Methods

Defining Java Methods

- Always define within a class.
- Specify:
  - Access modifier
  - Static keyword
  - Arguments
  - Return type

```
[access-modifiers] [static] "return-type"
  "method-name" ([arguments]) {
      "java code block" … }
      return
```

SYRIAN PRIVATE UNIVERSITY

Examples of a Method

```
public float getAmountDue (String cust){
 // method variables
 int numberOfDays;
 float due;
 float lateFee = 1.50F;
 String customerName;
// method body
 numberOfDays = getOverDueDays();
 due = numberOfDays * lateFee;
 customerName = getCustomerName(cust);
 return due;
}
```

Declaration

Method variables

Method statements

Return

Declaring Variables

- You can declare variables anywhere in a class block, and outside any method.
- You must declare variables before they are used inside a method.
- It is typical to declare variables at the beginning of a class block.
- The scope or visibility of variables is determined in the code block.
- You must initialize method variables before using them.
- Class and instance variables are automatically initialized.

Examples of Variables in the Context of a Method

```
public float getAmountDue (String cust) {
 float due = 0;
 int numberOfDays = 0;
 float lateFee = 1.50F;
 {int tempCount = 1; // new code block
   due = numberOfDays * lateFee;
   tempCount++;
   …
 }                  // end code block
 return due;
}
```

Method variables

Temporary variables

SYRIAN PRIVATE UNIVERSITY
الجامعة السورية الخاصة

What Are Class Methods?

- Class methods are:
  - Shared by all instances
  - Useful for manipulating class variables
  - Declared as static

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

- A class method is called by using the name of the class or an object reference.

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

SYRIAN PRIVATE UNIVERSITY

What Are Class Methods?

A class method does not operate on a single object, and so it does not have a this reference. A class method can access only the class variables and class methods of its class.

You can call class methods by using an object reference before the dot, rather than the name of the class, but the method can still access only class variables and class methods.

You may want to create a method that is used outside of any instance context. Declare a method to be static, which may only call other static methods directly.

Static methods may not refer to their superclass or its methods.

Guided Practice: Class Methods or Instance Methods

```
public class Movie {

 private static float price = 3.50f;
 private String rating;
  …
 public static void setPrice(float newPrice) {
    price = newPrice;
 }
 public String getRating() {
    return String;
 }
}
```

```
Movie.setPrice(3.98f); Movie
mov1 = new Movie(…);
mov1.setPrice(3.98f);
String a = Movie.getRating();
String b = mov1.getRating();
```

Legal or not?

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);
  } …
```

Syrian Private University

Method Overloading

- **Method overloading**
  - **Methods of the same name declared in the same class**
  - **Must have different sets of parameters**
  - **Method signatures is formed from its name, together with the number and types of its arguments**
  - **The method return type is not considered part of the method signatures.**
- **Distinguishing Between Overloaded Methods**
  - **The compiler distinguishes overloaded methods by their signatures—the methods' names and the number, types and order of their parameters.**
- **Return types of overloaded methods**
  - ***Method calls cannot be distinguished by return type.***
- **Overloaded methods need not have the same number of parameters.**

Overloading Methods

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

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

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

A method signatures is formed from its name, together with the number and types of its arguments.

The method return type is not considered part of the method signatures.

Method overloading

Method overloading **is essential to allow the same method name to be used with different argument types.**

Even differences in the ordering of arguments are sufficient to distinguish two methods.

جامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

```
class Tree {     int height;
  Tree () {
    System.out.println("Planting a seedling");       height = 0;
  }
  Tree (int i) {
    System.out.println("Creating new Tree that is " + i + " feet tall");    height = i;
  }
  void info()              {  System.out.println("Tree is " + height + " feet tall");   }
  void info(String s) {  System.out.println(s + ": Tree is " + height + " feet tall");   }
}
public class BM {
 public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();       t.info("overloaded method");
}
    // Overloaded constructor:
    new Tree();     }   }
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

f ? (5)

f1(int)
f2(int)
f3(int)
f4(int)
f5(long)
f6(float)
f7(double)

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
char x = 'x';
```

```
f ? (x)
```

```
f1(char)
f2(int)
f3(int)
f4(int)
f5(long)
f6(float)
f7(double)
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
byte x = 0;
```

```
f ? (x)
```

```
f1(byte)
f2(byte)
f3(short)
f4(int)
f5(long)
f6(float)
f7(double)
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
short x = 0;
```

```
f ? (x)
```

```
f1(short)
f2(short)
f3(short)
f4(int)
f5(long)
f6(float)
f7(double)
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
int x = 0;
```

```
f ? (x)
```

```
f1(int)
f2(int)
f3(int)
f4(int)
f5(long)
f6(float)
f7(double)
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
long x = 0;
```

```
f ? (x)
```

```
f1(long)
f2(long)
f3(long)
f4(long)
f5(long)
f6(float)
f7(double)
```

SYRIAN PRIVATE UNIVERSITY

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
float x = 0;
```

f ? (x)

```
f1(float)
f2(float)
f3(float)
f4(float)
f5(float)
f6(float)
f7(double)
```

Overloading with primitives

What happens when a primitive is handed to an overloaded method.

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(double x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(float x)
void f3(double x)
```

```
void f4(int x)
void f4(long x)
void f4(float x)
void f4(double x)
```

```
void f5(long x)
void f5(float x)
void f5(double x)
```

```
void f6(float x)
void f6(double x)
```

```
void f7(double x)
```

```
double x = 0;
```

```
f ? (x)
```

```
f1(double)
f2(double)
f3(double)
f4(double)
f5(double)
f6(double)
f7(double)
```

Overloading with primitives

```
void f1(char x)
void f1(byte x)
void f1(short x)
void f1(int x)
void f1(long x)
void f1(float x)
void f1(double x)
```

```
void f2(byte x)
void f2(short x)
void f2(int x)
void f2(long x)
void f2(float x)
void f2(char x)
```

```
void f3(short x)
void f3(int x)
void f3(long x)
void f3(byte x)
void f3(char x)
```

```
void f4(int x)
void f4(byte x)
void f4(short x)
void f4(char x)
```

```
void f5(char x)
void f5(byte x)
void f5(short x)
```

```
void f6(char x)
void f6(byte x)
```

```
void f7(char x)
```

```
double x = 0;
```

f ? (x)

```
f1(x);
f2((float)x);
f3((long)x);
f4((int)x);
f5((short)x);
f6((byte)x);
f7((char)x);
```

Overloading on return values

```
void    f() { }
int     f() { }
```

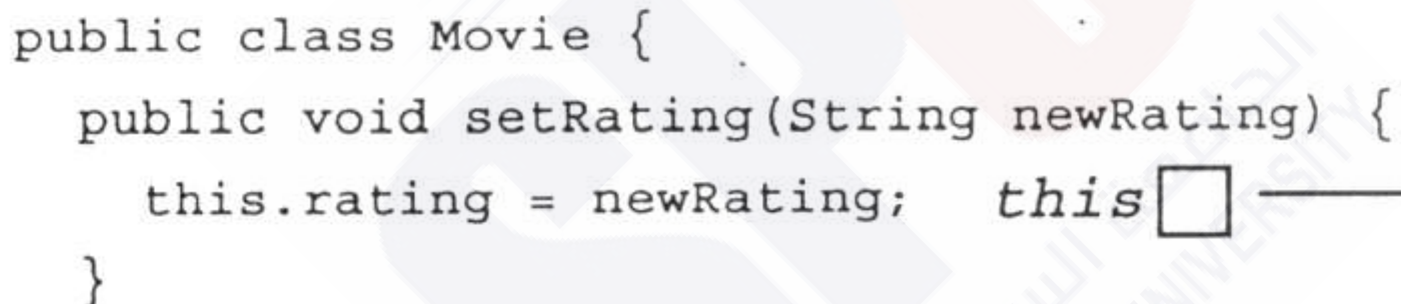
You cannot use return value types to distinguish overloaded methods.

SYRIAN PRIVATE UNIVERSITY الجامعة السورية الخاصة

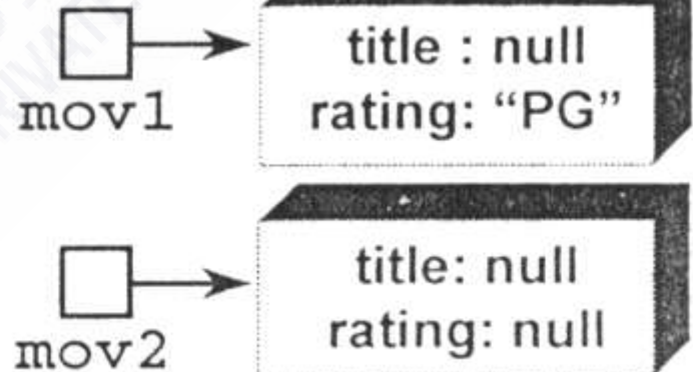
```
package ex1d;
public class BM {
  static int f1 () {
      System.out.println (" koko ");
         return 1;
  }
  public static void main(String[] args) {
    f1 ();
    System.out.println (" ------------------------- ");
    int x = f1 ();
  }
}
```

Using the `this` Reference

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




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



The this keyword

```
class Banana {
   void  f  (int i) {
      / * ... */
   }
}
```

```
Banana a = new Banana(), b = new Banana();
```

```
a.f (1);
b.f (2);
```

How can that method know whether it's being called for the object a or b?

The compiler does some undercover work for you.

```
Banana.f (a,1);
Banana.f (b,2);
```

The reference to the object that's being manipulated passed to the method f( ).

The this keyword

```
a.f (1);
b.f (2);
```

```
Banana.f (a,1);
Banana.f (b,2);
```

Suppose you're inside a method and you'd like to get the reference to the current object.

The this keyword—which can be used only inside a method—produces the reference to the object the method has been called for.

```java
public class Leaf {
    int i = 0;
    Leaf increment () {
        i++;
        return   this;
    }
    void   print () {
         System.out.println ("i = " + i);
    }
     public static void main (String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
}
```

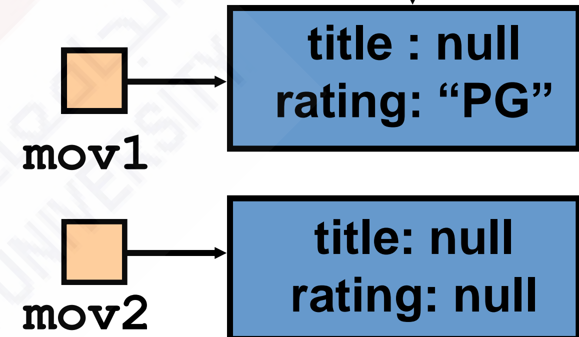
Using the `this` Reference

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

the two following statements are equivalent:

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

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



There are two circumstances where <u>you must use an explicit this reference</u>:

1. **When the name of an instance variable is hidden by a formal argument of an instance method.** For example, if there is an instance variable called name, and an argument that is also called name, then the argument hides the instance variable. Any reference to name accesses the argument, not the variable. To access the instance variable, you must use this.name.
2. **When you need to pass a reference to the current object as an argument to another method**

```
class A {
  int x = 5;
  void F () {
    if (x == 22)
        fff (this);
    System.out.println (x);
  }
  void fff (A   S) {     S.x = 8;    }
}

public class BM {
  public static void main(String[] args) {
    A t = new A(), y = new A();
    t.x = 99;       y.x = 22;
    t.F();          y.F();
    System.out.println ("----------------------------------");
    System.out.println (t.x);      System.out.println (y.x);      } }
```

```
99
8
--------
99
8
```

Initializing Instance Variables

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

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

- All instance variables are initialized implicitly to default values depending on data type.
- More complex initialization should be placed in a constructor.

`final` Variables, Methods, and Classes

- A `final` variable is a constant and cannot be modified.
  - It must therefore be initialized.
  - It is often declare `public static` for external use.
- A `final` method cannot be overridden by a subclass.
- A `final` class cannot be subclassed. Cannot be inherited from

```
public final class Color {
  public final static Color black=new Color(0,0,0);
  …
}
```

Final data

A field that is both static and final has only one piece of storage that cannot be changed.

When using final with object references rather than primitives, with a primitive, final makes the *value* a constant, but with an object reference, final makes the *reference* a constant.

Once the reference is initialized to an object, it can never be changed to point to another object. However, the object itself can be modified; Java does not provide a way to make any arbitrary object a constant.

\javag\exp\ex2m\ex2L

Blank finals

Java allows the creation of *blank finals*, which are fields that are declared as final but are not given an initialization value. In all cases, the blank final *must* be initialized before it is used, and the compiler ensures this.

However, blank finals provide much more flexibility in the use of the final keyword since, for example, a final field inside a class can now be different for each object, and yet it retains its immutable quality.

```java
class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}
public class BlankFinal {
  private final  int    i = 0;          // Initialized final
  private final  int    j;              // Blank final
  private final  Poppet    p;        // Blank final reference
  // Blank finals MUST be initialized in the constructor:
  public BlankFinal() {
    j = 1;                          // Initialize blank final
    p = new Poppet(1);    // Initialize blank final reference
  }
  public BlankFinal(int x) {
    j = x;                          // Initialize blank final
    p = new Poppet(x);     // Initialize blank final reference
  }
  public static void main(String[] args) {
          new BlankFinal();                new BlankFinal(47);      }    }
```

Final arguments

Java allows you to make arguments final by declaring them as such in the argument list. This means that inside the method you cannot change what the argument *reference* points to.

SYRIAN PRIVATE UNIVERSITY
الجامعة السورية الخاصة

```
class Gizmo {
  public void spin() {}
}
public class FinalArguments {
  void   with   (final    Gizmo     g) {
    //! g = new Gizmo();                    // Illegal -- g is final
  }
  void without(Gizmo    g) {
    g = new Gizmo();                        // OK -- g not final
    g.spin();
  }
  // void f(final   int     i) {
  //    i++;    }                            // Can't change
  // You can only read from a final primitive:
  int   g (final   int    i) {
          return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);         }      }
```

Final methods

There are two reasons for final methods:

The first is to put a “lock” on the method to prevent any inheriting class from changing its meaning. This is done for design reasons when you want to make sure that a method’s behavior is retained during inheritance and **cannot be overridden**.

The second reason for final methods is efficiency.

SYRIAN PRIVATE UNIVERSITY الجامعة السورية الخاصة

final and private

Any private methods in a class are implicitly final. Because you can't access a private method, you can't override it. You can add the final specifier to a private method, but it doesn't give that method any extra meaning.

ex2o

Final classes

When you say that an entire class is final (by preceding its definition with the final keyword), you state that you don't want to inherit from this class or allow anyone else to do so.

ex2p

Note that the fields of a final class can be final or not, as you choose. The same rules apply to final for fields regardless of whether the class is defined as final. However, because it prevents inheritance, all *methods* in a final class are implicitly final, since there's no way to override them. You can add the final specifier to a method in a final class, but it doesn't add any meaning.

Reclaiming Memory

- When all references to an object are lost, the object is marked for garbage collection.
- Garbage collection reclaims memory that is used by the object.
- Garbage collection is automatic.
- There is no need for the programmer to do anything, but the programmer can give a hint to `System.gc();`.

AL-ALBAYAN PRIVATE UNIVERSITY

Using the `finalize()` Method

- If an object holds a resource such as a file, then the object should be able to clean it up.
- 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?

The only solution is to manage such resources manually.

No guarantee regarding when this will happen or that it will happen before the program exits.

SYRIAN PRIVATE UNIVERSITY الجامعة السورية الخاصة

```
class Book {
  boolean  checkedOut  =  false;
  Book (boolean checkOut) {    checkedOut = checkOut;    }
  void checkIn () {      checkedOut = false;     }
  public void finalize() {
    if (checkedOut)
      System.out.println("Error: checked out");
  }   }

public class BM {
  public static void main(String[] args) {
    Book novel = new Book(true);
    novel.checkIn();
    // Drop the reference, forget to checkIn();
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
  }  }
```