CHAPTER 8

OBJECTS AND CLASSES

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
10/30/2011

Chapter Goals

- To understand the concepts of classes, objects and encapsulation
- To implement instance variables, methods and constructors
- To be able to design, implement, and test your own classes
- To understand the behavior of object references, static variables and static methods

In this chapter, you will learn how to discover, specify, and implement your own classes, and how to use them in your programs.

Contents

- Object-Oriented Programming
- Implementing a Simple Class
- Specifying the Public Interface of a Class
- Designing the Data Representation
- Implementing Instance Methods
- Constructors
- Static Variables and Methods

8.1 Object-Oriented Programming

- You have learned structured programming
  - Breaking tasks into subtasks
  - Writing re-usable methods to handle tasks
- We will now study Objects and Classes
  - To build larger and more complex programs
  - To model objects we use in the world

A class describes objects with the same behavior. For example, a Car class describes all passenger vehicles that have a certain capacity and shape.

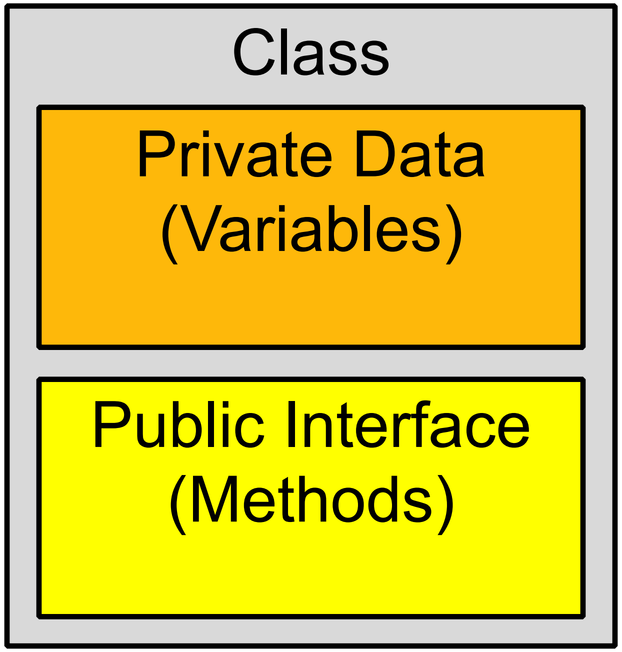
Objects and Programs

- Java programs are made of objects that interact with each other
  - Each object is based on a class
  - A class describes a set of objects with the same behavior
- Each class defines a specific set of methods to use with its objects
  - For example, the `String` class provides methods:
    - Examples: `length()` and `charAt()` methods

```
String greeting = “Hello World”;
int len = greeting.length();
char c1 = greeting.charAt(0);
```

Diagram of a Class

- Private Data
  - Each object has its own private data that other objects cannot directly access
  - Methods of the public interface provide access to private data, while hiding implementation details:
  - This is called Encapsulation
- Public Interface
  - Each object has a set of methods available for other objects to use
  - E.g., Java API

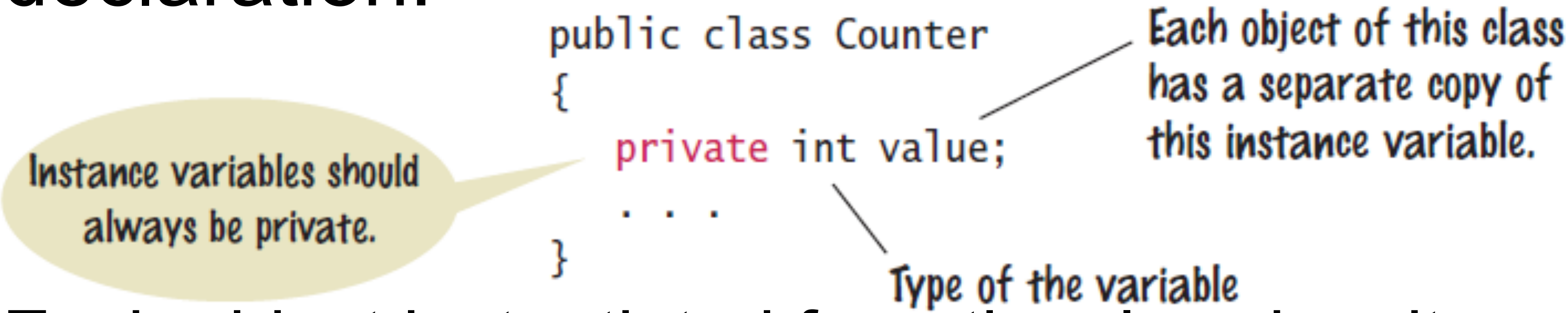


8.2 Implementing a Simple Class

- Example: Tally Counter: A class that models a mechanical device that is used to count people
  - For example, to find out how many people attend a concert or board a bus
- What should it do?
  - Increment the tally
  - Get the current total

Tally Counter Class

- Specify instance variables in the class declaration:



```
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

- Each object instantiated from the class has its own set of instance variables
  - Each tally counter has its own current count
- Access Specifiers:
  - Classes (and interface methods) are `public`
  - Instance variables are always `private`

Instantiating Objects

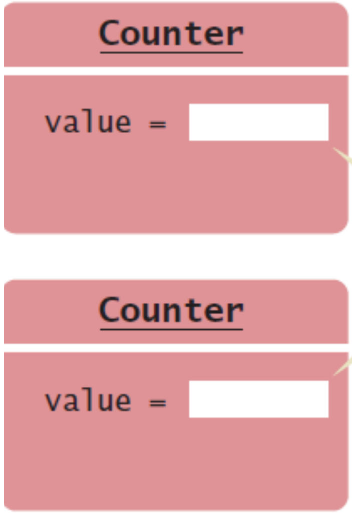
- Objects are created based on classes
  - Use the new operator to construct objects
  - Give each object a unique name (like variables)
- You have used the new operator before:

```
Scanner in = new Scanner(System.in);
```

- Creating two instances of Counter objects:

Class name · Object name · Class name

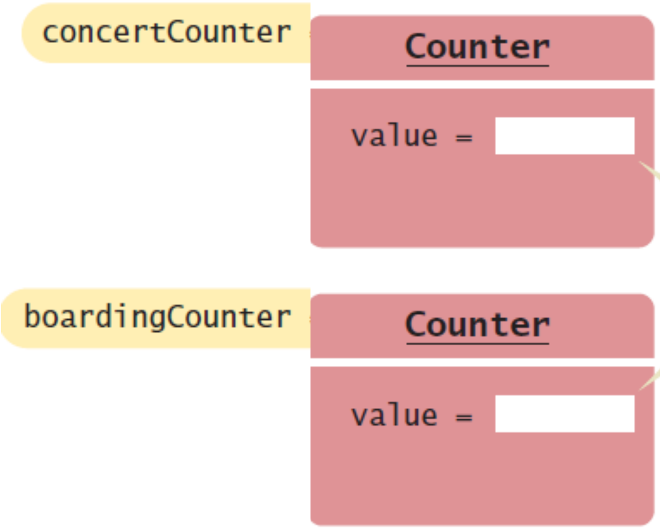
```
Counter concertCounter  = new Counter();
Counter boardingCounter = new Counter();
```



Use the new operator to construct objects of a class.

Tally Counter Methods

- Design a method named count that adds 1 to the instance variable
- Which instance variable?
  - Use the name of the object
    - concertCounter.count()
    - boardingCounter.count()



```
public class Counter
{
   private int value;

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

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

8.3 Public Interface of a Class

- When you design a class, start by specifying the public interface of the new class
  - Example: A Cash Register Class
    - What tasks will this class perform?
    - What methods will you need?
    - What parameters will the methods need to receive?
    - What will the methods return?

Task	Method	Returns
Add the price of an item	`addItem(double)`	***void***
Get the total amount owed	`getTotal()`	***double***
Get the count of items purchased	`getCount()`	***int***
Clear the cash register for a new sale	`clear()`	***void***

Writing the Public Interface

```
/**
  A simulated cash register that tracks the item count
  and the total amount due.
*/
public class CashRegister
{
  /**
    Adds an item to this cash register.
    @param price: the price of this item
  */
  public void addItem(double price)
  {
    // Method body
  }
  /**
    Gets the price of all items in the current sale.
    @return the total price
  */
  public double getTotal()  ...
```

Javadoc style comments document the class and the behavior of each method

The method declarations make up the *public interface* of the class

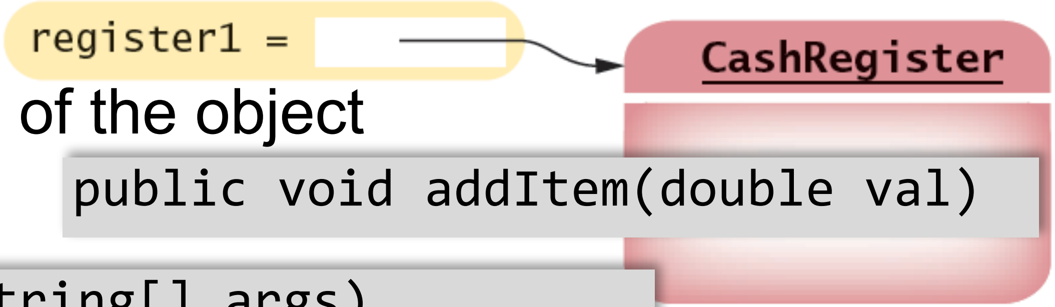
The data and method bodies make up the *private implementation* of the class

Non-static Methods Means…

- We have been writing *class* methods using the `static` modifier:

```
public static void addItem(double val)
```

- For non-static (*instance*) methods, you must instantiate an object of the class before you can invoke methods



  - Then invoke methods of the object

```
public void addItem(double val)
```

```
public static void main(String[] args)
{
  // Construct a CashRegister object
  CashRegister register1 = new CashRegister();
  // Invoke a non-static method of the object
  register1.addItem(1.95);
}
```

Accessor and Mutator Methods

- Many methods fall into two categories:

1) Accessor Methods: '**get**' methods

- Asks the object for information without changing it
- Normally return a value of some type

```
public double getTotal() {  }
public int getCount() {  }
```

2) Mutator Methods: '**set**' methods

- Changes values in the object
- Usually take a parameter that will change an instance variable
- Normally return void

```
public void addItem(double price) {  }
public void clear() {  }
```

8.4 Designing the Data Representation

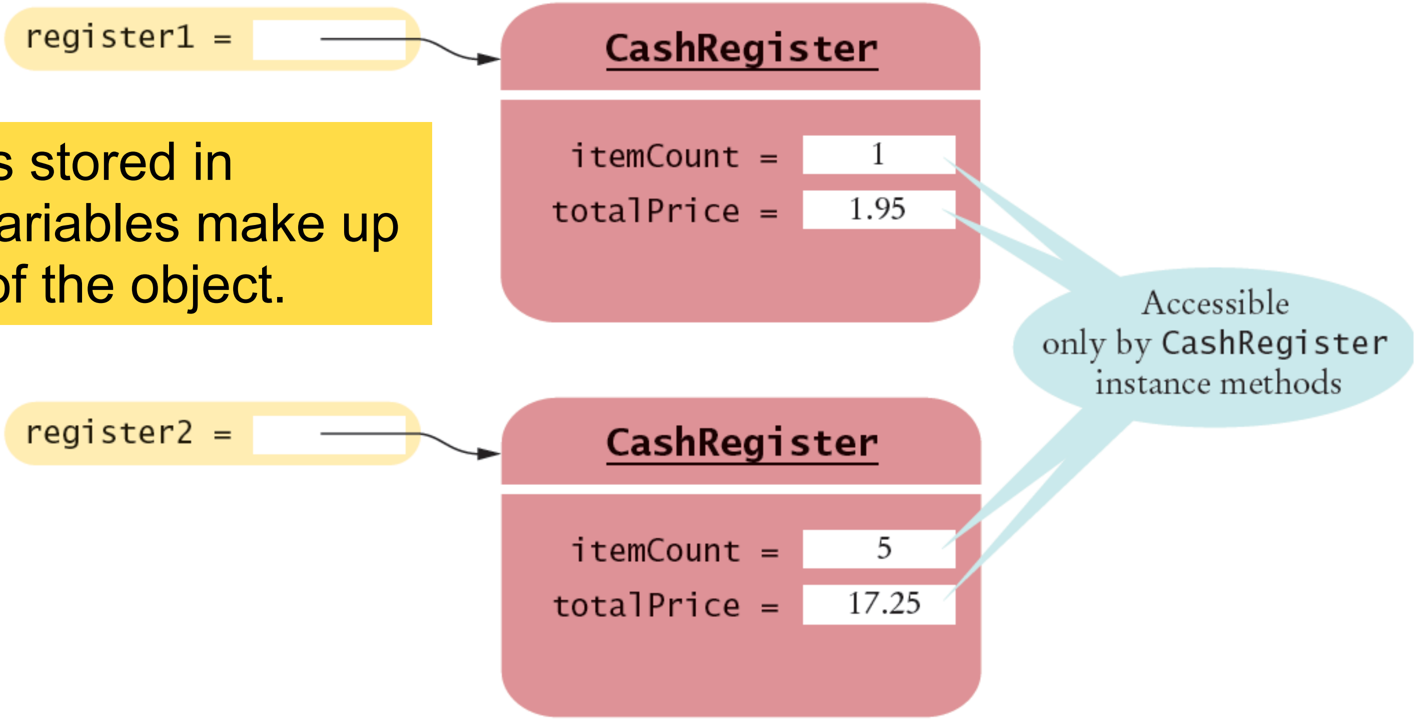
- An object stores data in instance variables
  - Variables declared inside the class
  - All methods inside the class have access to them
    - Can change or access them
  - What data will our `CashRegister` methods need?

Task	Method	Data Needed
Add the price of an item	`addItem()`	`total, count`
Get the total amount owed	`getTotal()`	`total`
Get the count of items purchased	`getCount()`	`count`
Clear the cash register for a new sale	`clear()`	`total, count`

An object holds instance variables that are accessed by methods

Instance Variables of Objects

- Each object of a class has a separate set of instance variables.



The values stored in instance variables make up the **state** of the object.

Accessing Instance Variables

- private instance variables cannot be accessed from methods outside of the class

```
public static void main(String[] args)
{
   . . .
   System.out.println(register1.itemCount); // Error
   . . .
}
```

The compiler will not allow this violation of privacy

- Use accessor methods of the class instead!

```
public static void main(String[] args)
{
   . . .
   System.out.println( register1.getCount() ); // OK
   . . .
}
```

Encapsulation provides a public interface and hides the implementation details.

8.5 Implementing Instance Methods

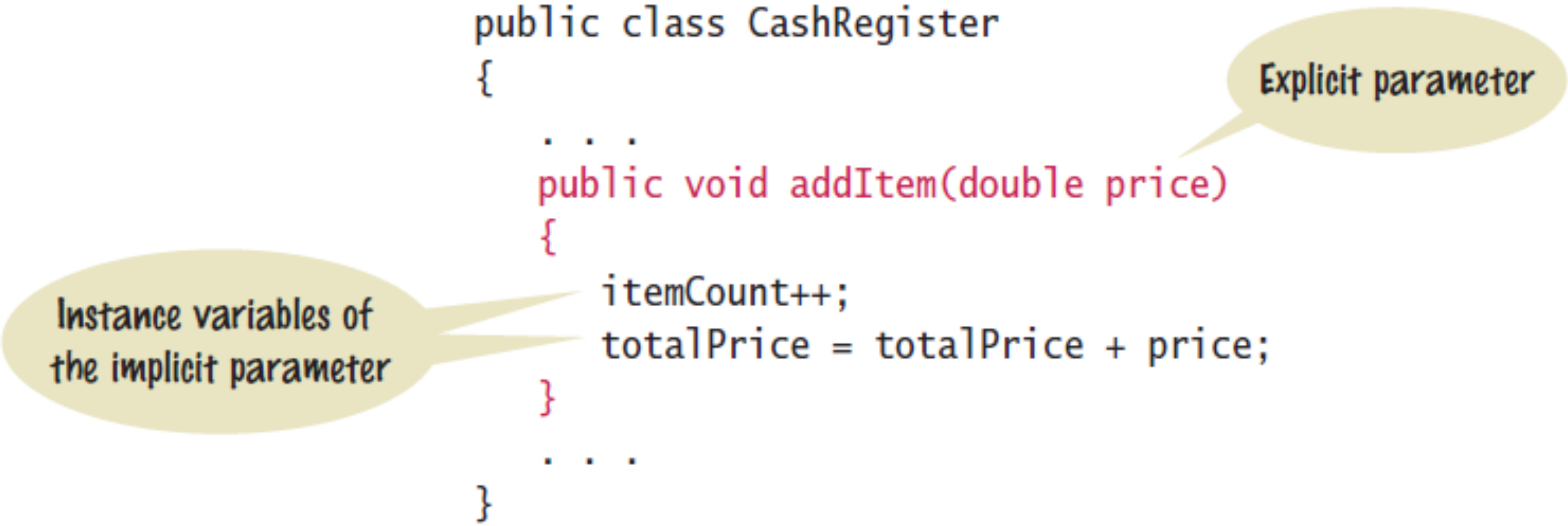
- Implement instance methods that will use the private instance variables

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

Task	Method	Returns
Add the price of an item	*addItem(double)*	***void***
Get the total amount owed	*getTotal()*	***double***
Get the count of items purchased	*getCount()*	***int***
Clear the cash register for a new sale	*clear()*	***void***

Syntax 8.2: Instance Methods

- Use instance variables inside methods of the class
  - There is no need to specify the implicit parameter (name of the object) when using instance variables inside the class
  - Explicit parameters must be listed in the method declaration



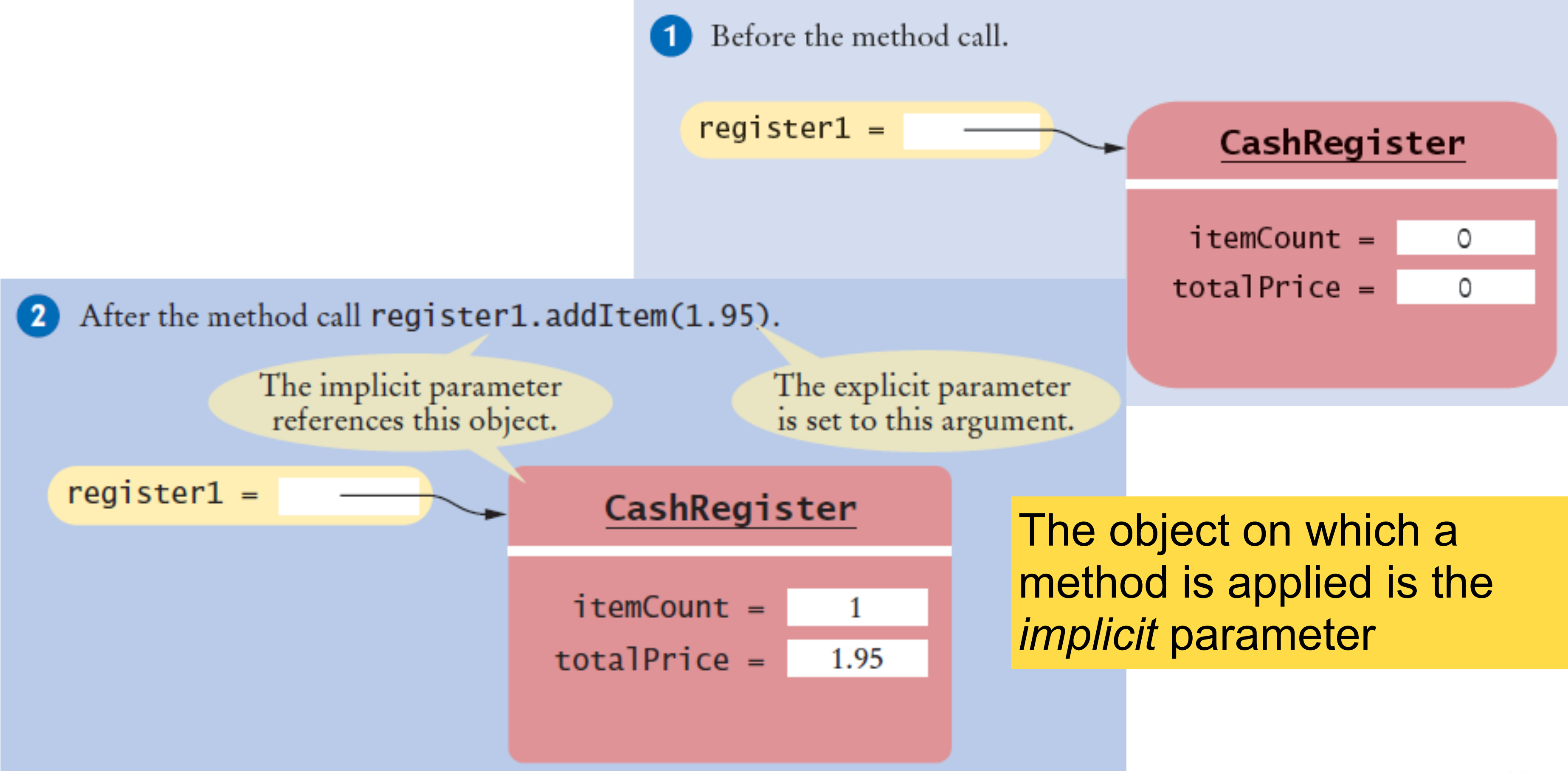
```
public class CashRegister
{
   . . .
   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }
   . . .
}
```

Explicit parameter

Instance variables of the implicit parameter

Implicit and Explicit Parameters

- When an item is added, it affects the instance variables of the object on which the method is invoked


1 Before the method call.
register1 =
CashRegister
itemCount = 0
totalPrice = 0
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

The object on which a method is applied is the *implicit* parameter

8.6 Constructors

- A *constructor* is a method that initializes instance variables of an object
  - It is automatically called when an object is created
  - It has exactly the same name as the class

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

Constructors never return values, but do not use void in their declaration

Multiple Constructors

- A class can have more than one constructor
  - Each must have a unique set of parameters

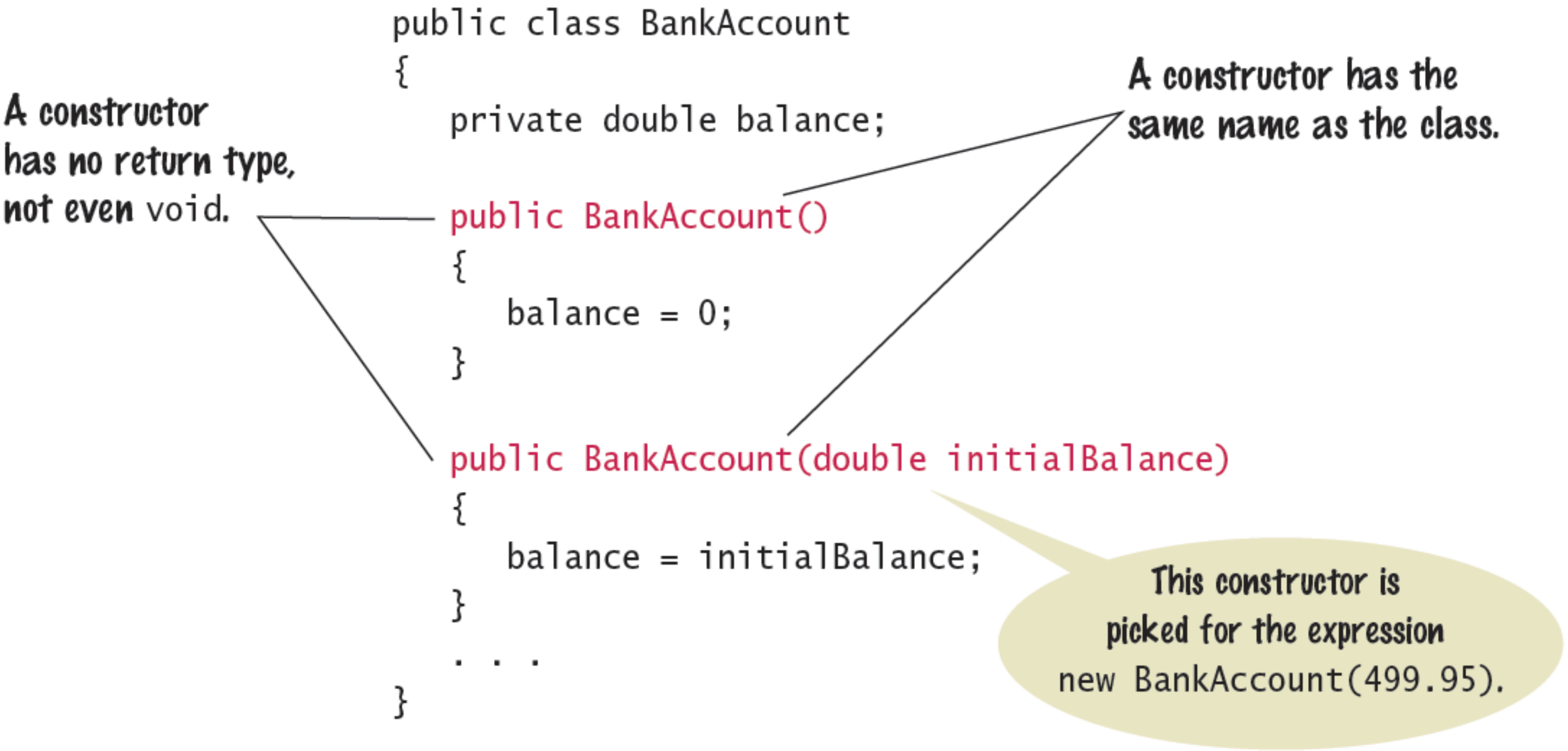
```
public class BankAccount
{
   . . .
   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount( ) { . . . }
   /**
      Constructs a bank account with a given balance.
      @param initialBalance the initial balance
   */
   public BankAccount(double initialBalance)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

The compiler picks the constructor that matches the construction parameters.

```
BankAccount joesAccount = new BankAccount();
BankAccount lisasAccount = new BankAccount(499.95);
```

Syntax 8.3: Constructors

- One constructors is invoked when the object is created with the new keyword



```
public class BankAccount
{
   private double balance;

   public BankAccount()
   {
      balance = 0;
   }

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

A constructor has no return type, not even void.

A constructor has the same name as the class.

This constructor is picked for the expression new BankAccount(499.95).

The Default Constructor

- If you do not supply any constructors, the compiler will make a default constructor automatically
  - It takes no parameters
  - It initializes all instance variables

```
public class CashRegister
{
   . . .
   /**
      Does exactly what a compiler generated constructor would do.
   */
   public CashRegister()
   {
      itemCount = 0;
      totalPrice = 0;
   }
}
```

By default, numbers are initialized to `0`, booleans to `false`, and objects as `null`.

CashRegister.java

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

   /**
      Constructs a cash register with cleared i
   */
   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;
   }
}
```

Common Error 8.2

- Trying to Call a Constructor
  - You cannot call a constructor like other methods
  - It is 'invoked' for you by the new reserved word

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

  - You cannot invoke the constructor on an existing object:

```
register1.CashRegister(); // Error
```

  - But you can create a new object using your existing reference

```
CashRegister register1 = new CashRegister();
Register1.newItem(1.95);
CashRegister register1 = new CashRegister();
```

Common Error 8.3

- Declaring a Constructor as `void`
  - Constructors have no return type
  - This creates a method with a return type of `void` which is NOT a constructor!
    - The Java compiler does not consider this an error

```
public class BankAccount
{
   /**
      Intended to be a constructor.
   */
   public void BankAccount( )
   {
      . . .
   }
}
```

Not a constructor…. Just another method that returns nothing (void)

Special Topic 8.2

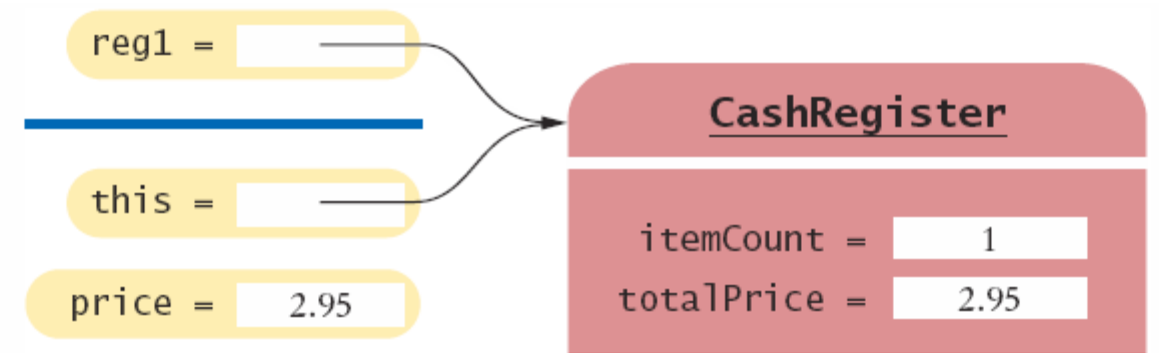
- Overloading
  - We have seen that multiple constructors can have exactly the same name
    - They require different lists of parameters
  - Actually any method can be overloaded
    - Same method name with different parameters

```
void print(CashRegister register)    { . . . }
void print(BankAccount account)      { . . . }
void print(int value)                { . . . }
Void print(double value)             { . . . }
```

  - Your book does not use overloading
    - Except as required for constructors

The this reference

- Methods receive the ‘implicit parameter’ in a reference variable called ‘this’
  - It is a reference to the object the method was invoked on:



  - It can clarify when instance variables are used:

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

Constructor this reference

- Sometimes people use the this reference in constructors
  - It makes it very clear that you are setting the instance variable:

```
public class Student
{
  private int id;
  private String name;
  public Student(int id, String name)
  {
    this.id = id;
    this.name = name;
  }
}
```

8.11 Static Variables and Methods

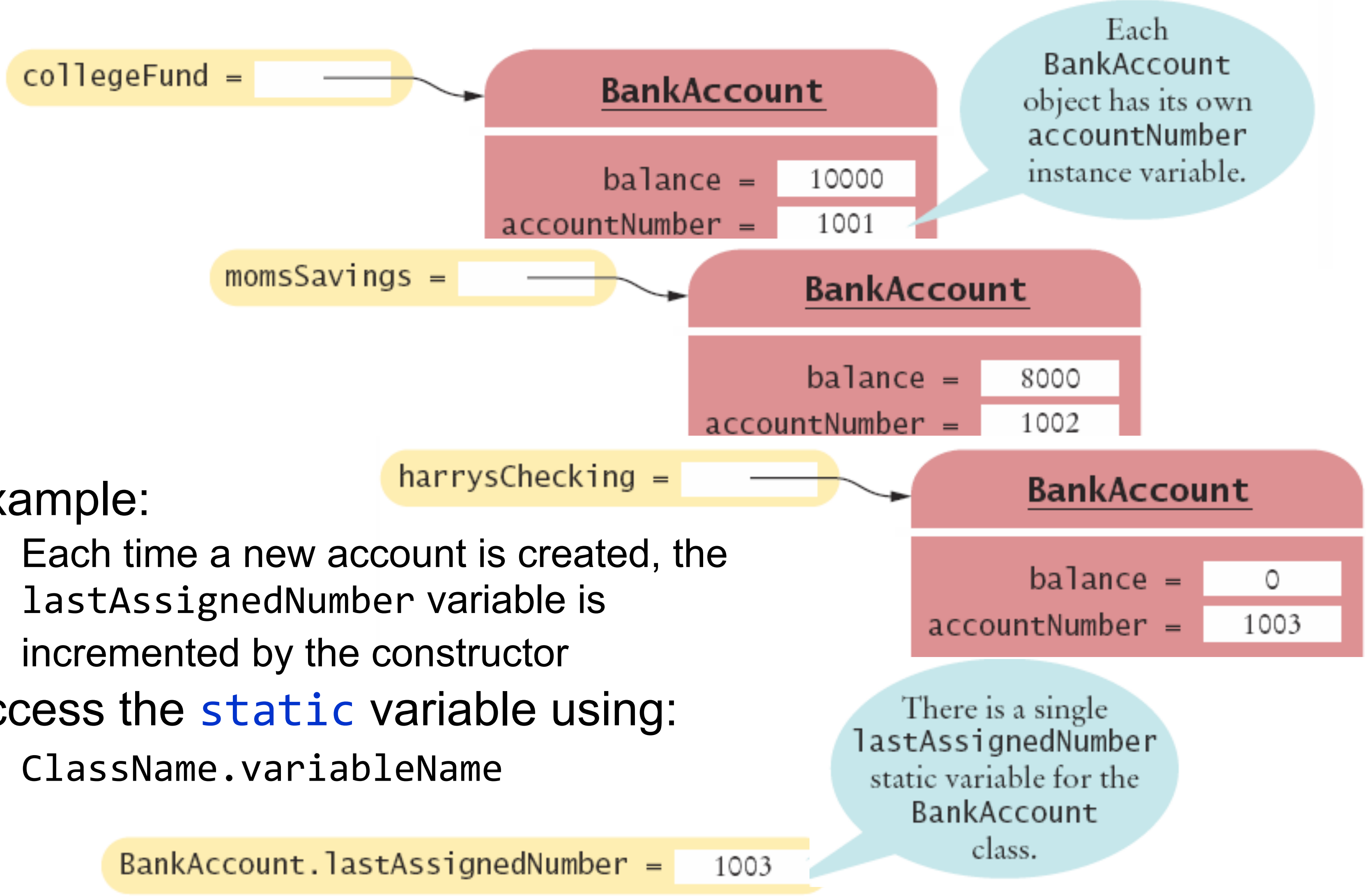
- Variables can be declared as static in the Class declaration
  - There is one copy of a static variable that is shared among all objects of the Class

```
public class BankAccount
{
  private double balance;
  private int accountNumber;
  private static int lastAssignedNumber = 1000;

  public BankAccount()
  {
    lastAssignedNumber++;
    accountNumber = lastAssignedNumber;
  }
  . . .
}
```

Methods of any object of the class can use or change the value of a static variable

Using Static Variables



collegeFund =

BankAccount

balance = 10000

accountNumber = 1001

momsSavings =

BankAccount

balance = 8000

accountNumber = 1002

harrysChecking =

BankAccount

balance = 0

accountNumber = 1003

Each BankAccount object has its own accountNumber instance variable.

There is a single lastAssignedNumber static variable for the BankAccount class.

BankAccount.lastAssignedNumber = 1003

- Example:
  - Each time a new account is created, the `lastAssignedNumber` variable is incremented by the constructor
- Access the `static` variable using:
  - `ClassName.variableName`

Using Static Methods

- The Java API has many classes that provide methods you can use without instantiating objects
  - The `Math` class is an example we have used
  - `Math.sqrt(value)` is a `static` method that returns the square root of a value
  - You do not need to instantiate the `Math` class first
- Access `static` methods using:
  - `ClassName.methodName()`