Advanced Computer Programming

[Lecture 06]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

OBJECTS and CLASSES

In an **object-oriented** program, you don’t simply manipulate numbers and strings, but you work with **objects** that are meaningful for your application. Objects with the same behavior are grouped into **classes**. A programmer provides the desired behavior by specifying and implementing **methods** for these classes.

Object-Oriented Programming

Definition

Object-oriented programming is a programming style in which tasks are solved by collaborating objects, where **objects** have their own set of data, together with a set of methods that act upon the data.

Some objects you've used:

Object-Oriented Programming

Definition

Object-oriented programming is a programming style in which tasks are solved by collaborating objects, where **objects** have their own set of data, together with a set of methods that act upon the data.

Some objects you've used:

- `String` objects to work with strings.
- `Scanner` objects for input operations.

Classes and Objects

Definition

A **class** describes a set of objects with the same behavior.

Classes and Objects

Definition

A **class** describes a set of objects with the same behavior.

Example:

- The `String` class describes the behavior of all strings;
  - How a string stores its characters.
  - Which methods can be used with strings (`length, substring, charAt, ...`).
  - How the methods are implemented.

Classes and Objects

Definition

Every class has a **public interface**: a collection of methods through which the objects of the class can be manipulated.

Classes and Objects

Definition

Every class has a **public interface**: a collection of methods through which the objects of the class can be manipulated.

Definition

Encapsulation is the act of providing a public interface and hiding the implementation details. Encapsulation enables changes in the implementation without affecting users of a class.

Encapsulation in Real World

You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.

Encapsulation in Real World

A driver of an electric car doesn’t have to learn new controls even though the car engine is very different. Neither does the programmer who uses an object with an improved implementation as long as the same methods are used.

Classes Define Types

Recall that:

Type = Size + Operations

Classes Define Types

Recall that:

Type = Size + Operations

When you define a class you should define

- The **data** that its objects use (defining the size of its objects).
- The **methods** with which other objects can interact (defining operations).

Implementing a Simple Class

Tally Counter

Implementing a Simple Class

- Choosing a name for the class: `Counter`

Implementing a Simple Class

- Choosing a name for the class: `Counter`
- Methods (Operations) we need
  - Increasing by one: `count()`
  - See the counter's value: `getValue()`

Implementing a Simple Class

- Choosing a name for the class: `Counter`
- Methods (Operations) we need
  - Increasing by one: `count()`
  - See the counter's value: `getValue()`
- In Java, you use the `new` operator to construct objects:

```
Counter tally = new Counter();
```

- In Java, you use the dot operator to access object methods:

```
tally.count();
tally.count();
int result = tally.getValue(); // Sets result to 2
```

Implementing a Simple Class

- Choosing a name for the class: `Counter`
- Methods (Operations) we need
  - Increasing by one: `count()`
  - See the counter's value: `getValue()`
- In Java, you use the `new` operator to construct objects:

```
Counter tally = new Counter();
```

- In Java, you use the dot operator to access object methods:

```
tally.count();
tally.count();
int result = tally.getValue(); // Sets result to 2
```

- Specifying how each counter object stores its data: `int value`
  - An object stores its data in **instance variables**.

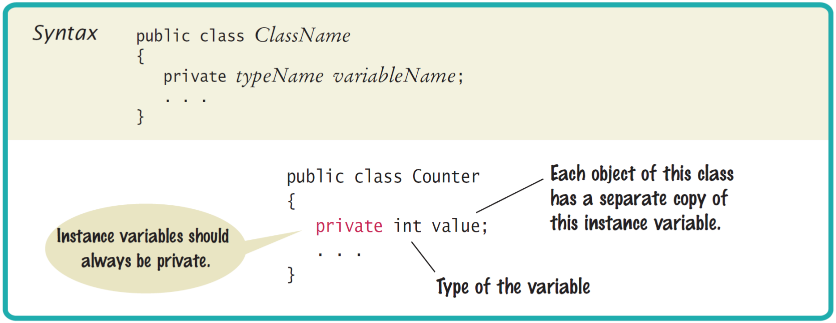
Instance Variables

Definition

An **instance variable** is a storage location that is present in each object of the class.

An instance variable declaration consists of the following parts:

- A **modifier** (private)
- The **type** of the instance variable (such as int)
- The **name** of the instance variable (such as value)



Syntax

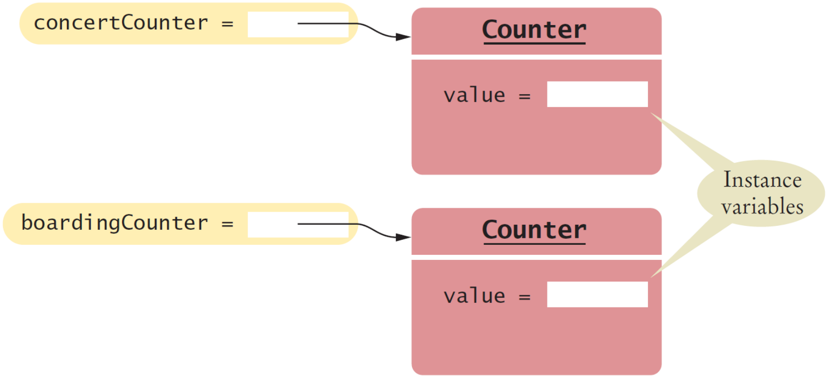
```
public class ClassName
{
   private typeName variableName;
   . . .
}
```

```
public class Counter
{
   private int value;
   . . .
}
```

Instance variables should always be private.
Each object of this class has a separate copy of this instance variable.
Type of the variable

Instance Variables

Each object of a class has its own set of instance variables. For example, consider `concertCounter` and `boardingCounter` to be two objects of the `Counter` class;



Instance Methods

Important

An instance method can access the instance variables of the object on which it acts (different from `static` methods).

- The `count` method advances the counter value by 1.

```
public void count()
{
    value = value + 1;
}
```

- The `getValue` method returns the current value.

```
public int getValue()
{
    return value;
}
```

- Both of the methods work with the instance variable `value`, **the one belonging to the object on which the method is invoked**.

The `private` Access Specifier

Usage

The `private` specifier restricts access to the methods of the same class.

- A user cannot simply access the instance variables (`value` for example). Because they are declared with `private` access specifier.
- Private instance variables are an essential part of encapsulation. They allow a programmer to hide the implementation of a class from a class user.

The Public Interface of a Class

Definition

The **public interface** of a class consists of all methods that a user of the class may want to apply to its objects.

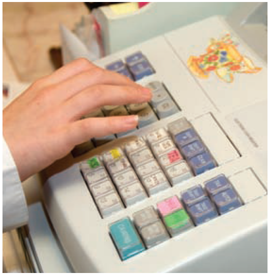
The Public Interface of a Class

Definition

The **public interface** of a class consists of all methods that a user of the class may want to apply to its objects.

When designing a class, you start by specifying its public interface. For example we want the following methods on a cash register object:

- Add the price of an item.
- Get the total amount owed, and the count of items purchased.
- Clear the cash register to start a new sale.



The Public Interface of a Class

Implementation v.s. Interface

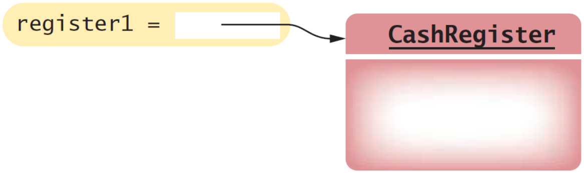
The method declarations and comments make up the **public interface** of the class. The data and the method bodies make up the **private implementation** of the class.

```
/**
   A simulated cash register that tracks the item
   count and the total amount due.
*/
public class CashRegister
{
   private data—see Section 8.4

   /**
      Adds an item to this cash register.
      @param price  the price of this item
   */
   public void addItem(double price)
   {
      implementation—see Section 8.5
   }

   /**
      Gets the price of all items in the current sale.
      @return  the total price
   */
   public double getTotal()
   {
      implementation—see Section 8.5
   }

   /**
      Gets the number of items in the current sale.
      @return  the item count
   */
   public int getCount()
   {
      implementation—see Section 8.5
   }

   /**
      Clears the item count and the total.
   */
   public void clear()
   {
      implementation—see Section 8.5
   }
}
```

Object Reference

When you create an object of a class using the `new` operator, it creates that object somewhere in the memory and return a reference to it.
For example:

```
CashRegister register1 = new CashRegister();
```



Mutators and Accessors

Instance methods of the public interface can be classified into two categories:

- **Mutator** method: modifies the object on which it operates.

  ```
  tally.count();
  ```

Mutators and Accessors

Instance methods of the public interface can be classified into two categories:

- **Mutator** method: modifies the object on which it operates.
  `tally.count();`
- **Accessor** method: queries the object for some information without changing it.
  `tally.getValue();`

Designing the Data Representation

When implementing a class, you have to determine which data each object needs to store.

- Go through all methods and consider their data requirements (start with the accessor methods).
    - You should choose to compute or to store data.

Designing the Data Representation

When implementing a class, you have to determine which data each object needs to store.

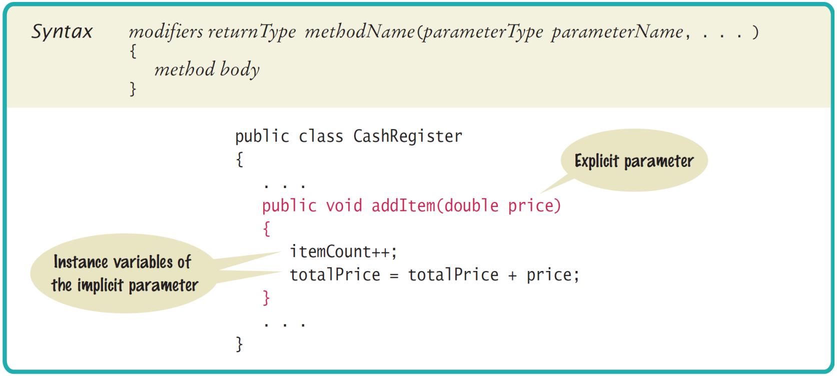
- Go through all methods and consider their data requirements (start with the accessor methods).
  - You should choose to compute or to store data.

Back to cash register example, we choose to store data for number of items and total price:

```
public class CashRegister
{
   private int itemCount;
   private double totalPrice;
   . . .
}
```

Implementing Instance Methods

When implementing a class, you need to provide the bodies for all methods.



Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*

```
{
   method body
}
```

```
public class CashRegister
{
   . . .
   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }
   . . .
}
```

Explicit parameter
Instance variables of the implicit parameter

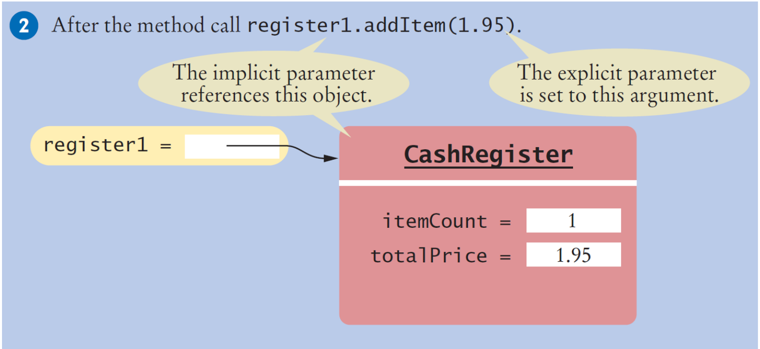
Implicit and Explicit Parameters

Definition

The object on which a method is invoked is called the **implicit parameter** of the method.

Definition

parameters that are explicitly mentioned in the method declaration, are called **explicit parameters**.


2 After the method call register1.addItem(1.95).
The implicit parameter references this object.
The explicit parameter is set to this argument.
register1 =
CashRegister
itemCount = 1
totalPrice = 1.95

Constructors

Usage

A **constructor** initializes the instance variables of an object. The constructor is automatically called whenever an object is created with the `new` operator.

The name of a constructor is identical to the name of its class. Constructors never return values.

```
public class CashRegister
{
   . . .

   /**
      Constructs a cash register with cleared item count and total.
   */
   public CashRegister() // A constructor
   {
      itemCount = 0;
      totalPrice = 0;
   }
}
```

Constructors

A class can have multiple constructors. This allows you to declare objects in different ways.

```
public class BankAccount
{
   . . .

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount() { . . . }

   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance) { . . . }
}
```

When you construct an object, the compiler chooses the constructor that matches the arguments that you supply.

Constructors

If you do not initialize an instance variable in a constructor, it is automatically set to a default value:

- Numbers are set to zero.
- Boolean variables are initialized as false.
- Object and array references are set to the special value `null` that indicates that no object is associated with the variable.
  Don't forget to initialize object references in a constructor.

If you do not provide a constructor, a constructor with no arguments is generated.

Example

```java
/**
   A simulated cash register that tracks the item count and
   the total amount due.
*/
public class CashRegister
{
   private int itemCount;
   private double totalPrice;

   /**
      Constructs a cash register with cleared item count and total.
   */
   public CashRegister()
   {
      itemCount = 0;
      totalPrice = 0;
   }

   /**
      Adds an item to this cash register.
      @param price the price of this item
   */
   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }

   /**
      Gets the price of all items in the current sale.
      @return the total amount
   */
   public double getTotal()
   {
      return totalPrice;
   }

   /**
      Gets the number of items in the current sale.
      @return the item count
   */
   public int getCount()
   {
      return itemCount;
   }

   /**
      Clears the item count and the total.
   */
   public void clear()
   {
      itemCount = 0;
      totalPrice = 0;
   }
}
```

Method Overloading

Definition

When the same method name is used for more than one method, then the name is **overloaded**.

Method Overloading

Definition

When the same method name is used for more than one method, then the name is **overloaded**.

In Java you can overload method names provided that the parameters are different (in their types or count). Consider the following methods:

- **`public int add(int a)`**
  returns `a + 1`.
- **`public int add(int a, int b)`**
  returns `a + b`.
- **`public string add(string a, string b)`**
  returns `a + b`.

When you call an overloaded method, the compiler chooses the one that matches the arguments that you supply.

Testing a Class

Definition

A **unit test** verifies that a class works correctly in isolation, outside a complete program.

You can write a tester class, which is a class with a `main` method that contains statements to run methods of another class. A tester class typically carries out the following steps:

- Construct one or more objects of the class that is being tested.
- Invoke one or more methods.
- Print out one or more results.
- Print the expected results.

Testing a Class: Example

```java
/**
   This program tests the CashRegister class.
*/
public class CashRegisterTester
{
   public static void main(String[] args)
   {
      CashRegister register1 = new CashRegister();
      register1.addItem(1.95);
      register1.addItem(0.95);
      register1.addItem(2.50);
      System.out.println(register1.getCount());
      System.out.println("Expected: 3");
      System.out.printf("%.2f\n", register1.getTotal());
      System.out.println("Expected: 5.40");
   }
}
```

Program Run

```
3
Expected: 3
5.40
Expected: 5.40
```

Testing a Class: Example

To produce a program, you need to combine the `CashRegister` and `CashRegisterTester` classes.

1. Make a new subfolder for your program.
2. Make two files, one for each class.
3. Compile both files.
4. Run the test program.

Implementing a Class: Guidlines

1. Get an informal list of the responsibilities of your objects.
2. Specify the public interface.
3. Document the public interface.
4. Determine instance variables.
5. Implement constructors and methods.
6. Test your class.

Exercise (`Tally.java`)

Implement the tally counter class.

Patterns for Object Data

- Keeping a Total
  An instance variable for the total is updated in methods that increase or decrease the total amount.

Patterns for Object Data

- Keeping a Total

  An instance variable for the total is updated in methods that increase or decrease the total amount.
- Counting Events

  A counter that counts events is incremented in methods that correspond to the events.

Patterns for Object Data

- Keeping a Total

  An instance variable for the total is updated in methods that increase or decrease the total amount.
- Counting Events

  A counter that counts events is incremented in methods that correspond to the events.
- Collecting Values

  An object can collect other objects in an array or array list.

Patterns for Object Data

- Keeping a Total
  An instance variable for the total is updated in methods that increase or decrease the total amount.
- Counting Events
  A counter that counts events is incremented in methods that correspond to the events.
- Collecting Values
  An object can collect other objects in an array or array list.
- Managing Properties of an Object
  An object property can be accessed with a getter method and changed with a setter method.

Patterns for Object Data

- Keeping a Total
  An instance variable for the total is updated in methods that increase or decrease the total amount.
- Counting Events
  A counter that counts events is incremented in methods that correspond to the events.
- Collecting Values
  An object can collect other objects in an array or array list.
- Managing Properties of an Object
  An object property can be accessed with a getter method and changed with a setter method.
- Modeling Objects with Distinct States
  If your object can have one of several states that affect the behavior, supply an instance variable for the current state.

Patterns for Object Data

- Keeping a Total
  An instance variable for the total is updated in methods that increase or decrease the total amount.
- Counting Events
  A counter that counts events is incremented in methods that correspond to the events.
- Collecting Values
  An object can collect other objects in an array or array list.
- Managing Properties of an Object
  An object property can be accessed with a getter method and changed with a setter method.
- Modeling Objects with Distinct States
  If your object can have one of several states that affect the behavior, supply an instance variable for the current state.
- Describing the Position of an Object
  To model a moving object, you need to store and update its position.

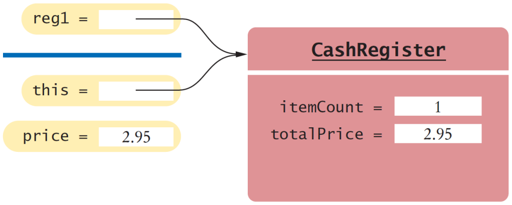
The this Reference

Usage

In an instance method, the **this** reference refers to the implicit parameter.

You don't usually need to use the this reference, but you can.

```
void addItem(double price)
{
   this.itemCount++;
   this.totalPrice = this.totalPrice + price;
}
```



Static Variables and Methods

Usage

A **`static` variable** belongs to the class, not to any object of the class.

Usage

A **`static` method** is not invoked on an object.

- In `static` methods, you have no access to `this` reference.
- To use `static` variables or methods, you should use the dot operator in front of the class name instead of the object reference (remember the `Math` class).