Java Coding 5

To object or not…

From the beginning…

- History of programming paradigms
  - GoTo Programming (spaghetti code!)
  - Structured Programming
  - Object-Oriented Programming

- Paradigm changes response to
  - Need to build ever larger programs
  - Correctly
  - On time
  - On budget

Key Attributes of OOP

- Abstraction, Encapsulation, Inheritance, Polymorphism

What?

- Ease of reuse
  - Speeds implementation & facilitates maintenance.
  - Component-based approach
    - Easy to use existing code modules
    - Easy to modify code to fit circumstances!
- A natural way to view/model world
  - Makes design quicker, easier & less error-prone.

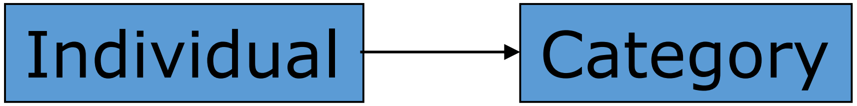
The world as we see it…

- Look around & what do you see?
  - Things (books, chairs, tables, people!)
- Actually, see individual things!
  - Ayse, David, my textbook, that chair, Mehmet’s pencil, etc.

- The world is
  - a set of things
  - interacting with each other.

Describing the world (1)

- Describe a particular person
  - Ayse has long blond hair, green eyes, is 1.63m tall, weighs 56Kg and studies computer engineering. Now lying down asleep.
  - Mehmet studies electronics, has short black hair and brown eyes. He is 180cm and 75 kilos. Now running to class!
- Notice how all have specific values of
  - name, height, weight, eye colour, state, …



Individual → Category

Describing the world (2)

- Describe some particular books

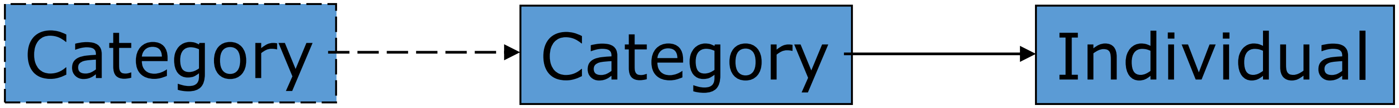
- Your textbooks for example

- What features (properties & functionality) characterize a book?

- How about cars?

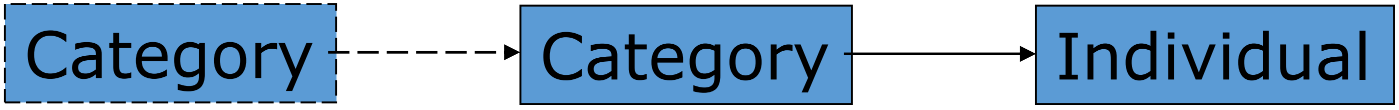
Describing the world (3)

- Type/category determine an object’s properties & functionality
  - Person
    - has name, height, weight, can run, sleep, …
  - Category gives default properties
    - “Ayse is a person with green eyes.”
      We infer/assume she has two of them, as well as two legs, arms, nose, mouth, hair, can speak, run, sleep, etc!
    - Can concentrate on “relevant” properties



Describing the world (4)

- We have categories of categories as well
- living things (animals (elephants, cats, dogs)
- person (student (undergraduate, graduate)
- faculty member(prof, assoc prof, assist prof), admin staff)
- furniture (living room, kitchen, bedroom)



Category

Category

Individual

Java OOP terminology

- **Class** - Category
  - Properties/states
  - Functionality/Services (examines/alters state)

data

methods

- **object** - Individual/unique thing *(an instance of a class)*
  - Particular value for each property/state
  - & functionality of all members of class.

Java OOP terminology

- **Class** - Category
  - Properties/states
  - Functionality/Services (examines/alters state)

data

methods

- Class acts as blueprint for creating new objects
- Properties/states correspond to memory locations having particular values
- Functionality corresponds to the methods that examine/manipulate the property values

Objects

- **Object:** an entity in your program that you can manipulate by calling one or more of its methods.
- **Method:** consists of a sequence of instructions that can access the data of an object.
  - You do not know what the instructions are
  - You do know that the behavior is well defined
- `System.out` has a `println` method
  - You do not know how it works
  - What is important is that it does the work you request of it

Classes

- A class describes a set of objects with the same behavior.
- Some string objects:

```
"Hello World"
"Goodbye"
"Mississippi"
```

- You can invoke the same methods on all strings.

- `System.out` is a member of the `PrintStream` class that writes to the console window.
- You can construct other objects of `PrintStream` class that write to different destinations.
- All `PrintStream` objects have methods `println` and `print`.

Classes

- Objects of the `PrintStream` class have a completely different behavior than the objects of the `String` class.
- Different classes have different responsibilities
  - A string knows about the letters that it contains
  - A string doesn't know how to send itself to a console window or file.

© Peter Mukherjee/iStockphoto.

- All objects of the `Window` class share the same behavior.

Constructing Objects

Objects of the `Rectangle` class describe rectangular shapes.

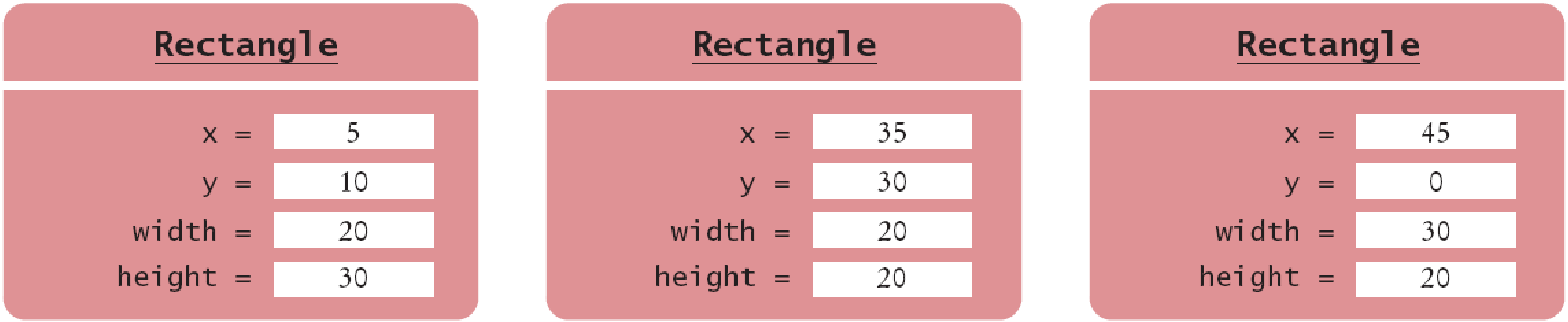
The `Rectangle` class belongs to the package `java.awt`

Constructing Objects

- The `Rectangle` object is not a rectangular shape.
- It is an object that contains a set of numbers.
  - The numbers describe the rectangle
- Each rectangle is described by:
  - The *x*- and *y*-coordinates of its top-left corner
  - Its width
  - And its height.

Constructing Objects

- In the computer, a `Rectangle` object is a block of memory that holds four numbers.



Rectangle

x = 5
y = 10
width = 20
height = 30

Rectangle

x = 35
y = 30
width = 20
height = 20

Rectangle

x = 45
y = 0
width = 30
height = 20

Figure 5 Rectangle Objects

Constructing Objects

- Use the `new` operator, followed by a class name and arguments, to construct new objects.

  ```
  new Rectangle(5, 10, 20, 30)
  ```

- Detail:
  - The `new` operator makes a `Rectangle` object
  - It uses the parameters (in this case, 5, 10, 20, and 30) to initialize the data of the object
  - It returns the object
- The process of creating a new object is called construction.
- The four values 5, 10, 20, and 30 are called the construction arguments.

Constructing Objects

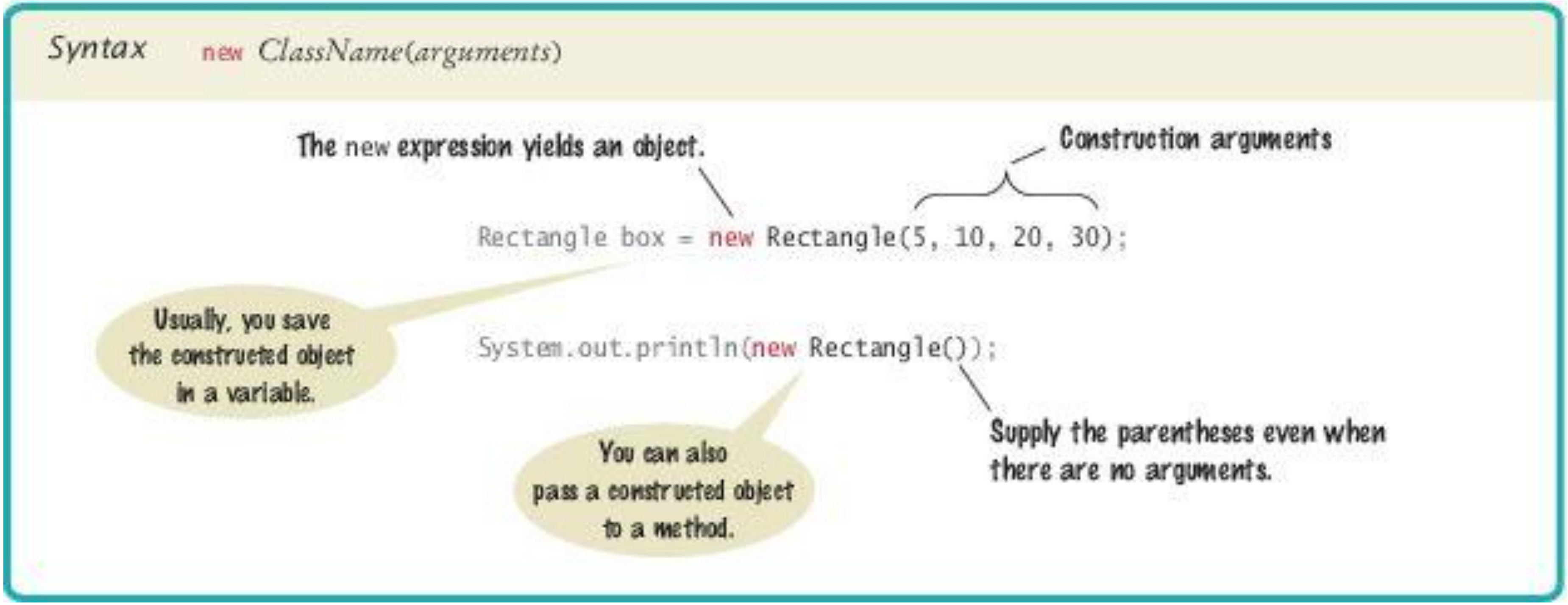
- Usually the output of the new operator is stored in a variable:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

- Additional constructor:

```
new Rectangle()
```

Syntax 2.3 Object Construction



Syntax new *ClassName(arguments)*

The new expression yields an object.

Construction arguments

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Usually, you save the constructed object in a variable.

```
System.out.println(new Rectangle());
```

You can also pass a constructed object to a method.

Supply the parentheses even when there are no arguments.

Accessor and Mutator Methods

- **Accessor method:** does not change the internal data of the object on which it is invoked.
  - Returns information about the object
  - Example: `length` method of the `String` class
  - Example: double `width = box.getWidth()`;
- **Mutator method:** changes the data of the object

  `box.translate(15, 25)`;
  - The top-left corner is now at (20, 35).

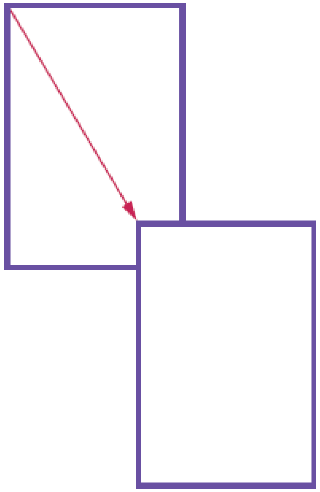


Figure 6 Using the `translate` Method to Move a Rectangle

Instance Variables and Encapsulation

© Jasmin Awad/iStockphoto.

Tally counter

- Simulator statements:

```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // Sets result to 2
```

- Each counter needs to store a variable that keeps track of the number of simulated button clicks.

Instance Variables

- Instance variables store the data of an object.
- Instance of a class: an object of the class.
- An instance variable is a storage location present in each object of the class.
- The class declaration specifies the instance variables:

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

- An object's instance variables store the data required for executing its methods.

Instance Variables

- An instance variable declaration consists of the following parts:
  - access specifier (`private`)
  - type of variable (such as `int`)
  - name of variable (such as `value`)
- You should declare all instance variables as `private`.

Instance Variables

- Each object of a class has its own set of instance variables.

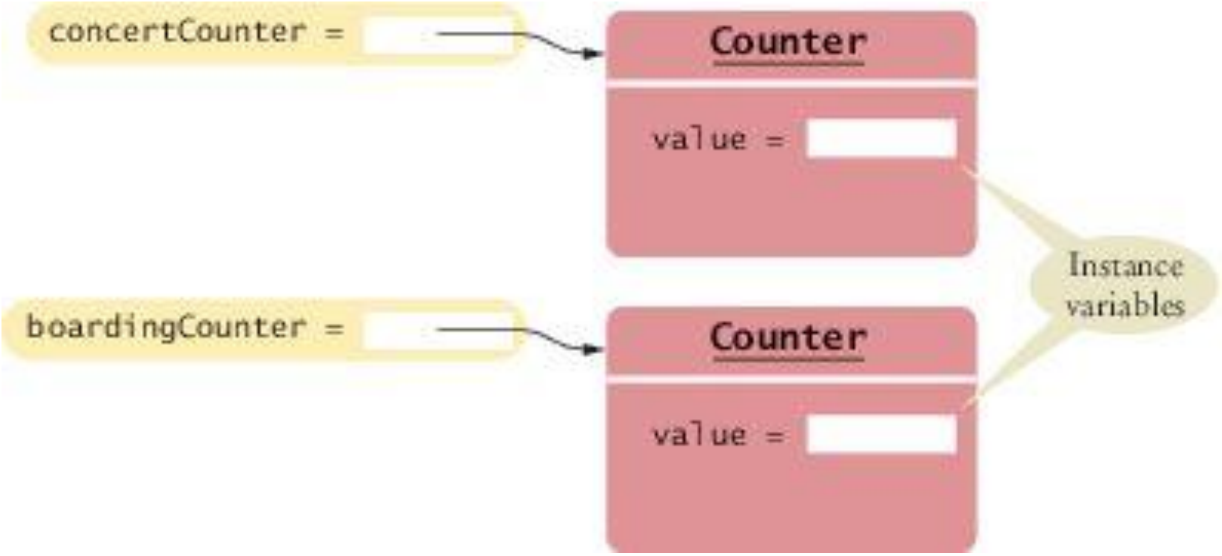
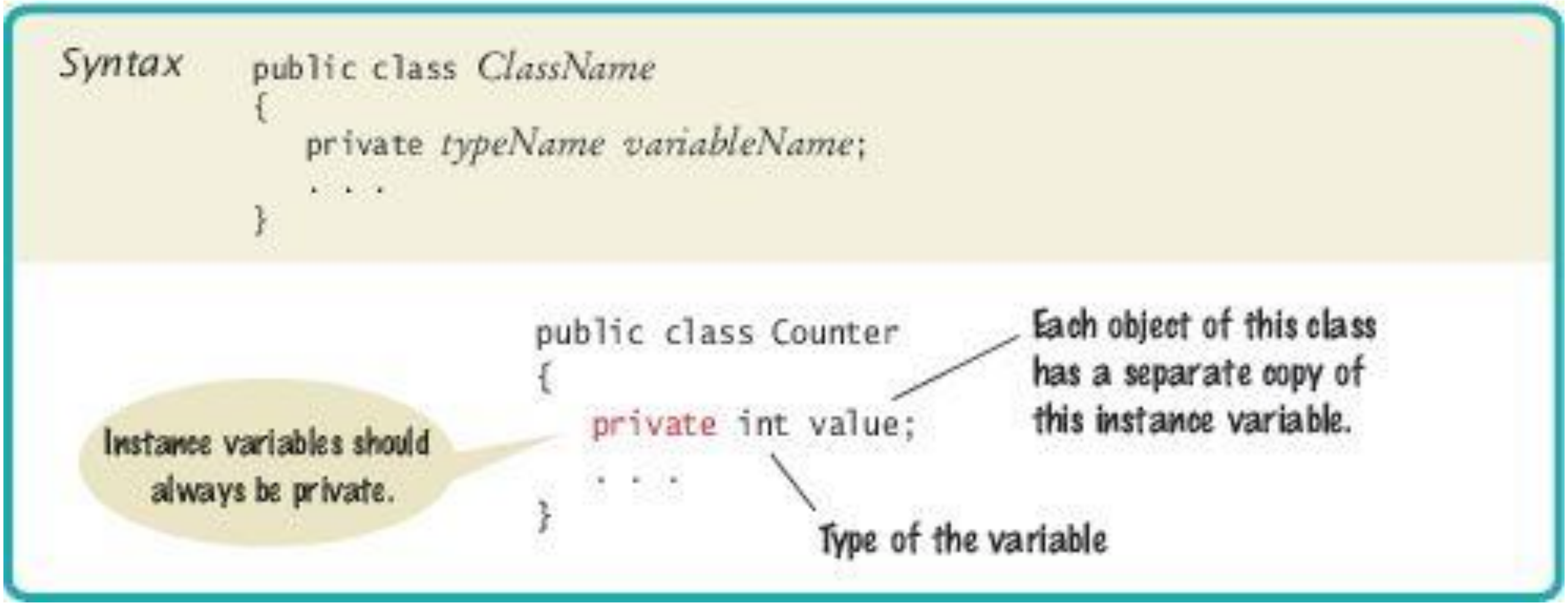


Figure 10 Instance Variables

Syntax 2.5 Instance Variable Declaration



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

The Methods of the Counter Class

- The click method advances the counter value by 1:

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

  - Affects the value of the instance variable of the object on which the method is invoked
  - The method call concertCounter.click();
    - Advances the value variable of the concertCounter object

The Methods of the Counter Class

- The getValue method returns the current value:

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

- The return statement
  - Terminates the method call
  - Returns a result (the return value) to the method's caller
- Private instance variables can only be accessed by methods of the same class.

Encapsulation

- Encapsulation is the process of hiding implementation details and providing methods for data access.
- To encapsulate data:
  - Declare instance variables as private and
  - Declare public methods that access the variables
- Encapsulation allows a programmer to use a class without having to know its implementation.
- Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.

section_5/Counter.java

```
/**
   This class models a tally counter.
*/
public class Counter
{
   private int value;

   /**
      Gets the current value of this counter.
      @return the current value
   */
   public int getValue()
   {
      return value;
   }

```

Continued

section_5/Counter.java

```java
   /**
      Advances the value of this counter by 1.
   */
   public void click()
   {
      value = value + 1;
   }

   /**
      Resets the value of this counter to 0.
   */
   public void reset()
   {
      value = 0;
   }
}
```

Specifying the Public Interface of a Class

- In order to implement a class, you first need to know which methods are required.
- Essential behavior of a bank account:
  - deposit money
  - withdraw money
  - get balance

Specifying the Public Interface of a Class

- We want to support method calls such as the following:

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

- Here are the method headers needed for a `BankAccount` class:

```
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

Specifying the Public Interface of a Class: Method Declaration

- A method's *body* consisting of statements that are executed when the method is called:

```
public void deposit(double amount)
{
   implementation - filled in later
}
```

- You can fill in the method body so it compiles:

```
public double getBalance()
{
   // TODO: fill in implementation
   return 0;
}
```

Specifying the Public Interface of a Class

- `BankAccount` methods were declared as `public`.
- `public` methods can be called by all other methods in the program.
- Methods can also be declared `private`
  - `private` methods only be called by other methods in the same class
  - `private` methods are not part of the public interface

Specifying Constructors

- Initialize objects
- Set the initial data for objects
- Similar to a method with two differences:
  - The name of the constructor is always the same as the name of the class
  - Constructors have no return type

Specifying Constructors: BankAccount

- Two constructors

```
public BankAccount()
public BankAccount(double initialBalance)
```

- Usage

```
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

Specifying Constructors: BankAccount

- The constructor name is always the same as the class name.
- The compiler can tell them apart because they take different arguments.
- A constructor that takes no arguments is called a no-argument constructor.
- BankAccount's no-argument constructor - header and body:

```
public BankAccount()
{
    constructor body—implementation filled in later
}
```

- The statements in the constructor body will set the instance variables of the object.

BankAccount Public Interface

- The constructors and methods of a class go inside the class declaration:

```
public class BankAccount
{
   // private instance variables--filled in later
   // Constructors
   public BankAccount()
   {
      // body--filled in later
   }
   public BankAccount(double initialBalance)
   {
      // body--filled in later
   }
```

Continued

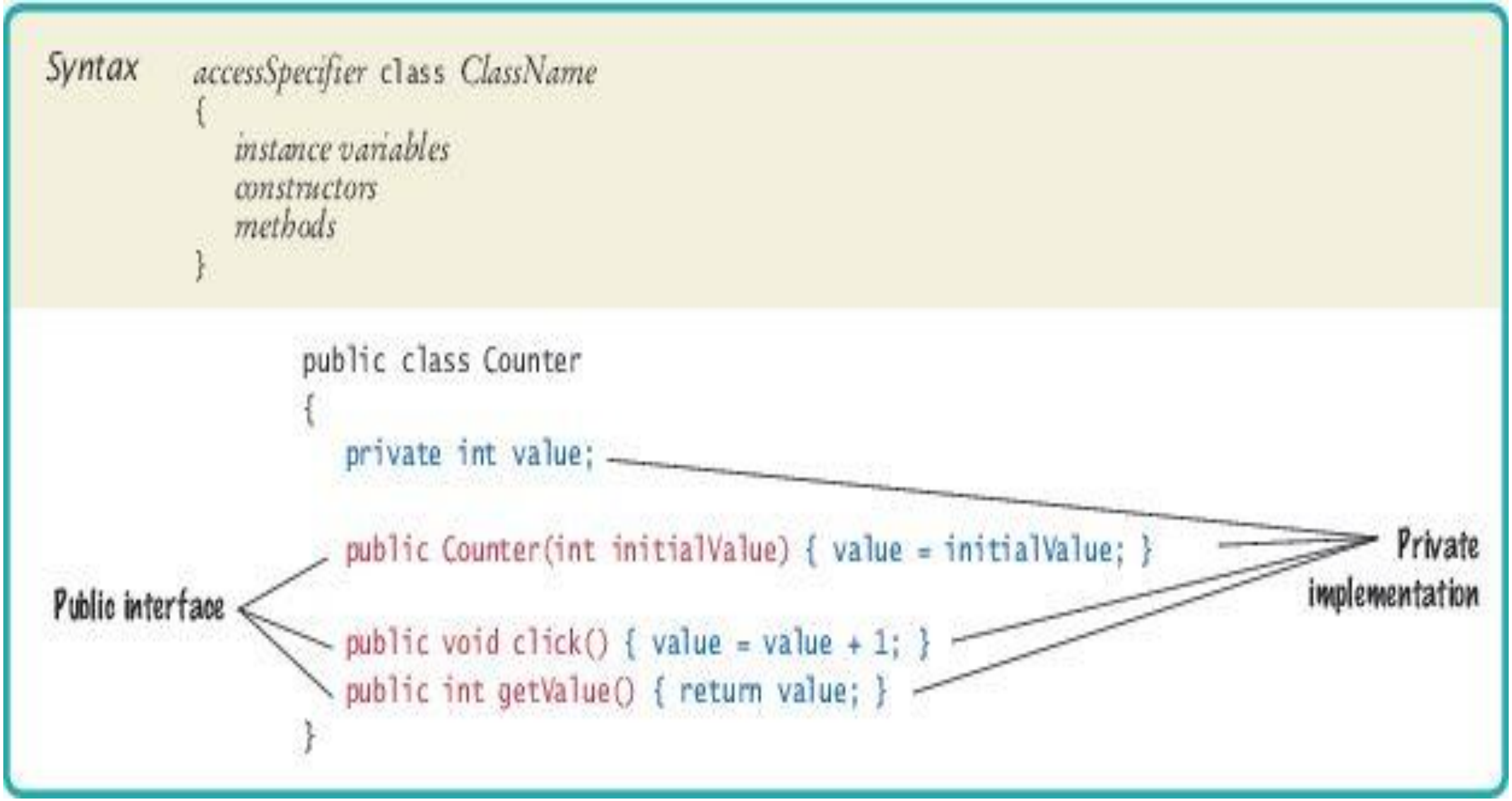
BankAccount Public Interface

```
   // Methods
   public void deposit(double amount)
   {
      // body--filled in later
   }
   public void withdraw(double amount)
   {
      // body--filled in later
   }
   public double getBalance()
   {
      // body--filled in later
   }
}
```

Specifying the Public Interface of a Class

- `public` constructors and methods of a class form the **public interface** of the class.
- These are the operations that any programmer can use.

Syntax 2.6 Class Declaration



Syntax

```
accessSpecifier class ClassName
{
   instance variables
   constructors
   methods
}
```

```
public class Counter
{
   private int value;

   public Counter(int initialValue) { value = initialValue; }

   public void click() { value = value + 1; }
   public int getValue() { return value; }
}
```

Public interface
Private implementation

Using the Public Interface

- Example: transfer money

```
// Transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

- Example: add interest

```
double interestRate = 5; // 5 percent interest
double interestAmount =
   momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

- Programmers use objects of the BankAccount class to carry out meaningful tasks
  - without knowing how the BankAccount objects store their data
  - without knowing how the BankAccount methods do their work

Commenting the Public Interface – Documenting a Method

- Start the comment with a `/**`.
- Describe the method's purpose.
- Describe each parameter:
  - start with `@param`
  - name of the parameter that holds the argument
  - a short explanation of the argument
- Describe the return value:
  - start with `@return`
  - describe the return value
- Omit `@param` tag for methods that have no arguments.
- Omit the `@return` tag for methods whose return type is void.
- End with `*/`.

Commenting the Public Interface – Documenting a Method

- Example:

```
/** Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   implementation—filled in later
}
```

Commenting the Public Interface – Documenting a Method

- Example:

```
/** Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   implementation—filled in later
}
```

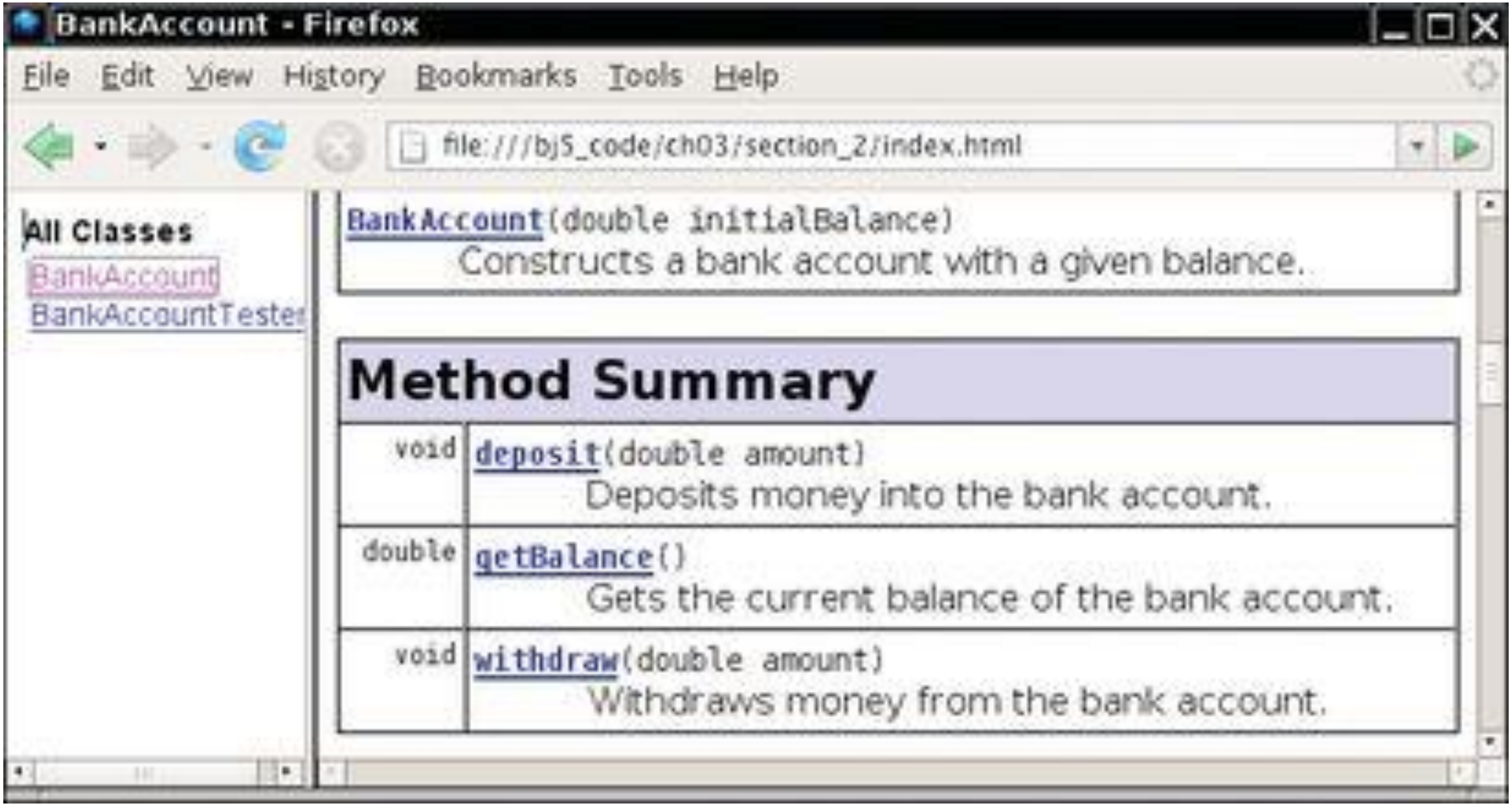
Commenting the Public Interface – Documenting a Class

- Place above the class declaration.
- Supply a brief comment explaining the class's purpose.
- Example:

```
/** A bank account has a balance that can be changed by
    deposits and withdrawals.
*/
public class BankAccount
 { . . . }
```

- Provide documentation comments for:
  - every class
  - every method
  - every parameter variable
  - every return value

Method Summary



All Classes

BankAccount

BankAccountTester

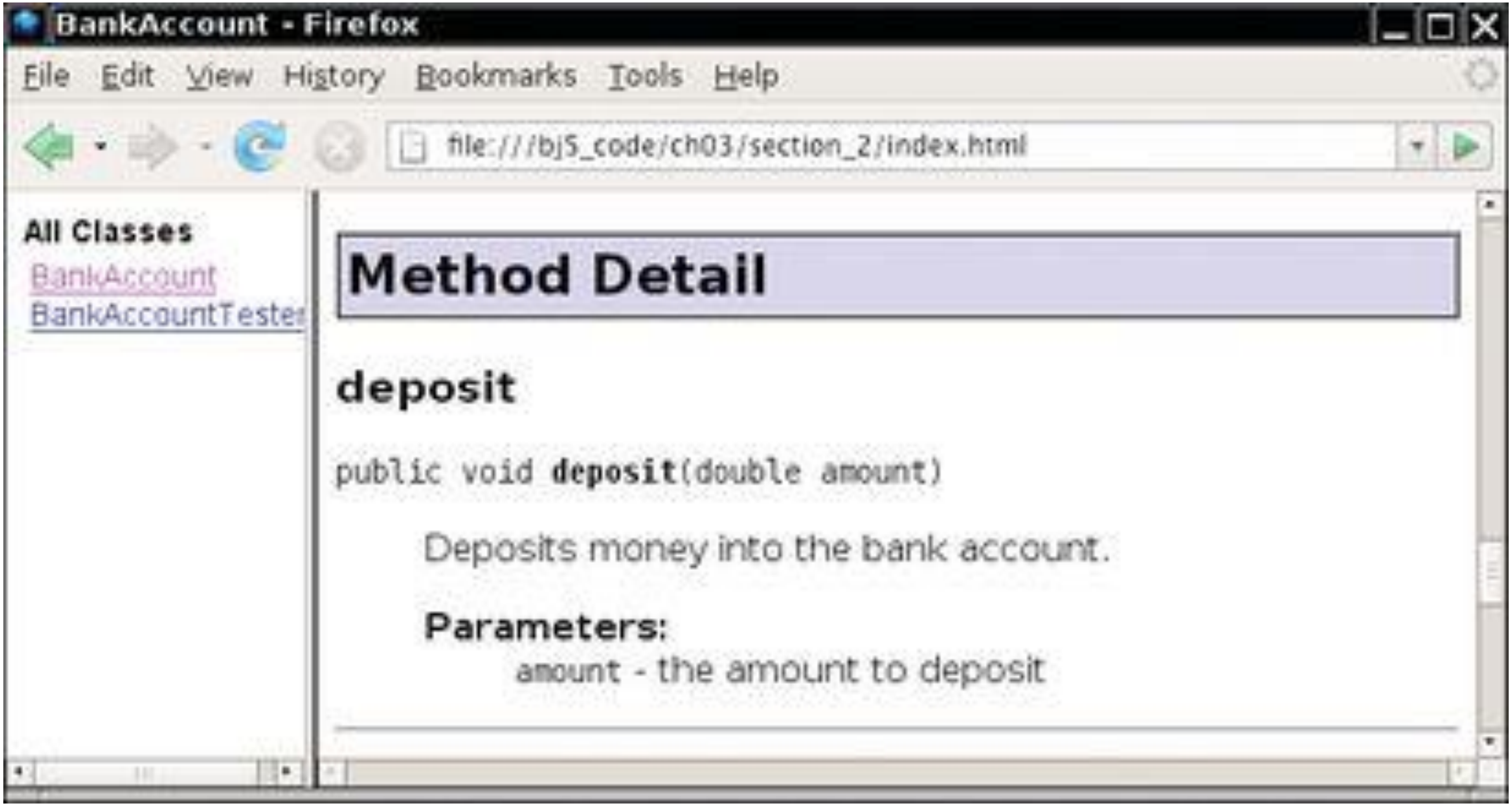
BankAccount(double initialBalance)
Constructs a bank account with a given balance.

Method Summary

void	deposit(double amount) Deposits money into the bank account.
double	getBalance() Gets the current balance of the bank account.
void	withdraw(double amount) Withdraws money from the bank account.

Figure 11 A Method Summary Generated by `javadoc`

Method Details


BankAccount - Firefox
File Edit View History Bookmarks Tools Help
file:///bj5_code/ch03/section_2/index.html
All Classes
BankAccount
BankAccountTester

Method Detail

deposit

```
public void deposit(double amount)
```

Deposits money into the bank account.

Parameters:
amount - the amount to deposit

Figure 12 Method Detail Generated by `javadoc`

Self Check 2.29

How can you use the methods of the public interface to *empty* the `harrysChecking` bank account?

Answer:

```
harrysChecking.withdraw(harrysChecking.getBalance())
```

Self Check 2.30

What is wrong with this sequence of statements?

```
BankAccount harrysChecking = new BankAccount(10000);
System.out.println(harrysChecking.withdraw(500));
```

Answer: The `withdraw` method has return type `void`. It doesn’t return a value. Use the `getBalance` method to obtain the balance after the withdrawal.

Self Check 2.31

Suppose you want a more powerful bank account abstraction that keeps track of an *account number* in addition to the balance. How would you change the public interface to accommodate this enhancement?

Answer: Add an `accountNumber` parameter to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method – the account number never changes after construction.

Self Check 2.32

Suppose we enhance the BankAccount class so that each account has an account number. Supply a documentation comment for the constructor

```
public BankAccount(int accountNumber, double initialBalance)
```

Answer:

```
/**
   Constructs a new bank account with a given initial balance.
   @param accountNumber the account number for this account
   @param initialBalance the initial balance for this account
*/
```

Providing the Class Implementation

- The implementation of a class consists of:
  - instance variables
  - the bodies of constructors
  - the bodies of methods.

Providing Instance Variables

- Determine the data that each bank account object contains.
- What does the object need to remember so that it can carry out its methods?
- Each bank account object only needs to store the current balance.
- `BankAccount` instance variable declaration:

```
public class BankAccount
{
   private double balance;
   // Methods and constructors below
   . . .
}
```

Providing Constructors

- Constructor's job is to initialize the instance variables of the object.
- The no-argument constructor sets the balance to zero.

```
public BankAccount()
{
   balance = 0;
}
```

- The second constructor sets the balance to the value supplied as the construction argument.

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

Providing Constructors - Tracing the Statement

Steps carried out when the following statement is executed:

```
BankAccount harrysChecking = new BankAccount(1000);
```

- Create a new object of type `BankAccount`. ❶
- Call the second constructor
  - because an argument is supplied in the constructor call
- Set the parameter variable `initialBalance` to `1000`. ❷
- Set the `balance` instance variable of the newly created object to `initialBalance`. ❸
- Return an object reference, that is, the memory location of the object.
- Store that object reference in the `harrysChecking` variable. ❹

Providing Constructors - Tracing the Statement

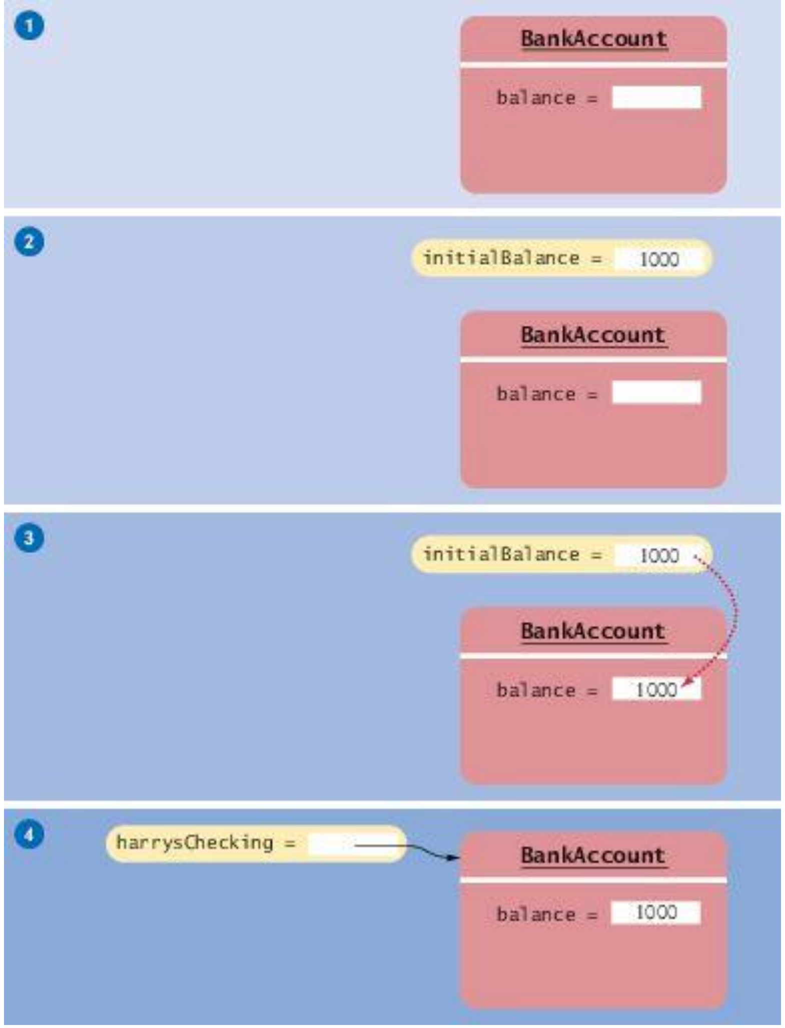


Figure 13 How a Constructor Works

Providing Methods

- Is the method an accessor or a mutator
  - Mutator method
    - Update the instance variables in some way
  - Accessor method
    - Retrieves or computes a result
- deposit method - a mutator method
  - Updates the balance

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

Providing Methods

- `withdraw` method - another mutator

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

- `getBalance` method - an accessor method
  - Returns a value

```
public double getBalance()
{
   return balance;
}
```

Table 3 Implementing Classes

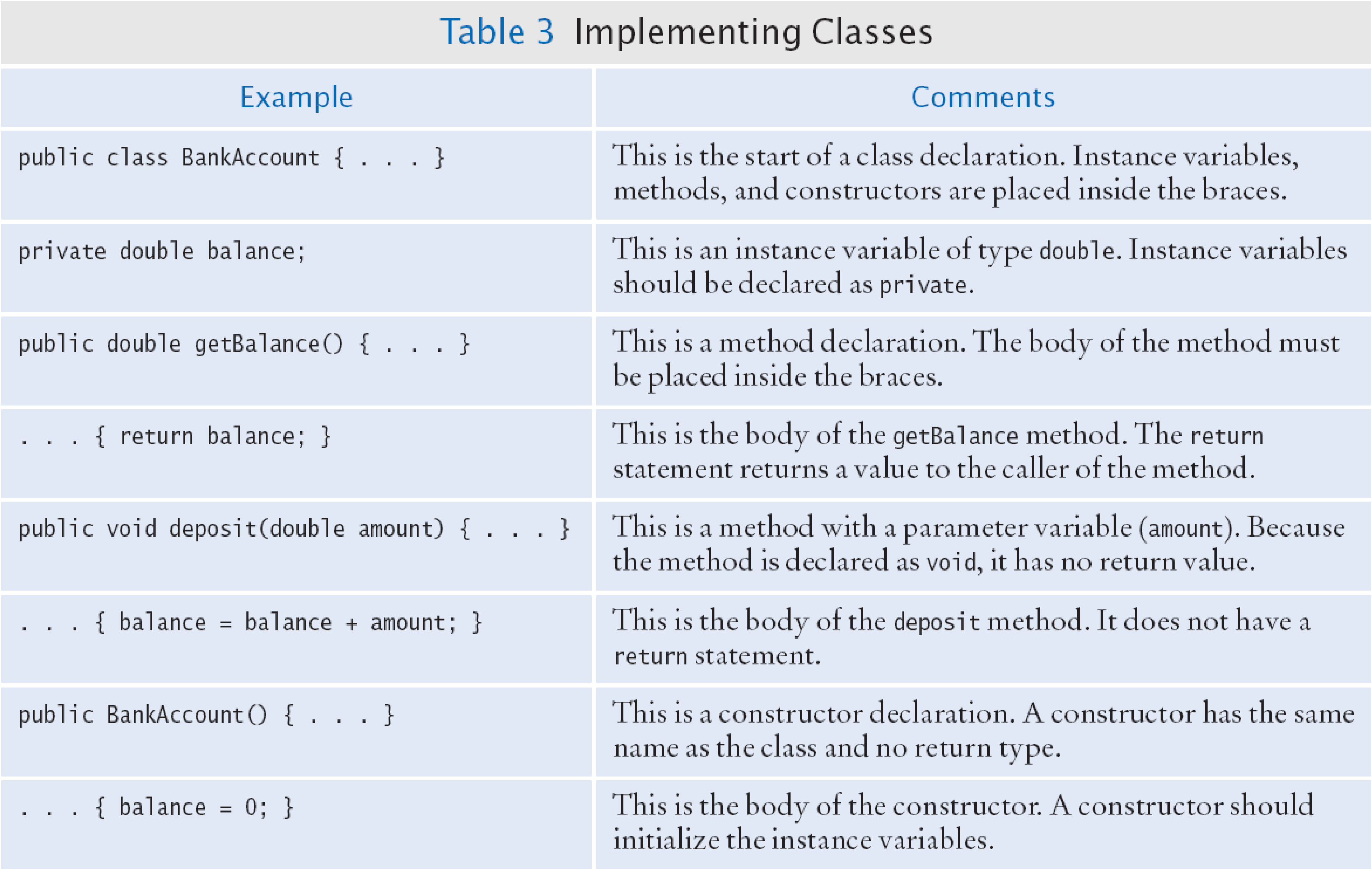


Table 3 Implementing Classes

Example	Comments
`public class BankAccount { . . . }`	This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.
`private double balance;`	This is an instance variable of type `double`. Instance variables should be declared as `private`.
`public double getBalance() { . . . }`	This is a method declaration. The body of the method must be placed inside the braces.
`. . . { return balance; }`	This is the body of the `getBalance` method. The `return` statement returns a value to the caller of the method.
`public void deposit(double amount) { . . . }`	This is a method with a parameter variable (`amount`). Because the method is declared as `void`, it has no return value.
`. . . { balance = balance + amount; }`	This is the body of the `deposit` method. It does not have a `return` statement.
`public BankAccount() { . . . }`	This is a constructor declaration. A constructor has the same name as the class and no return type.
`. . . { balance = 0; }`	This is the body of the constructor. A constructor should initialize the instance variables.

section_7/BankAccount.java

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private double balance;

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
   }

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

```

Continued

section_7/BankAccount.java

```java
   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      balance = balance - amount;
   }

   /**
      Gets the current balance of the bank account.
      @return the current balance
   */
   public double getBalance()
   {
      return balance;
   }
}
```

Self Check 2.34

Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance variables?

Answer: An instance variable needs to be added to the class:

```
private int accountNumber;
```

Self Check 2.35

Why does the following code not succeed in robbing mom's bank account?

```
public class BankRobber
{
   public static void main(String[] args)
   {
      BankAccount momsSavings = new BankAccount(1000);
      momsSavings.balance = 0;
   }
}
```

Answer: Because the `balance` instance variable is accessed from the `main` method of `BankRobber`. The compiler will report an error because `main` has no access to `BankAccount` instance variables.

Self Check 2.36

The `Rectangle` class has four instance variables: `x`, `y`, `width`, and `height`. Give a possible implementation of the `getWidth` method.

Answer:

```
public int getWidth()
{
   return width;
}
```

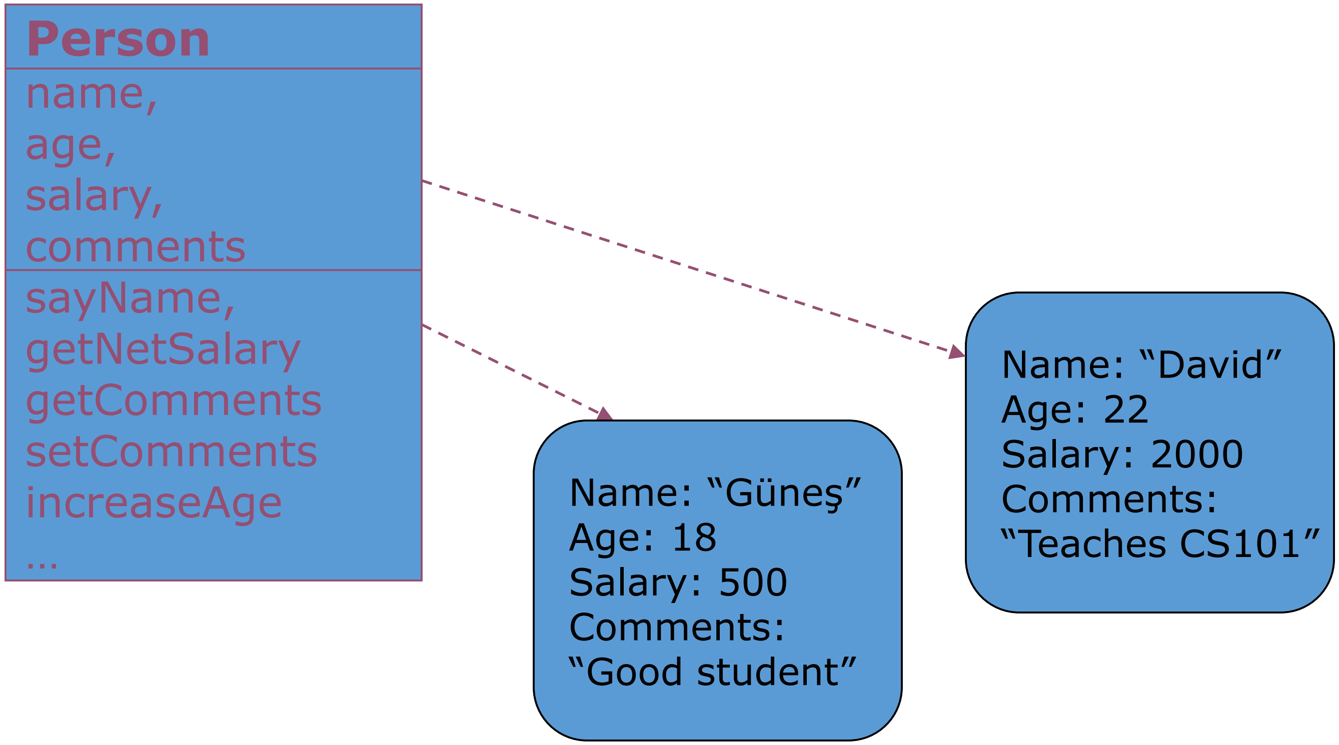
Self Check 2.37

Give a possible implementation of the `translate` method of the `Rectangle` class.

Answer: There is more than one correct answer. One possible implementation is as follows:

```
public void translate(int dx, int dy)
{
    int newx = x + dx;
    x = newx;
    int newy = y + dy;
    y = newy;
}
```

Another Example: Create & manipulate person objects



Person

name,
age,
salary,
comments

sayName,
getNetSalary
getComments
setComments
increaseAge
…

Name: “Güneş”
Age: 18
Salary: 500
Comments:
“Good student”

Name: “David”
Age: 22
Salary: 2000
Comments:
“Teaches CS101”

Tasks

- Write the Person class

- In the main method of another “test” class:
  - Create k person objects
  - Store the created person objects in an array
  - Print the contents of all objects in the array in the following format:

```
Person 1
Name: xxx
Age: xxx
Salary: xxx
Comments: xxx

Person 2
…
```

Hint:

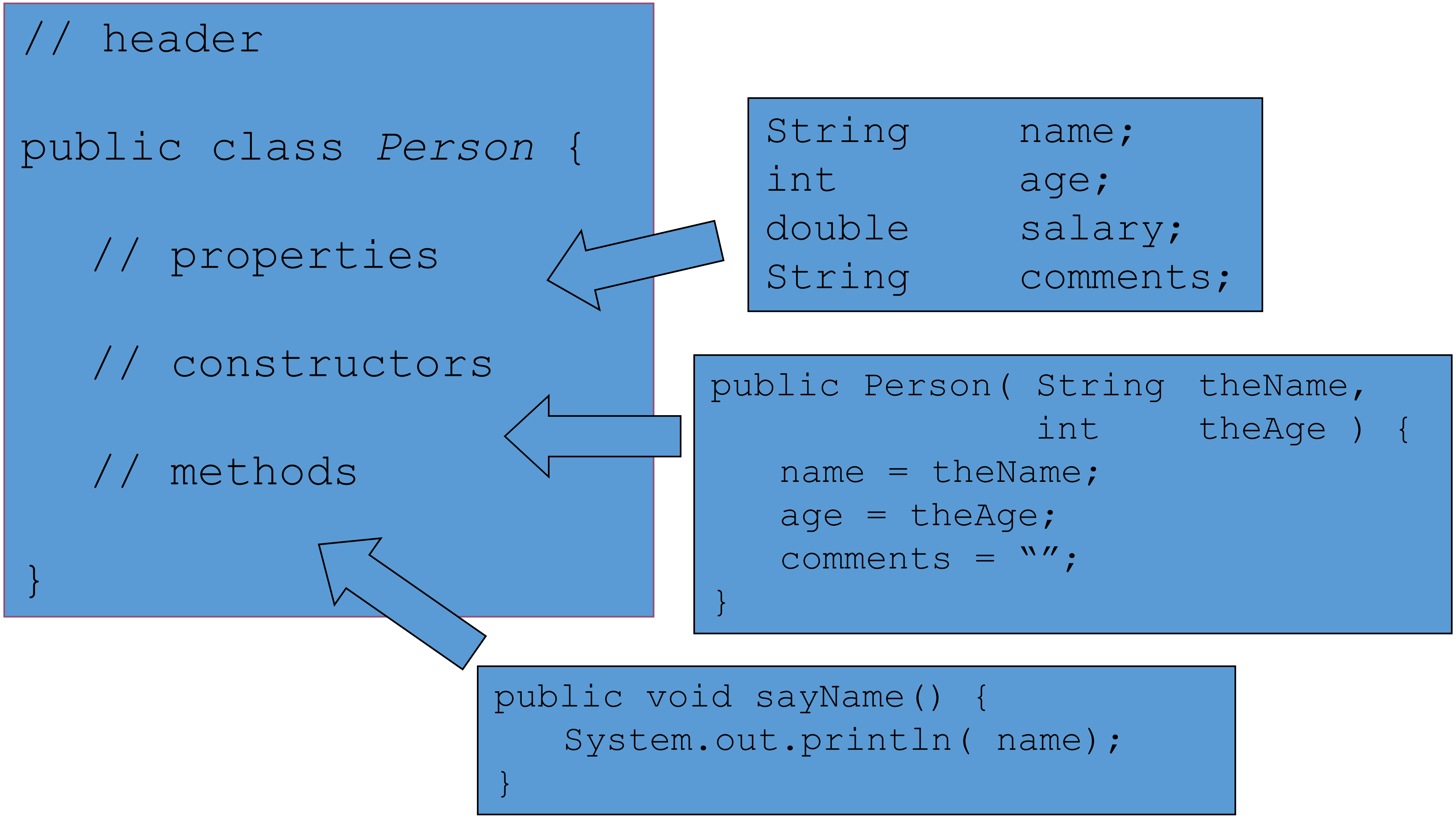
```
public String toString()
{
            return …;
}
```

Person

name,
age,
salary,
comments

getName,
getNetSalary
getComments
setComments
increaseAge
…

Coding Java Classes



```
// header

public class Person {

   // properties

   // constructors

   // methods

}
```

```
String      name;
int         age;
double      salary;
String      comments;
```

```
public Person( String  theName,
               int     theAge ) {
   name = theName;
   age = theAge;
   comments = “”;
}
```

```
public void sayName() {
   System.out.println( name);
}
```

Coding Java Classes

```
public String getName() {
    return name;
}
```

```
public String getComments() {
    return comments;
}
```

```
public void setComments( String someText) {
    comments = someText;
}
```

“get” & “set” methods for some properties (no setName!)

```
public void increaseAge() {
    age = age + 1;
}
```

```
public double getNetSalary( int baseRate) {
    double tax;
    tax = compoundInterest( baseRate);
    return salary – tax * 1.10;
}
```

Variables which are not parameters or properties must be defined locally.

Simplified Person Class

```
package myworld;
// Person - simple example only!
// Author: David, CS101

public class Person {

    // properties
    String name;
    int    age;

    // constructors
    public Person( String theName, int
theAge) {
        name = theName;
        age = theAge;
    }

    // methods
    public void increaseAge() {
        age = age + 1;
    }

    public void sayNameAndAge() {
        System.out.println( name + "\t" +
age );
    }
}
```

Declare properties
note private/package access.

Give initial values to
each of the properties

Define (instance) methods
that examine/change properties

Simplified PersonTest

```
import myworld.Person;
// PersonTest - demo Person class
// Author: David, CS101

public class PersonTest {
    public static void main( String[] args) {
        // VARIABLES
        Person  aStudent;
        Person  friend;

        // PROGRAM CODE
        aStudent = new Person( "Güneş", 18);
        friend = new Person( "David", 22);

        aStudent.sayNameAndAge();
        friend.sayNameAndAge();

        friend.increaseAge();
        aStudent.increaseAge();
        friend.increaseAge();

        System.out.println();
        aStudent.sayNameAndAge();
        friend.sayNameAndAge();
    }
} // end of class PersonTest
```

Declare variables
to hold Person objects

Create Person objects
& put them into the variables

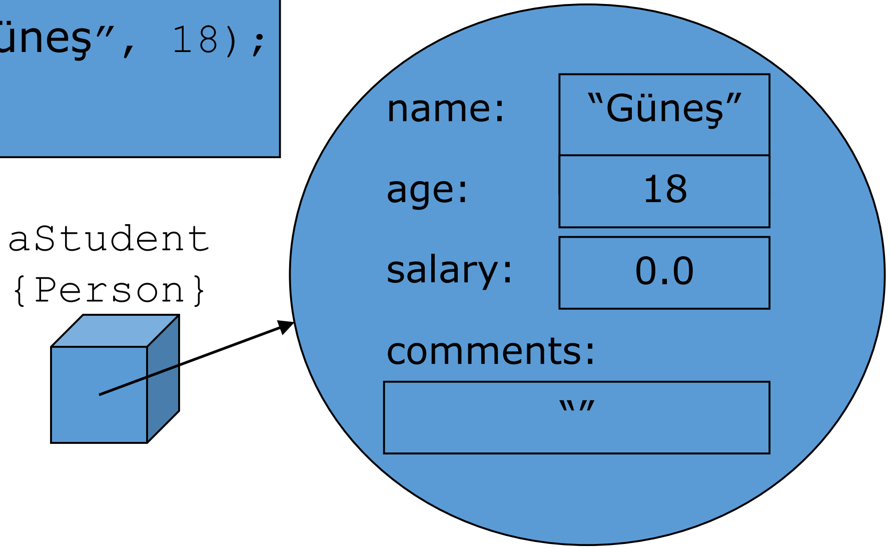
Use objects by calling
their methods

Creating & Using Objects

- Always
  - Declare variable to “hold” object
  - Create object using “new” statement
  - Call object’s methods

```
Person aStudent;
aStudent = new Person( “Güneş”, 18);
aStudent.sayName();
```

Put this in method of another class, (e.g main method)



aStudent
{Person}

name: “Güneş”

age: 18

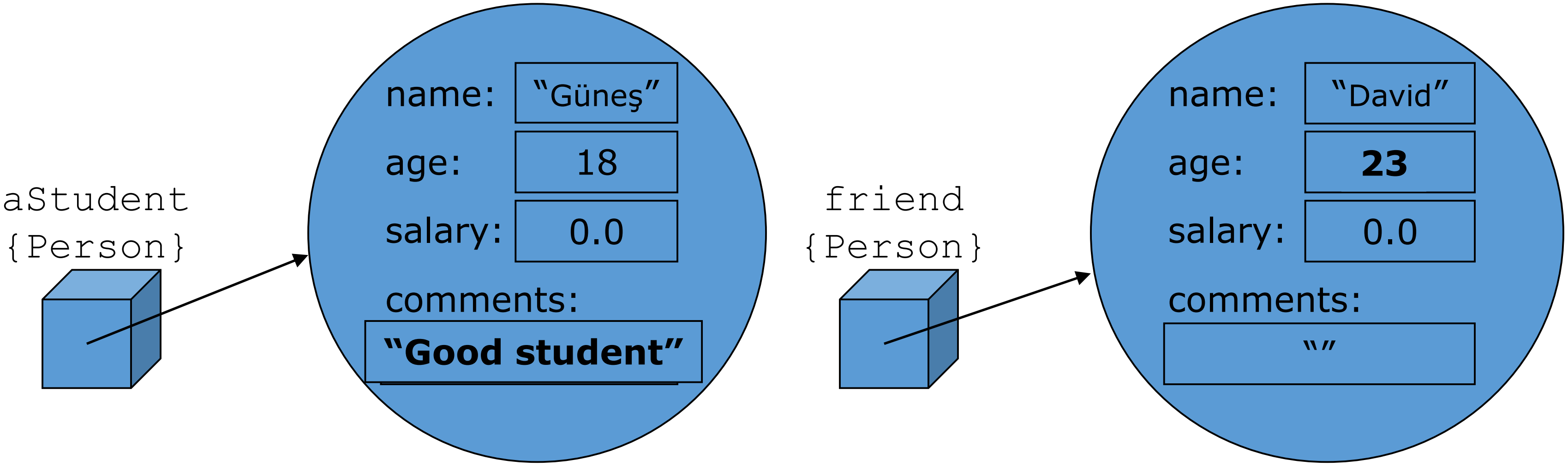
salary: 0.0

comments: “”

Creating & Using Objects

```
Person aStudent;
aStudent = new Person( "Güneş", 18);
```

```
Person friend;
friend = new Person( "David", 22);
```



aStudent
{Person}

name: "Güneş"
age: 18
salary: 0.0
comments:
"Good student"

friend
{Person}

name: "David"
age: **23**
salary: 0.0
comments:
""

```
friend.increaseAge();
aStudent.setComments( "Good student");
```

Other Examples: existing classes

- Random class

```
Random die;
die = new Random();
int face = die.nextInt(6) + 1;
System.out.println( face);
```

- StringTokenizer class

```
StringTokenizer tokens;
tokens = new StringTokenizer( “to be or not to be”);
while ( tokens.hasMoreTokens() ) {
   aWord = tokens.nextToken();
   System.out.println( aWord);
}
System.out.println( “done”);
```

Writing Your Own Classes

- Coins
- Dice
- Traffic lights
- TV
- Video
- Wallet
- Music CD
- Time/Date (in various formats!)