Objective

Designing Classes

- To learn how to choose appropriate classes to implement
- 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 and post conditions
- To understand the difference between instance methods and static methods
- To introduce the concept of static fields

Objective

- To understand the scope rules for local variables and instance fields
- To learn about packages

Study sections 8.1, 8.2, 8.3, 8.4, and 8.6 from you text book.

Choosing Classes

- A class represents a single concept from the problem domain
- Class: Names in the problem definition
- Method: Verbs in the problem definition
- Concepts from mathematics:
  - Point
  - Rectangle
  - Ellipse
- Concepts from real life
  - BankAccount
  - CashRegister

Choosing Classes

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

    Random // better name: RandomNumberGenerator

- Utility classes–no objects, only static methods and constants like: Math
- Program starters: only have a `main` method
- Don't turn actions into classes:

`Paycheck` is better name than `ComputePaycheck`

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

Cohesion

- 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:

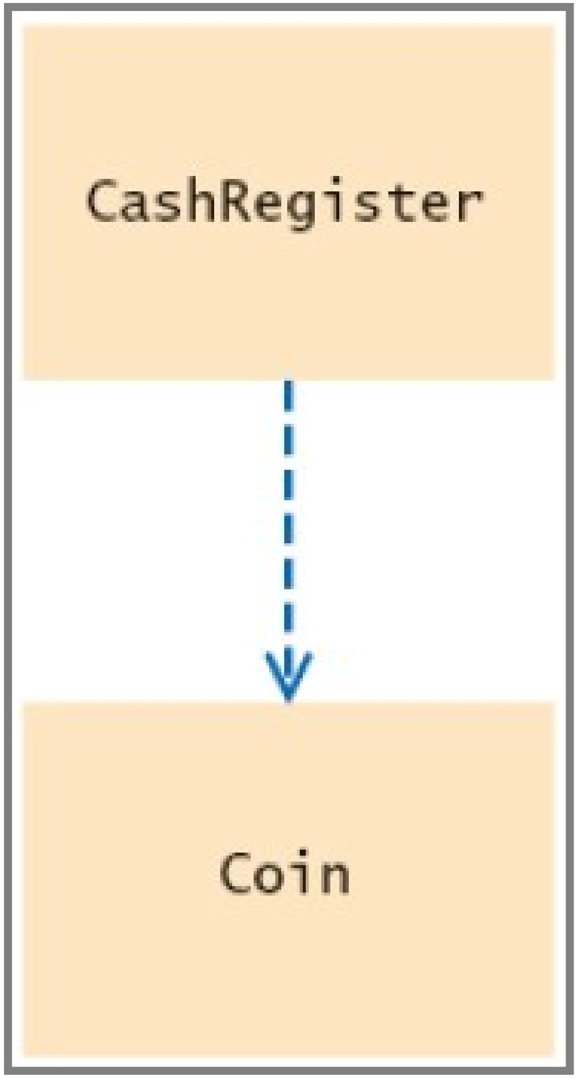
```
public class Coin
{
   public Coin(double aValue, String aName){ . . . }
   public double getValue(){ . . . }
   . . .
}

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

Coupling

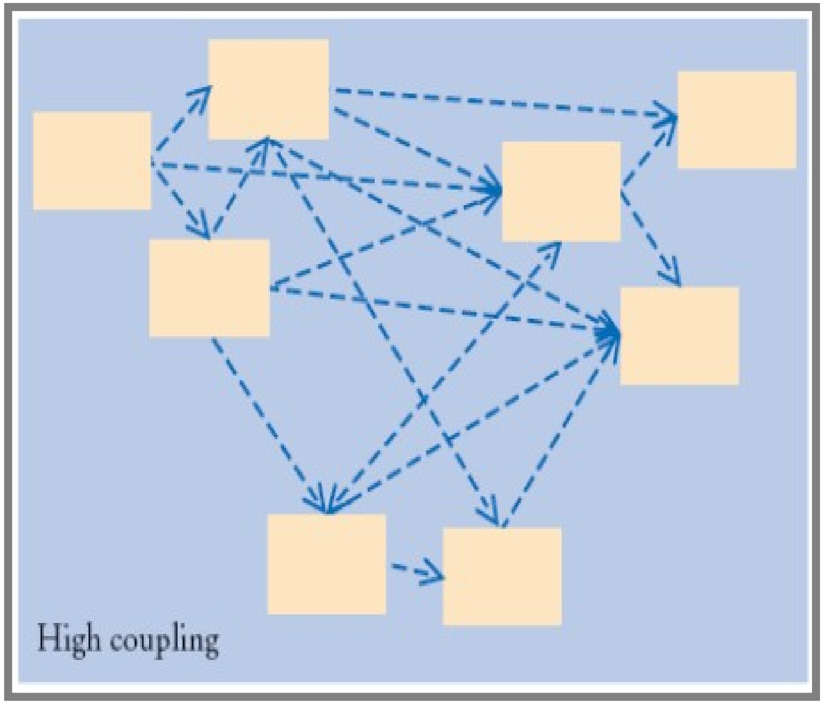
- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High Coupling = many class dependencies
- To visualize relationships draw class diagrams

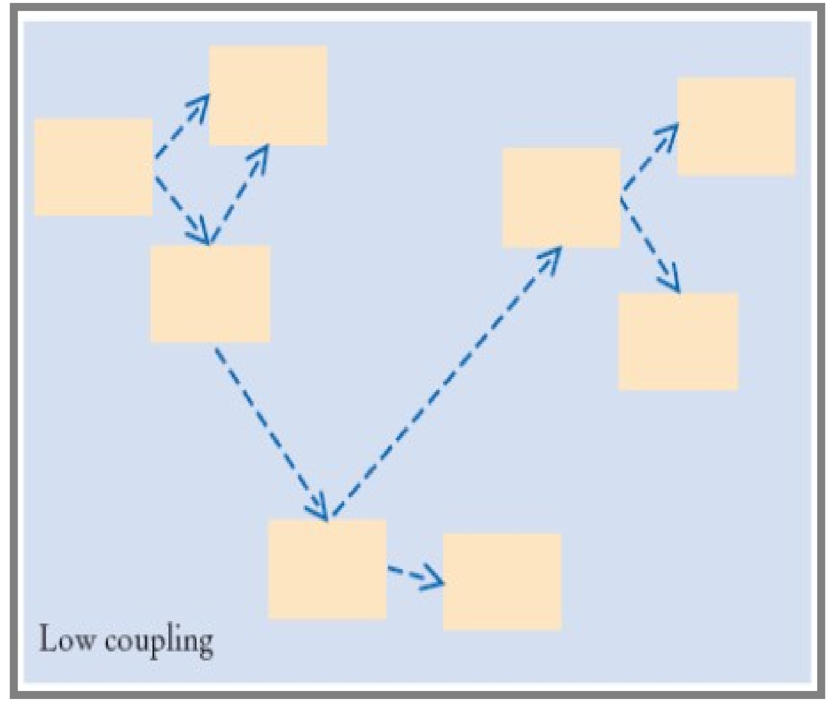
UML: Unified Modeling Language. Notation for object-oriented analysis and design

Coupling



High and Low Coupling Between Classes





Coupling

What is the problem with coupling?

- Delete unnecessary coupling.
- Minimize coupling to minimize the impact of interface changes

Maximize Cohesion
Minimize coupling (dependency)

Accessors, Mutators, and 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);
```

Accessors, Mutators, and Immutable Classes

- 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

Side Effects

Side effect of a method: any externally observable data modification

```
public void transfer(double amount, BankAccount other)
{
   balance = balance - amount;
   other.balance = other.balance + amount;
      // Modifies explicit parameter
}
```

Updating explicit parameter can be surprising to programmers; it is best to avoid it if possible.

Minimize side effect.

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`

It is best to decouple input/output from the actual work of your classes

Consistency

When you have a set of methods:

Follow a consistent scheme for their names and parameters. Example:

```
JoptionPane.showMessageDialog(null, message);
```

What is null argument here?

It seems that the `showMessageDialog` method needs an argument to specify the parent window, or null if no parent window required.

However, `showMessageDialog` method requires no parent window.

It could have been easily avoided.

Preconditions

- Precondition: Requirement that the caller of a method must meet
- Publish preconditions so the caller won't call methods with bad parameters

```
/**
   Deposits money into this account.
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

Preconditions

- Typical use:

–To restrict the parameters of a method

–To require that a method is only called when the object is in an appropriate state

- If precondition is violated, method is not responsible for computing the correct result.

Is it reasonable for a method to freely do anything if precondition is not met? What about formatting the hard disk of the user.

Preconditions

What a method can do If precondition is not met?

1. Method may throw exception if precondition violated, more Chapter 11 later.

```
if (amount < 0) throw new IllegalArgumentException();
balance = balance + amount;
```

2. Method can continue with the wrong data. The failure is the responsibility of the caller.

```
// if this makes the balance negative, it's the caller's fault
balance = balance + amount;
```

Preconditions

- Method doesn't have to test for precondition. (Test may be costly)

```
// if this makes the balance negative, it's the caller's fault
balance = balance + amount;
```

Preconditions

- Method can do an assertion check assert (more on Chapter 11 later)

```
assert amount >= 0;
balance = balance + amount;
```

To enable assertion checking:

`java -enableassertions MyProg`

Or

`java -en MyProg`

You can turn assertions off after you have tested your program, so that it runs at maximum speed

Preconditions

- Many beginner programmers silently return to the caller.

What is the problem with the following code?

```
if (amount < 0) return; // Not recommended; hard to debug
balance = balance + amount;
```

assert *condition*;

Example:

assert amount >= 0;

Purpose:

To assert that a condition is fulfilled. If assertion checking is enabled and the condition is false, an assertion error is thrown.

Postconditions

- Condition that is true after a method has completed
- If method call is in accordance with preconditions, it must ensure that postconditions are valid
- There are two kinds of postconditions:

1. The return value is computed correctly
2. The object is in a certain state after the method call is completed

Postconditions

```
/**
   Deposits money into this account.
   (Precondition: amount >= 0)
   @param amount the amount of money to deposit
   (Postcondition: getBalance() >= 0)
*/
```

Don't document trivial postconditions that repeat the `@return` clause description of the method.

Postconditions

Formulate pre- and post-conditions only in terms of the public interface of the class.

```
amount <= getBalance()
   // this is the way to state a postcondition


amount <= balance

   // wrong postcondition formulation
```

- Contract: If caller fulfills precondition, method must fulfill postcondition

Static Methods

Every method must be in a class

A static method <u>is not</u> 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

```
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 obje

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

Static Fields

A static field belongs to the class, not to any object of the class. Also called *class field*.

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

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

Static Fields

```
public BankAccount()
{
   // Generates next account number to be assigned
   lastAssignedNumber++; // Updates the static field


// Assigns field to account number of this bank
   account accountNumber = lastAssignedNumber;
      // Sets the instance field
}
```

Minimize the use of static fields. (Static final fields are ok.)

Static Fields

Three ways to initialize:

1. Do nothing. Field is 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 when class is loaded
}
```

3. Not worth learning it. Nobody use it.

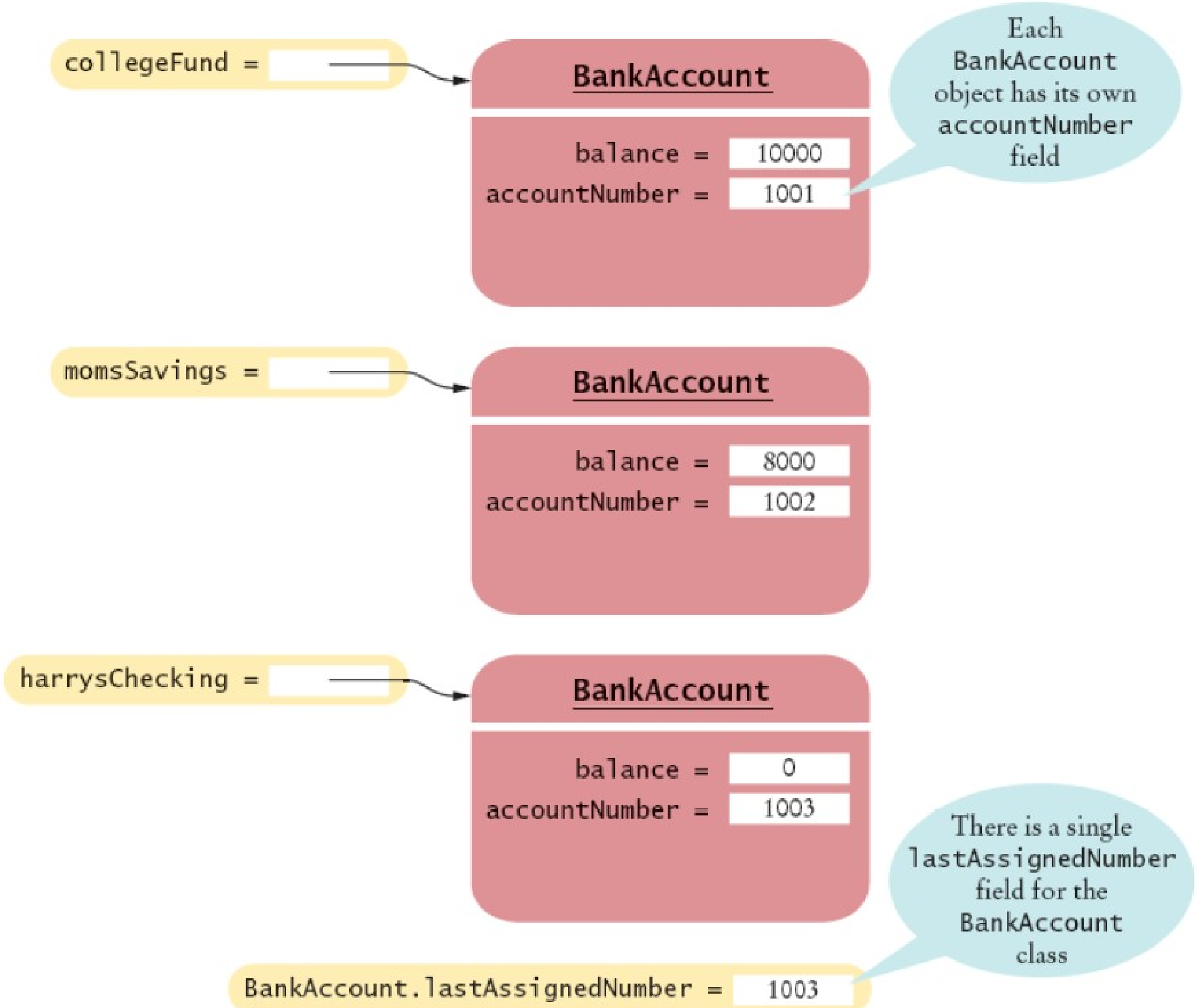
Static Fields

Static fields should always be declared as `private`

Exception: Static constants (final), which may be either private or public

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

A Static Field and Instance Fields



Static

Static methods are applied to entire class, not instances of objects.

Minimize using static fields and methods (why?)

The `main` method is static–there aren't any objects yet

Summary

1. A class should represent a single concept
2. The public interface of a class is cohesive if all of its features are related to the concept that the class represents
3. A class depends to another class if its method use that class in any way
4. An immutable class has no mutator methods
5. A side effect of a method is any externally observable data modification or action
6. Minimize side effect
7. Minimize coupling
8. Maximize cohesion
9. Precondition is the requirement that the caller of a method must meet

Summary

10. Postcondition is the state that is true after a method has completed
11. Formulate pre- and post-conditions only in terms of the public interface of the class.
12. A static variable belongs to the class, not to the object.
13. A static method is not invoked on an object.