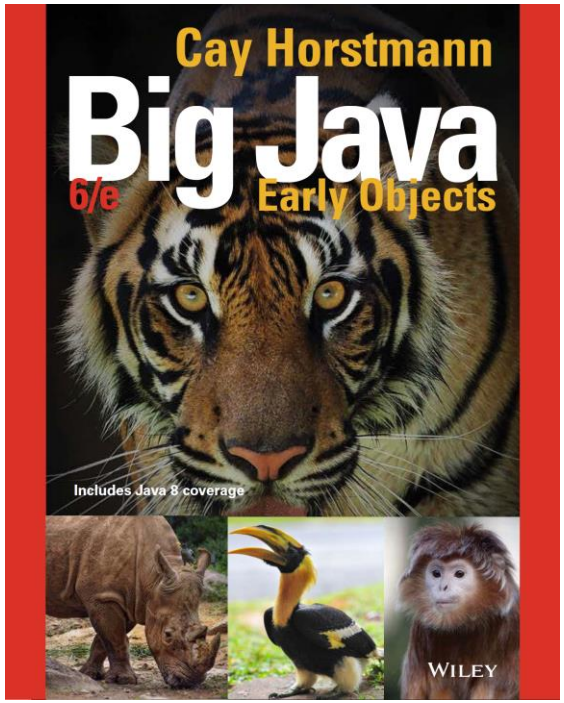


Big Java Early Objects by Cay Horstmann

Chapter 8 – Designing Classes Part I

Lecture Goals

- To learn how to discover appropriate classes for a given problem
- To understand the concepts of cohesion and coupling
- To minimize the use of side effects
- To document the responsibilities of methods and their callers with preconditions

Discovering Classes

- A class represents a single concept from the problem domain
- Name for a class should be a noun that describes concept
- Concepts from mathematics:

```
Point
Rectangle
Ellipse
```

- Concepts from real life:

```
BankAccount
CashRegister
```

Discovering Classes

- Actors (end in -er, -or) – objects do some kinds of work for you:

```
Scanner
Random // better name: RandomNumberGenerator
```

- Utility classes – no objects, only static methods and constants:

```
Math
```

- Program starters: only have a `main` method
- Don't turn actions into classes
  - *`Paycheck` is a better name than `ComputePaycheck`*

Self Check 8.1

What is the rule of thumb for finding classes?

Answer: Look for nouns in the problem description.

Self Check 8.2

Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Answer: Yes (`ChessBoard`) and no (`MovePiece`).

Cohesion

- A class should represent a single concept
- The public interface of a class is *cohesive* if all of its features are related to the concept that the class represents
- This class lacks cohesion:

```
public class CashRegister
{
   public void enterPayment(int dollars, int quarters,
      int dimes, int nickels, int pennies)
   ...
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   ...
}
```

Cohesion

- `CashRegister`, as described above, involves two concepts: *cash register* and *coin*
- Solution: Make two classes:

```
public class Coin
{
   public Coin(double aValue, String aName) { ... }
   public double getValue() { ... }
   ...
}

public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType)
      { ... }
   ...
}
```

Coupling

- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High coupling = Many class dependencies
- Minimize coupling to minimize the impact of interface changes
- To visualize relationships draw class diagrams
- UML: Unified Modeling Language
  - *Notation for object-oriented analysis and design*

Dependency

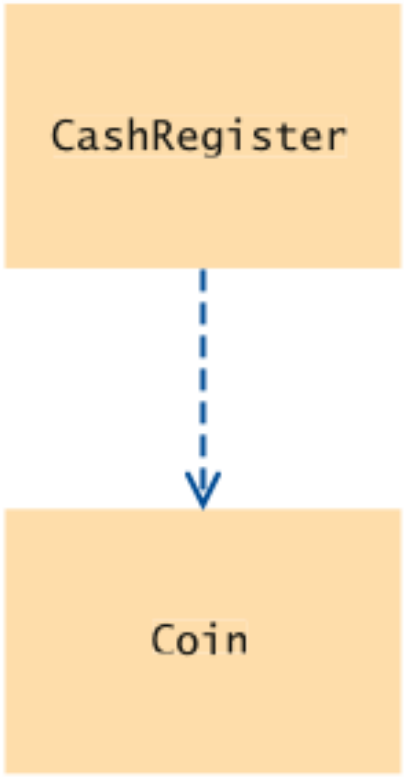


Figure 1
Dependency Relationship Between the `CashRegister` and `Coin` Classes

High and Low Coupling Between Classes

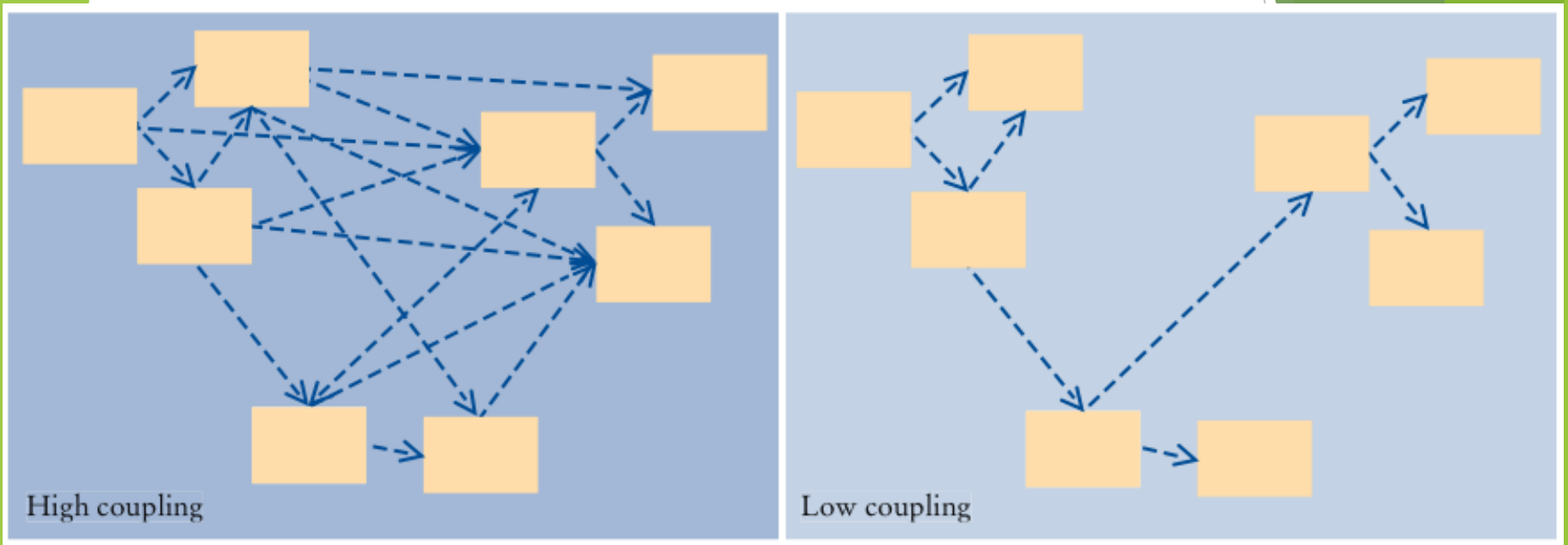


Figure 2 High and Low Coupling Between Classes

Self Check 8.3

Why is the `CashRegister` class not cohesive?

Answer: Some of its features deal with payments, others with coin values.

Self Check 8.4

Why does the `Coin` class not depend on the `CashRegister` class?

Answer: None of the `Coin` operations require the `CashRegister` class.

Self Check 8.5

Why should coupling be minimized between classes?

Answer: If a class doesn't depend on another, it is not affected by interface changes in the other class.

Immutable Classes

- **Accessor:** Does not change the state of the implicit parameter:

```
double balance = account.getBalance();
```

- **Mutator:** Modifies the object on which it is invoked:

```
account.deposit(1000);
```

- **Immutable class:** Has no mutator methods (e.g., `String`):

```
String name = "John Q. Public";
String uppercased = name.toUpperCase();
// name is not changed
```

- It is safe to give out references to objects of immutable classes; no code can modify the object at an unexpected time

Self Check 8.6

Is the `substring` method of the `String` class an accessor or a mutator?

Answer: It is an accessor — calling `substring` doesn't modify the string on which the method is invoked. In fact, all methods of the `String` class are accessors.

Side Effects

- **Side effect of a method:** Any externally observable data modification:

```
harrysChecking.deposit(1000);
```

- Modifying explicit parameter can be surprising to programmers— avoid it if possible:

```
public void addStudents(ArrayList<String> studentNames)
{
   while (studentNames.size() > 0)
   {
      String name = studentNames.remove(0);
      // Not recommended
      . . .
   }
}
```

Side Effects

- This method has the expected side effect of modifying the implicit parameter and the explicit parameter `other`:

```
public void transfer(double amount, BankAccount other
{
   balance = balance - amount;
   other.balance = other.balance + amount;
}
```

Side Effects

- Another example of a side effect is output:

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $"
      + balance);
}
```

Bad idea: Message is in English, and relies on `System.out`

- Decouple input/output from the actual work of your classes
- Minimize side effects that go beyond modification of the implicit parameter

Self Check 8.8

If `a` refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?

Answer: It is a side effect; this kind of side effect is common in object-oriented programming.

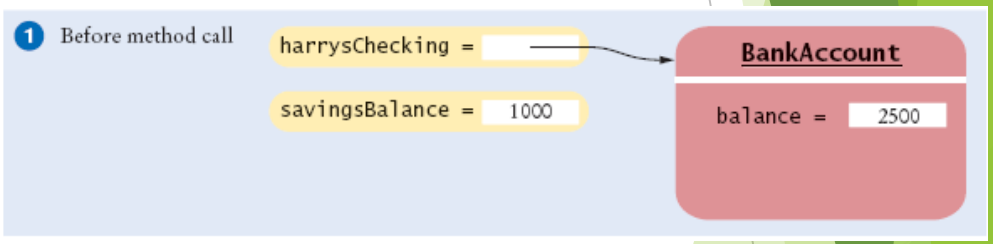
Common Error: Trying to Modify Primitive Type Parameters

- ```
  void transfer(double amount, double otherBalance)
  {
     balance = balance - amount;
     otherBalance = otherBalance + amount;
  }
  ```
- Won't work
- Scenario:

  ```
  double savingsBalance = 1000;
  harrysChecking.transfer(500, savingsBalance);
  System.out.println(savingsBalance);
  ```
- In Java, a method can never change parameters of primitive type
  ```

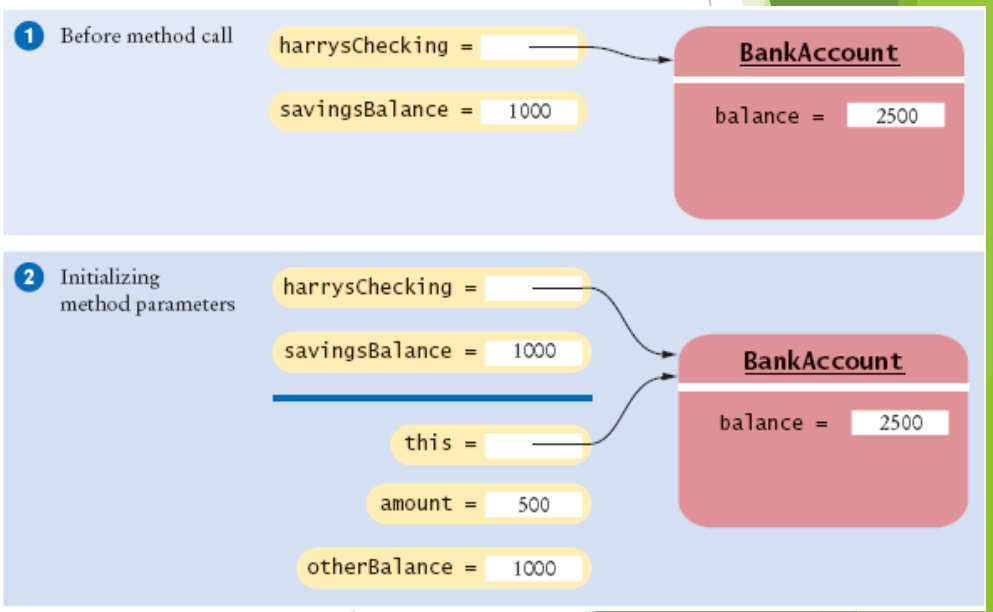
Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); 1
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}
```



Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); 1
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) 2
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}
```

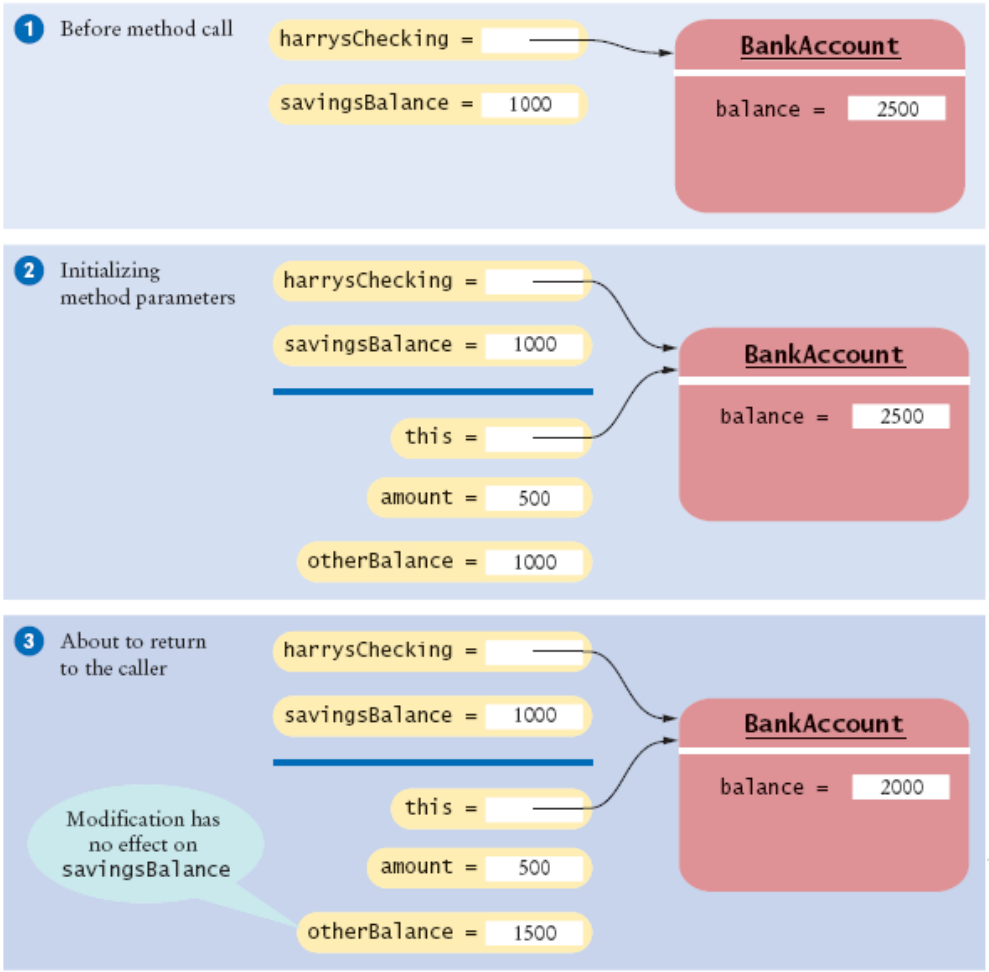


Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); (1)
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) (2)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
} (3)
```

Continued

Common Error: Trying to Modify Primitive Type Parameters



Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); 1
System.out.println(savingsBalance); 4
...
void transfer(double amount, double otherBalance) 2
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
} 3
```

Continued

Common Error: Trying to Modify Primitive Type Parameters

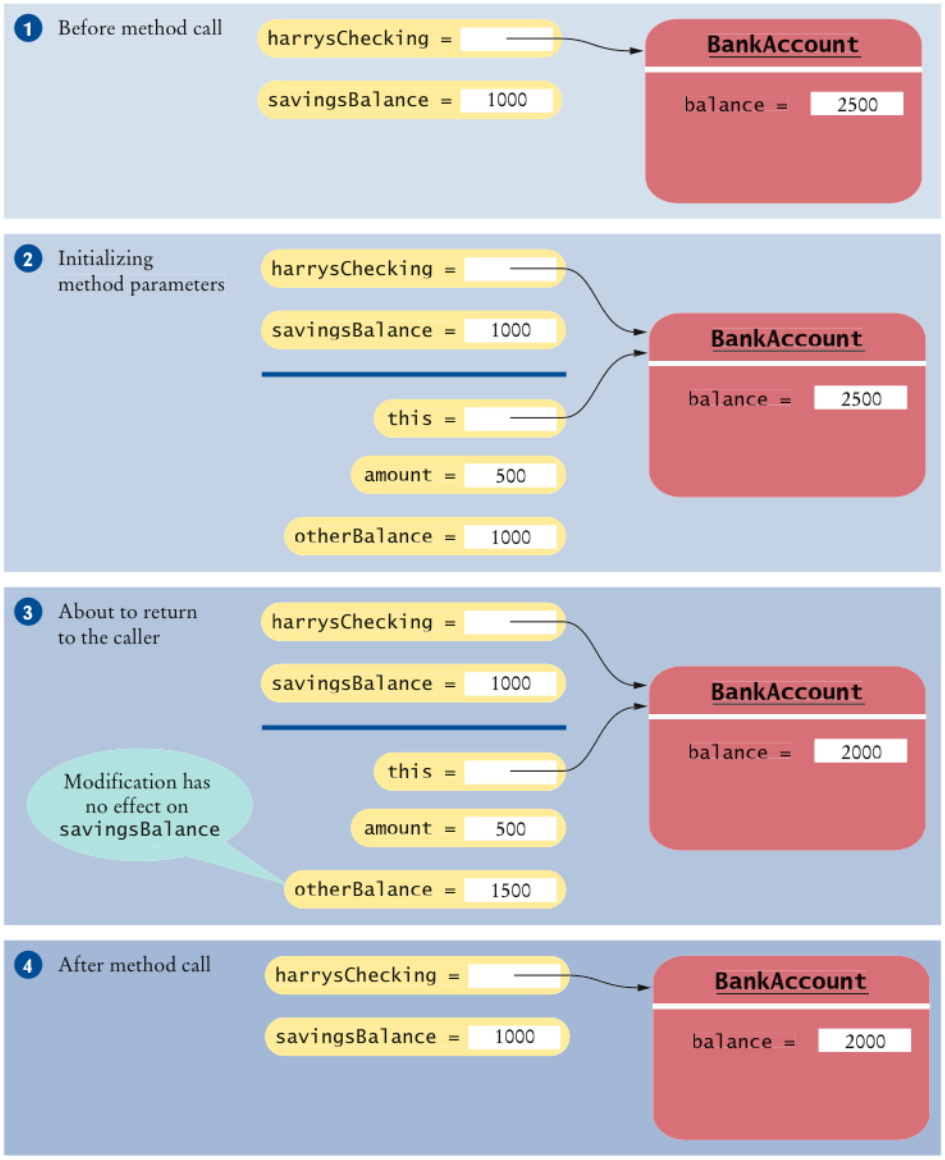


Figure 3 Modifying a Numeric Parameter Has No Effect on Caller

Call by Value and Call by Reference

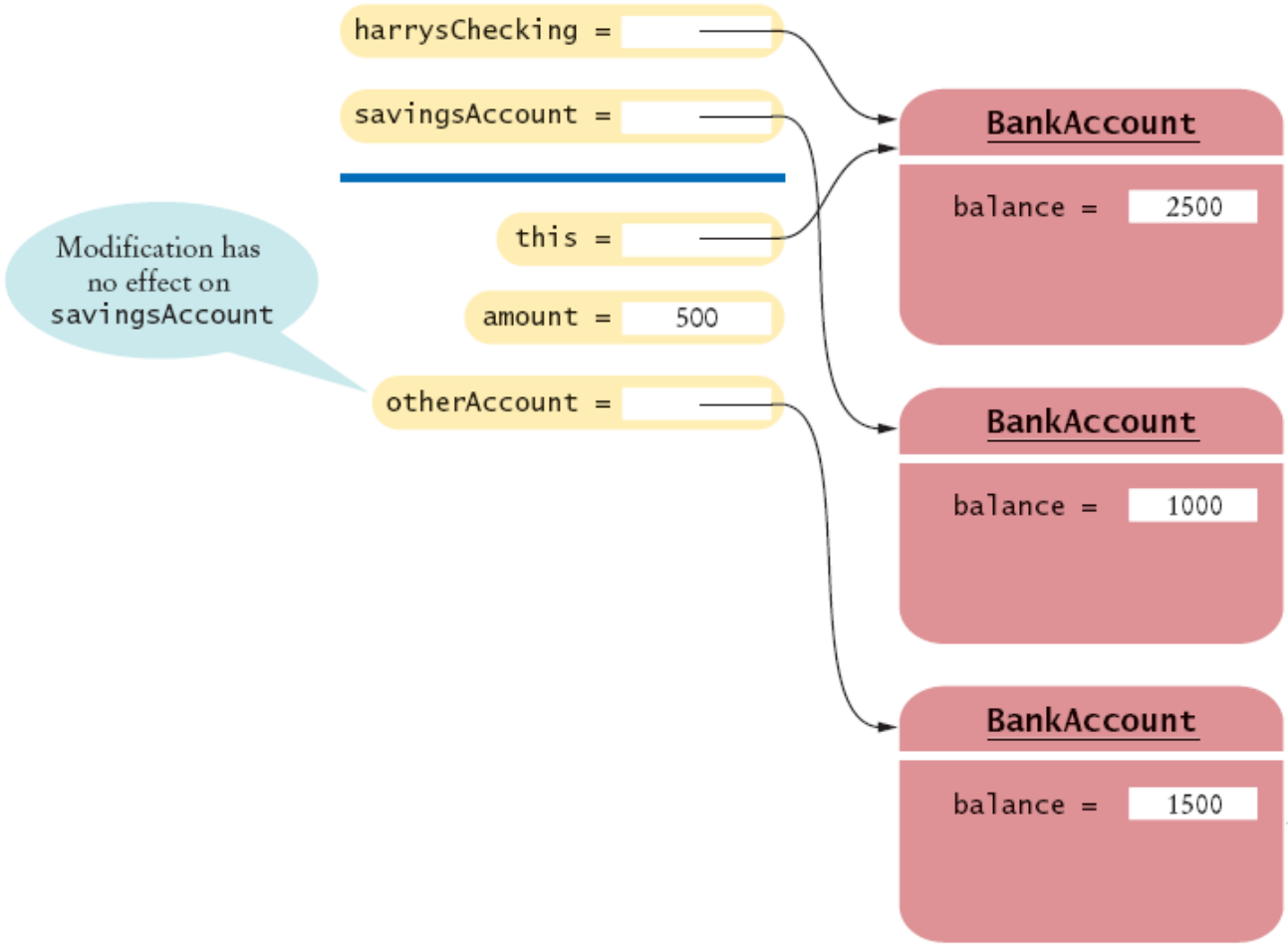
- **Call by value:** Method parameters are copied into the parameter variables when a method starts
- **Call by reference:** Methods can modify parameters
- Java has call by value
- A method can change state of object reference parameters, but cannot replace an object reference with another

Call by Value and Call by Reference

```
public class BankAccount
{
   public void transfer(double amount, BankAccount
      otherAccount)
   {
      balance = balance - amount;
      double newBalance = otherAccount.balance + amount;
      otherAccount = new BankAccount(newBalance);
      // Won't work
   }
}
```

Call by Value Example

```
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

Static Methods

- Every method must be in a class
- A static method is not invoked on an object
- Why write a method that does not operate on an object
- Common reason: encapsulate some computation that involves only numbers.
  - *Numbers aren't objects, you can't invoke methods on them. E.g.* `x.sqrt()` *can never be legal in Java*

Static Methods

- Example:

```
public class Financial
{
   public static double percentOf(double p, double a)
   {
      return (p / 100) * a;
   }
   // More financial methods can be added here.
}
```

- Call with class name instead of object:

```
double tax = Financial.percentOf(taxRate, total);
```

Static Methods

- If a method manipulates a class that you do not own, you cannot add it to that class
- A static method solves this problem:

```
public class Geometry
{
   public static double area(Rectangle rect)
   {
      return rect.getWidth() * rect.getHeight();
   }
   // More geometry methods can be added here.
}
```

- `main` is static — there aren't any objects yet

Self Check 8.12

Suppose Java had no static methods. How would you use the `Math.sqrt` method for computing the square root of a number *x*?

Answer:

```
Math m = new Math();
y = m.sqrt(x);
```

Self Check 8.13

The following method computes the average of an array list of numbers:

```
public static double average(ArrayList<Double> values)
```

Why must it be a static method?

Answer: You cannot add a method to the `ArrayList` class — it is a class in the standard Java library that you cannot modify.

Static Variables

- A static variable belongs to the class, not to any object of the class:

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

- If `lastAssignedNumber` was not `static`, each instance of `BankAccount` would have its own value of `lastAssignedNumber`

Static Variables

```
public BankAccount()
{
   // Generates next account number to be assigned
   lastAssignedNumber++; // Updates the static variable
   accountNumber = lastAssignedNumber;
   // Sets the instance variable
}
```

A Static Variable and Instance Variables

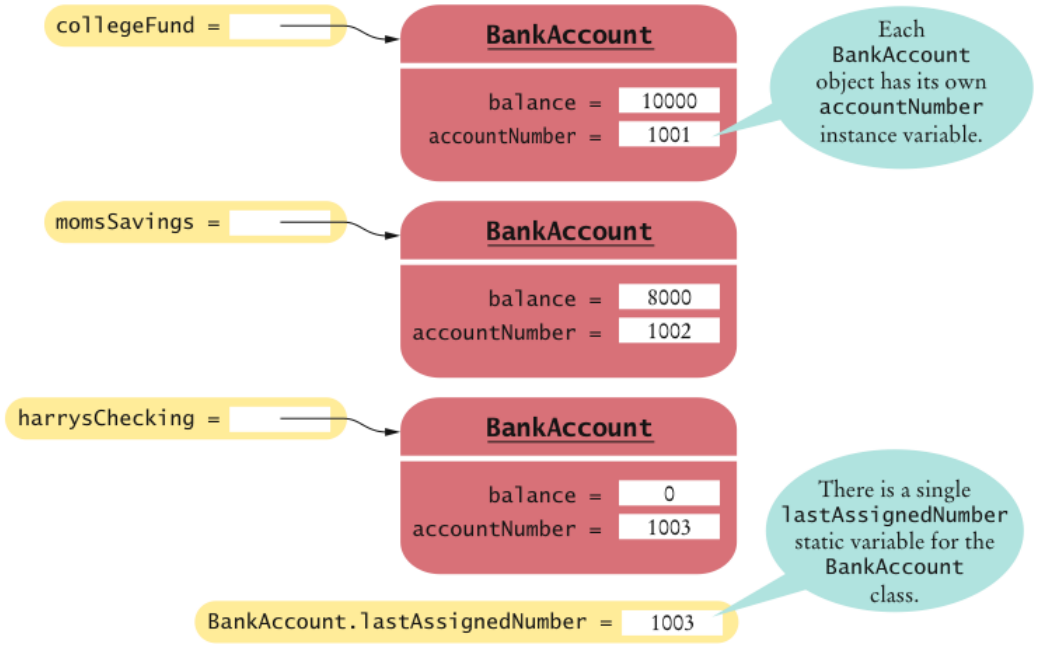


Figure 4
A Static Variable and Instance Variables

Static Variables

- Three ways to initialize:
  1. *Do nothing. variable is initialized with* `0` *(for numbers),* `false` *(for boolean values), or* `null` *(for objects)*
  2. *Use an explicit initializer, such as*

```
public class BankAccount
{
   ...
   private static int lastAssignedNumber = 1000;
      // Executed once,
}
```

  3. *Use a static initialization block*
- Static variables should always be declared as `private`

Static Variables

- Exception: Static constants, which may be either private or public:

```
public class BankAccount
{
   ...
   public static final double OVERDRAFT_FEE = 5;
   // Refer to it as BankAccount.OVERDRAFT_FEE
}
```

- Minimize the use of static variables (static final variables are ok)

Self Check 8.14

Name two static variables of the `System` class.

Answer: `System.in` and `System.out`.

Self Check 8.15

Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables `static`. Then `main` can call the other static methods, and all of them can access the static variables. Will Harry's plan work? Is it a good idea?

Answer: Yes, it works. Static methods can access static variables of the same class. But it is a terrible idea. As your programming tasks get more complex, you will want to use objects and classes to organize your programs.

Scope of Local Variables

- **Scope of variable:** Region of program in which the variable can be accessed
- Scope of a local variable extends from its declaration to end of the block that encloses it

Scope of Local Variables

- Sometimes the same variable name is used in two methods:

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

- These variables are independent from each other; their scopes are disjoint

Scope of Local Variables

- Scope of a local variable cannot contain the definition of another variable with the same name:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x);
   // Error - can't declare another variable
   // called r here
   ...
}
```

Scope of Local Variables

- However, can have local variables with identical names if scopes do not overlap:

```
if (x >= 0)
{
   double r = Math.sqrt(x);
   ...
   } // Scope of r ends here
else
{
   Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK - it is legal to declare another r here
   ...
}
```