- For example, to create two Movie objects:



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

title: "Gone with…" rating: "PG"	title: "Last Action…" rating: "PG-13"

This statement creates an instance variable of the Movie type named mov1. It then creates a new instance of Movie by using the new operator and assigns the object reference to the mov1 instance variable.

It is important to remember that the new operator returns a reference to the new object that points to the location of that object in memory.

Using the `new` Operator

The `new` operator performs the following actions:

- Allocates and initializes 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 with…");
```

mov1
(When instantiated)

title: "Gone with…"
rating: "PG"

Using the `new` Operator

Separating Variable Declaration from Object Creation

The declaration of an object reference and the creation of an object are completely independent. In the previous examples, these two parts were combined in a single statement:

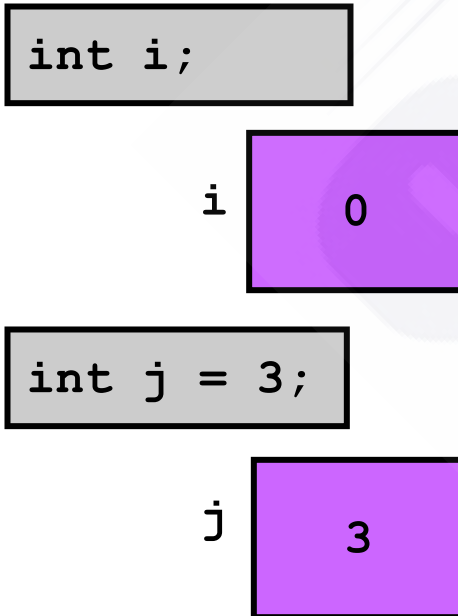
```
Movie mov1 = new Movie();
```

However, you can achieve the same effect with two separate statements, as follows:

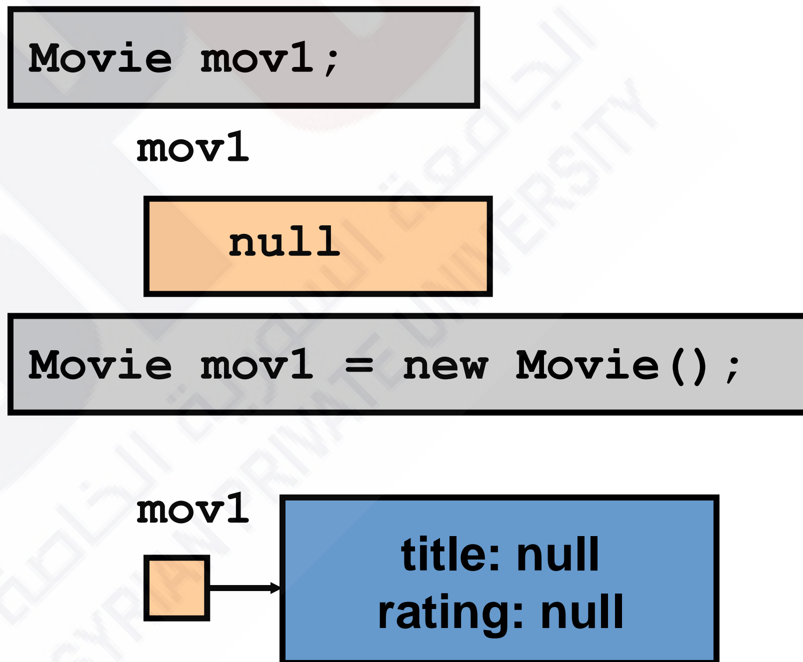
```
Movie mov1;            // Declare an object reference,
                       // capable of referring to a Movie.
mov1 = new Movie();    // Create Movie object, and return the
                       // reference to the mov1 variable.
```

Comparing Primitives and Objects

Primitive variables hold a value.



Object variables hold references.



Using the `null` Reference

- A special `null` value may be assigned to an object reference, but not to a primitive.
- You can compare object references to `null`.
- You can remove the association to an object by setting the object reference to `null`.

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

Using the `null` Reference

Discarding an Object:

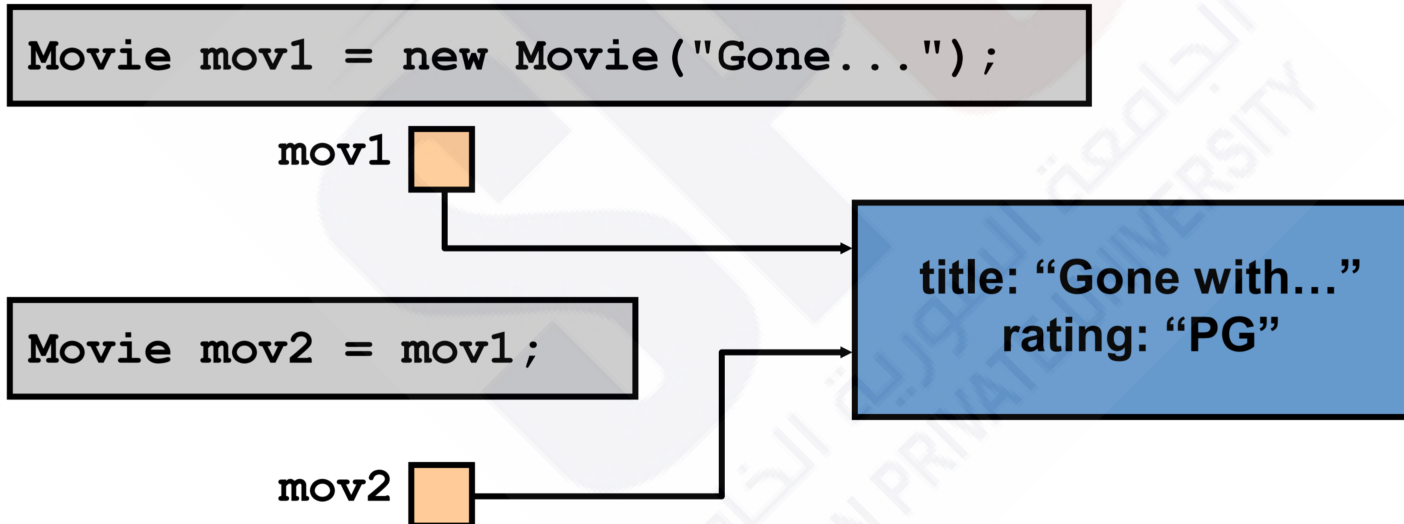
When you have <u>finished using an object</u>, you can set its object reference to null. This indicates that the variable no longer refers to the object.

When there are no more live references to an object, the object will be marked for garbage collection.

The Java Virtual Machine (JVM) automatically decrements the number of active references to an object whenever an object is dereferenced, goes out of scope, or the stored reference is replaced by another reference.

Assigning References

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



Primitive data types

Types: primitive types (gets special treatment)

Primitive type	Size
boolean	—
char	16-bit
byte	8-bit
short	16-bit
int	32-bit
long	64-bit
float	32-bit
double	64-bit
void	—

Size: don't change from one machine architecture to another as they do in most languages.

Primitive data types

Types: primitive types (gets special treatment)

Primitive type	Size
boolean	—
char	16-bit
byte	8-bit
short	16-bit
int	32-bit
long	64-bit
float	32-bit
double	64-bit
void	—

All **numeric** types are **signed**,
so don't go looking for unsigned types.

The **boolean** type
take the literal values **true** or **false**.

Primitive data types

Types: primitive types (gets special treatment)

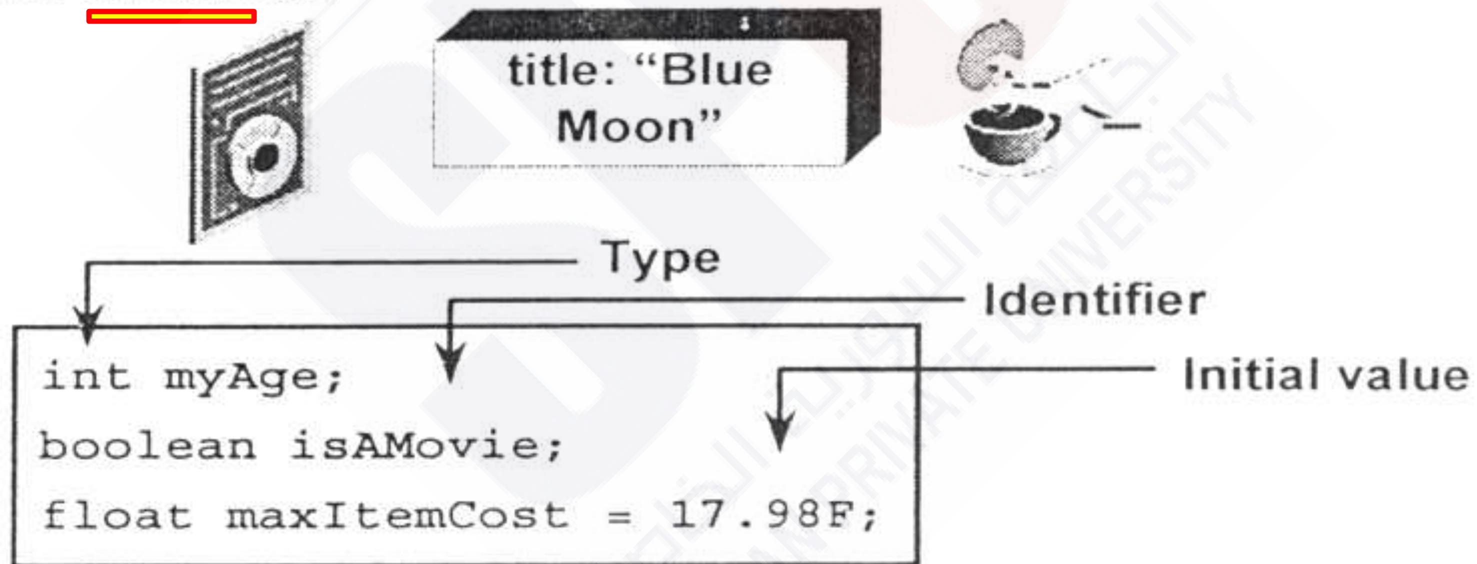
Primitive type	Size	Wrapper type
boolean	—	Boolean
char	16-bit	Character
byte	8-bit	Byte
short	16-bit	Short
int	32-bit	Integer
long	64-bit	Long
float	32-bit	Float
double	64-bit	Double
void	—	Void

The primitive data types also have classes for them.

```
Character C = new Character('x');
```

What Are Variables?

- A variable is a basic unit of storage.
- Variables must be explicitly declared.
- Each variable has a type, an identifier, and a scope.
- There are three types of variables: class, instance, and method.



Variables can be given a default value. If no default value is given, then the instance and class integer variables are set to 0.

What Are 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 a no-arg constructor if and only if a constructor is not explicitly provided.
  - If any constructor is explicitly provided, then the compiler *does not* generate the no-arg constructor.

Default constructors

A default constructor is one without arguments that is used to create a “basic object.”

If you create a class that has no constructors, the compiler will automatically create a default constructor.

```
class Bird {
   int i;
}
```

```
public class DC {
   public static void main(String[] args) {
      Bird nc = new Bird(); // Default!
   }
}
```

Default constructors

A default constructor is one without arguments that is used to create a "basic object."

If you create a class that has no constructors, the compiler will automatically create a default constructor.

If you define any constructors (with or without arguments), the compiler will not synthesize one for you.

```
class Hat {
  Hat   (int i)        {}
  Hat   (double d)    {}
}
```

```
new Hat ();
```

SYRIAN PRIVATE UNIVERSITY

Defining and Overloading Constructors

Because each constructor must have the same name anyway, this simply means that each constructor must have different numbers or types of arguments.

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

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

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

The `Movie` class now provides two constructors.

Sharing Code Between Constructors

A constructor can call another constructor of the same class by using the `this()` syntax.

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

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

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

What happens here?

[Syntax Rules](#)

When one constructor calls another by using the this() syntax, there are a few rules of syntax that you need to be aware of:

1. The call to this() must be the first statement in the constructor.
2. The arguments to this() must match those of the target constructor.

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

```
public class Flower {
 int petalCount = 0;      String s = new String("null");
 Flower (int petals) {      petalCount = petals;
  System.out.println("Constructor w/ int arg only, petalCount= " + petalCount);
 }
 Flower (String ss) {
  s = ss;      System.out.println("Constructor w/ String arg only, s=" + ss);
 }
 Flower(String s, int petals) {
  this(petals);
//!this(s);//Can't call two!// the constructor call must be the first thing you do
  this.s = s; // Another use of "this"
  System.out.println("String & int args");
 }
 Flower() {
  this("hi", 47);      System.out.println("default constructor (no args)");
 }
 void print() {
//! this(11); // Not inside non-constructor!
  System.out.println("petalCount = " + petalCount + " s = "+ s);
 }
 public static void main(String[] args) {
  Flower x = new Flower();      x.print();   } }
```

Initializing Variables

Do not initialize class variables in a constructor; constructors are for initializing instance variables, not class variables.

- **Class variables can be initialized at declaration.**
- **Initialization takes place when the class is loaded.**
- **Use a static initializer block for complex initialization.**
- **All class variables are initialized implicitly to default values depending on data type.**

```
public class Movie {
  private static double minPrice = 1.29;
  private String title, rating;
  private int length = 0;
```

Initializing Class Variables

Complex Initialization of Class Variables:

Complex initialization of class variables is performed in a static initialization block, or *static initializer*.

A static initializer is not named, has no return value, and begins with the static keyword, followed by a block of code inside braces.

It is similar to a constructor except that it executes only once and does not depend on any instance of the class.

```
public class Movie {
  private static double minPrice;
  static {
          Date todaysDate = new Date();
          minPrice = getMinPrice (todaysDate);
  }
}
```

SYRIAN PRIVATE UNIVERSITY
الجامعة السورية الخاصة

This code, like other static initializations, is executed only once: the first time you make an object of that class *or* the first time you access a static member of that class (even if you never make an object of that class).

```
class Spoon {
    static int i;
    static {
        i = 47;
    }
// . . .
```

```
class Tag {
    Tag  (int marker) {
            System.out.println  ("Tag(" + marker + ")");    }
}
class Card{
    Tag t1 = new Tag(1);
   Card() {
       System.out.println  ("Card ()");
      t3 = new Tag(33);
    }

   Tag t2 = new Tag(2);

   void f() {      System.out.println   ( "f()” ) ;   }

   Tag t3 = new Tag(3);
}
public class BM {
  public static void main(String[] args) {
     Card t = new Card();
     t.f();    }  }
```

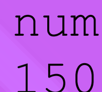
```
class BBB {
  BBB    (int marker)   { System.out.println ("BBB(" + marker + ")"); }
  void f (int marker)     { System.out.println ("f(" + marker + ")");  }
}
class aaa {
  static BBB b1 = new BBB(1);
  aaa() { System.out.println("aaa()");   b2.f(1); }
  void f2(int marker) { System.out.println("f2(" + marker + ")"); }
  static BBB b2 = new BBB(2);
}
class ccc{
  BBB        b3 = new BBB(3);          static  BBB  b4 = new  BBB(4);
  ccc ()          { System.out.println("ccc()");   b4.f(2); }
  void f3(int marker) { System.out.println("f3(" + marker + ")"); }
  static BBB b5 = new BBB(5);
}
public class BM {
  public static void main(String[] args) {
    System.out.println("Creating new ccc() in main");
    new ccc();     System.out.println("Creating new ccc() in main");
    new ccc();             t2.f2(1);         t3.f3(1);
  }
 static aaa  t2 = new aaa();
 static ccc  t3 = new ccc();    }
```

Passing Primitives into Methods

When a <u>primitive</u> or <u>object reference</u> value is passed into a method, a copy of the value is generated:

```
int num = 150;

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




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

This example prints out the following messages:

```
arg: 0
num: 150
```

Passing Object References into Methods

When an object reference is passed into a method, **the object is not copied** but **the pointer to the object is copied:**