Day 04

PROGRAMMING IN JAVA

FUNDAMENTALS OF TELECOMMUNICATIONS LAB

Dr. Huy Nguyen

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

PROGRAMMING IN JAVA

DESIGNING CLASSES

Chapter 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 and postconditions
- To understand static methods and variables
- To understand the scope rules for local variables and instance variables
- To learn about packages
- To learn about unit testing frameworks

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

What is the rule of thumb for finding classes?

Self Check

Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `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 static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   ...
   public void enterPayment(int dollars, int quarters, int
dimes, int nickels, int pennies)
   ...
 }
```

Cohesion

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

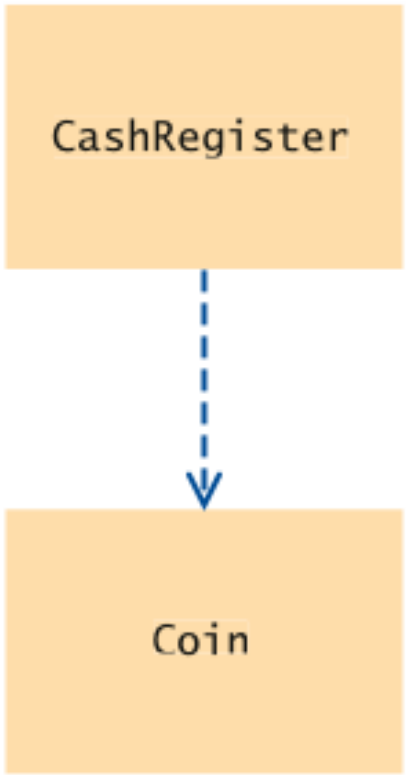
```
public class Coin
{
   public Coin(double aValue, String aName) { ... }
   public double getValue() { ... }
   ...
}

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

Coupling

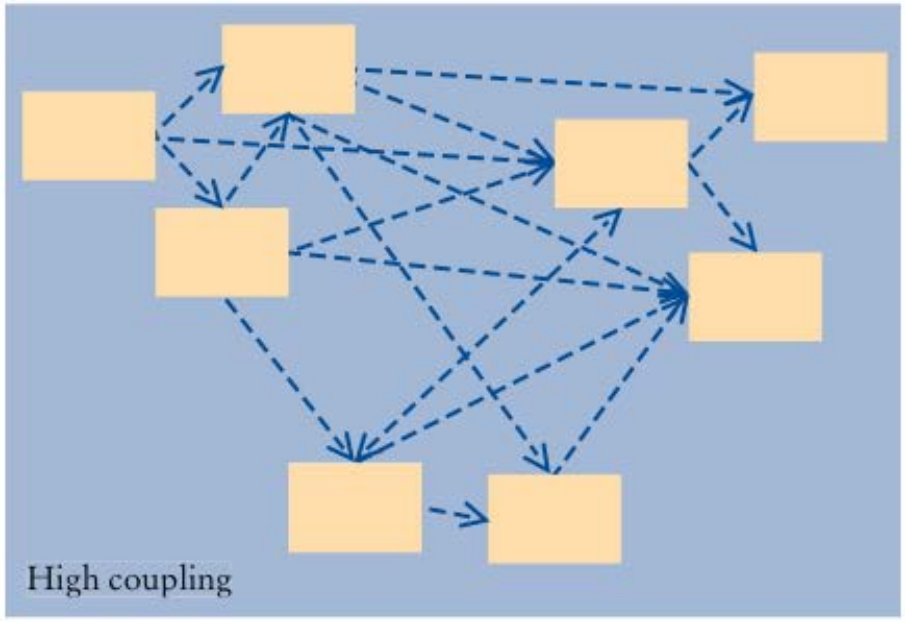
- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High coupling = Many class dependencies
- Minimize coupling to minimize the impact of interface changes
- To visualize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

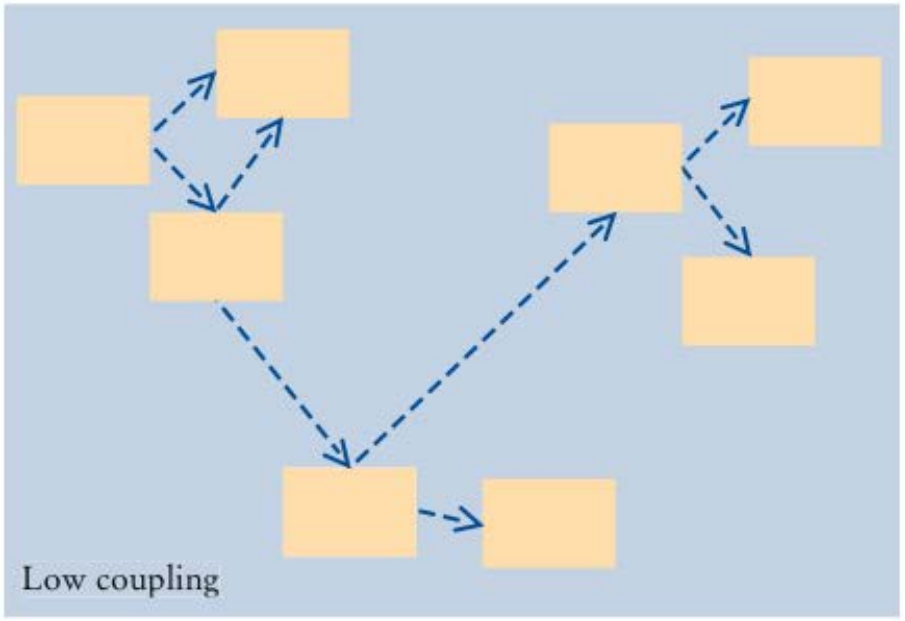
Dependency

- Dependency relationship between the CashRegister and Coin classes



High and Low Coupling Between Classes





Self Check

Why is the `CashRegister` class from the previous example not cohesive?

Self Check

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

Self Check

Why should coupling be minimized between classes?

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

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

Self Check

Is the `Rectangle` class immutable?

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

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

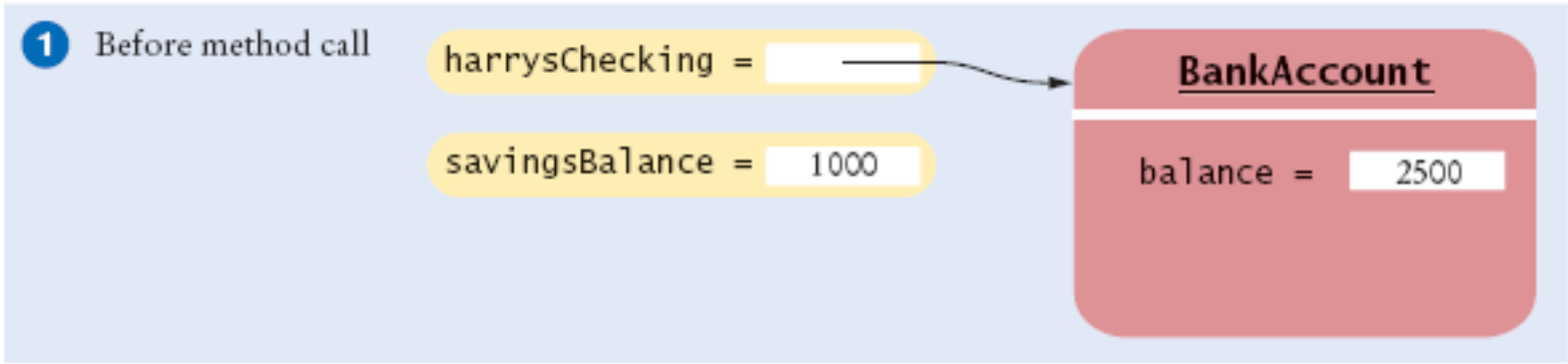
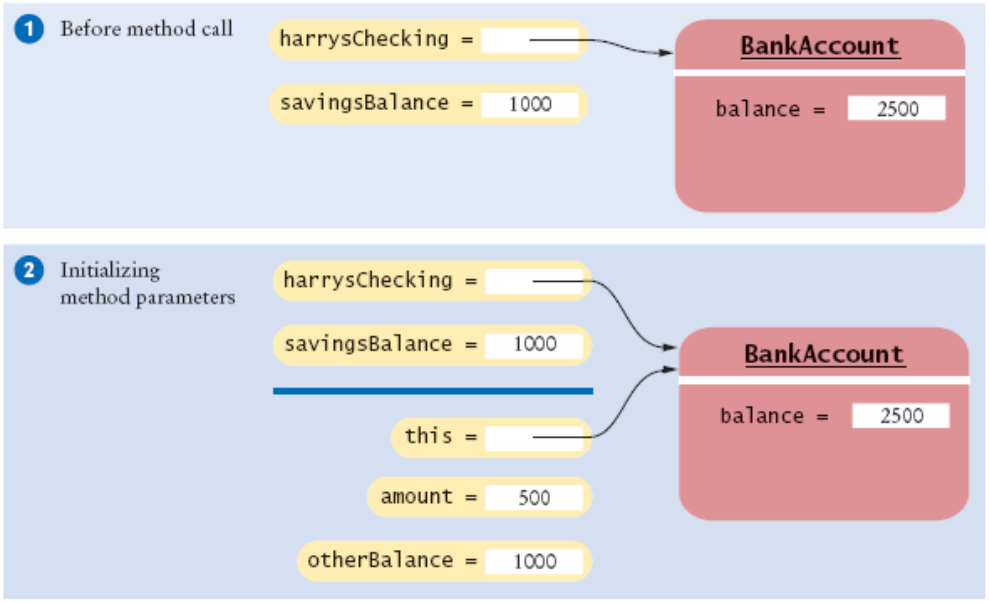
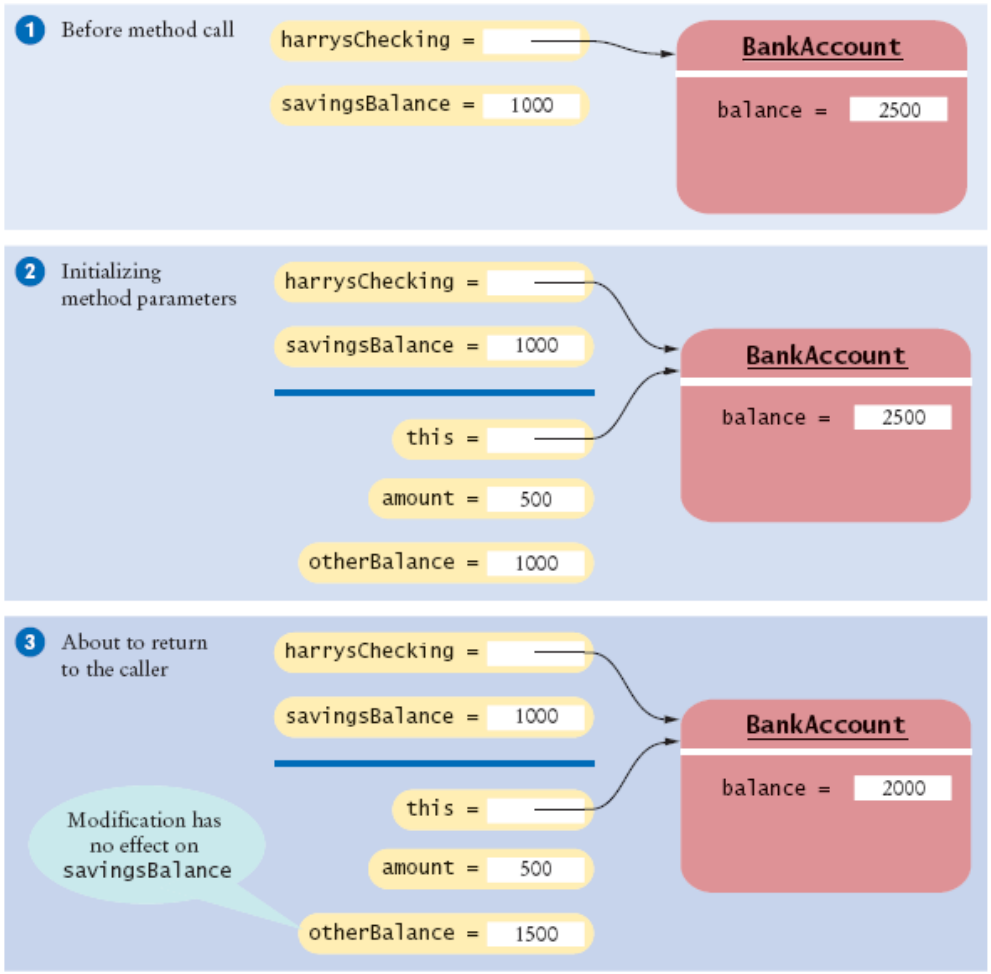
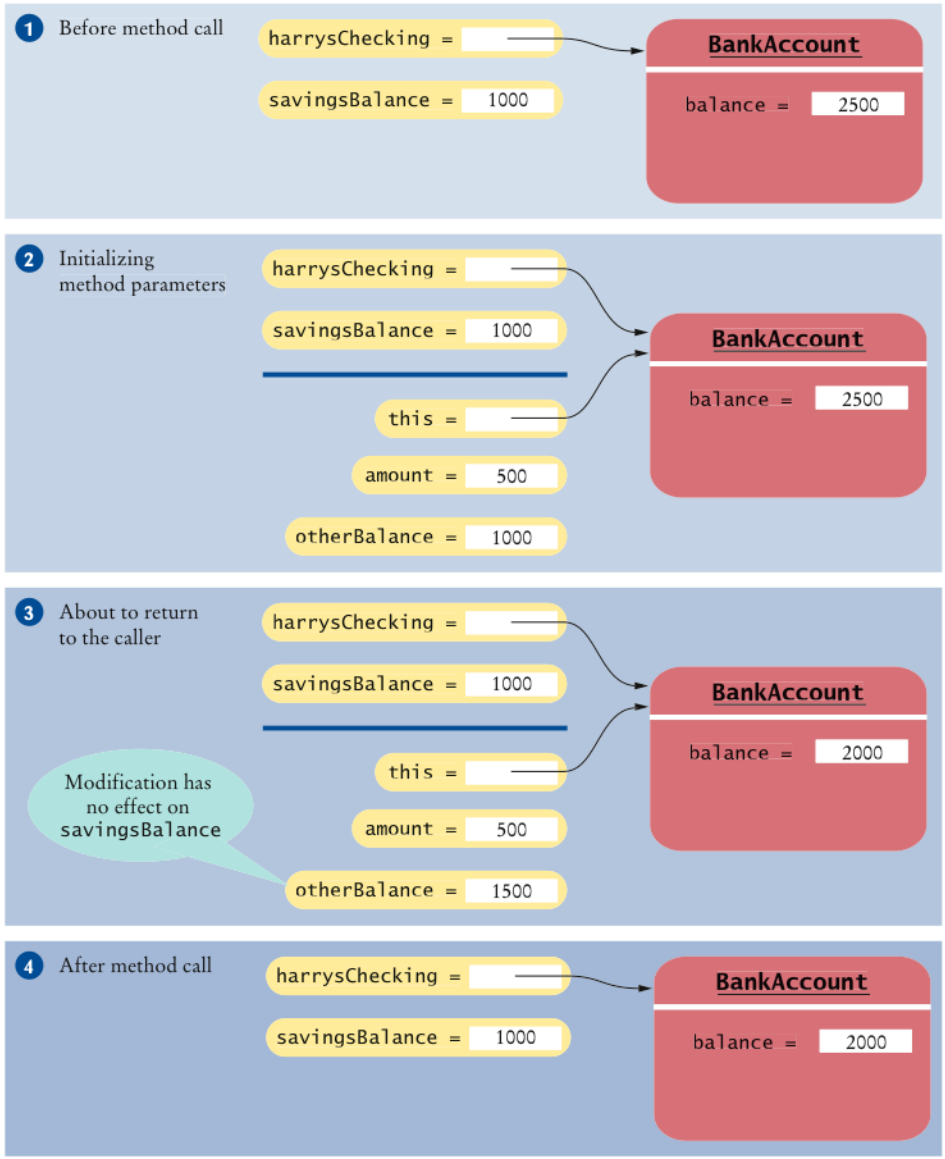
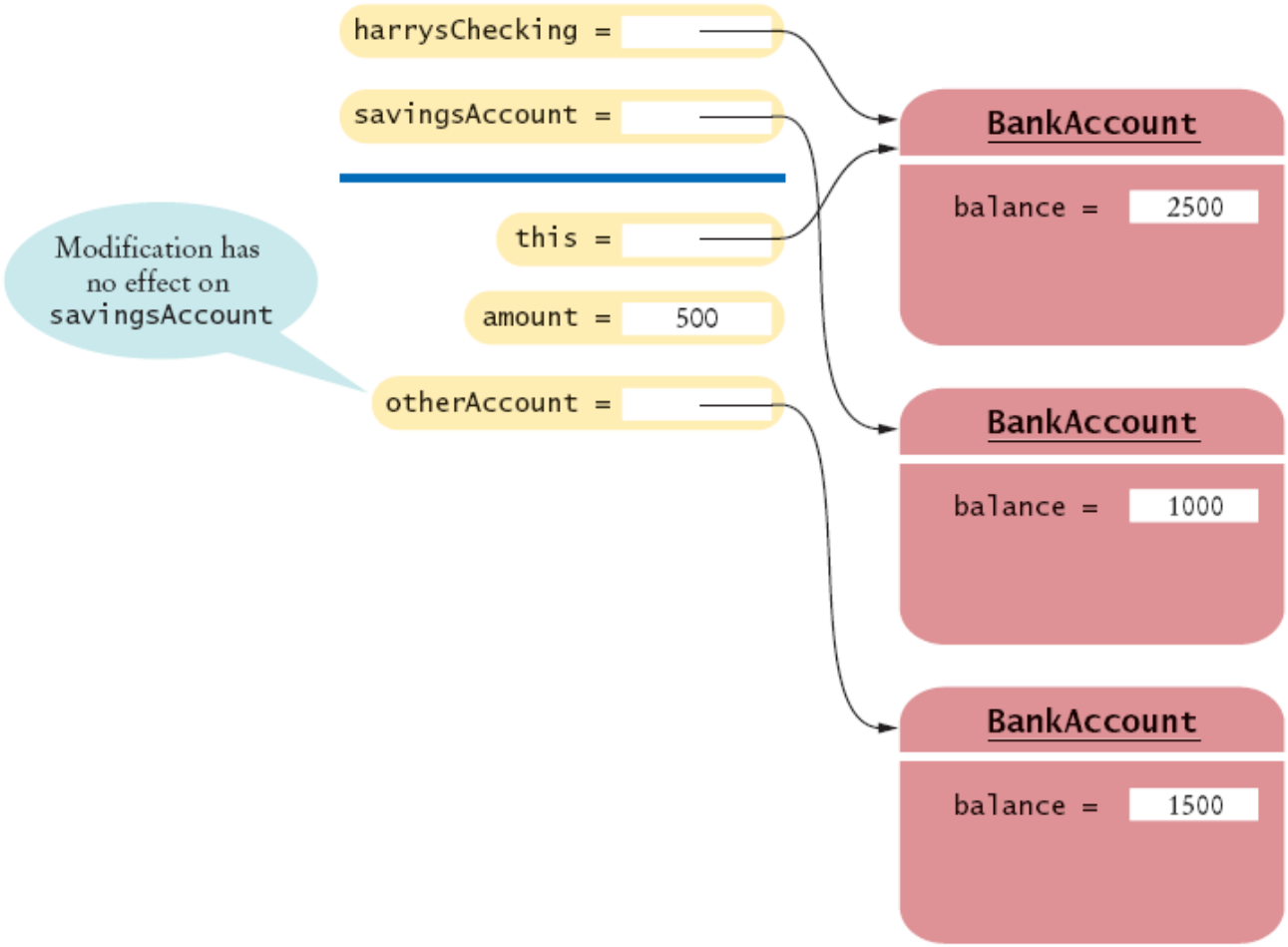
Self Check

Consider the `DataSet` class in the previous example. Suppose we add a method

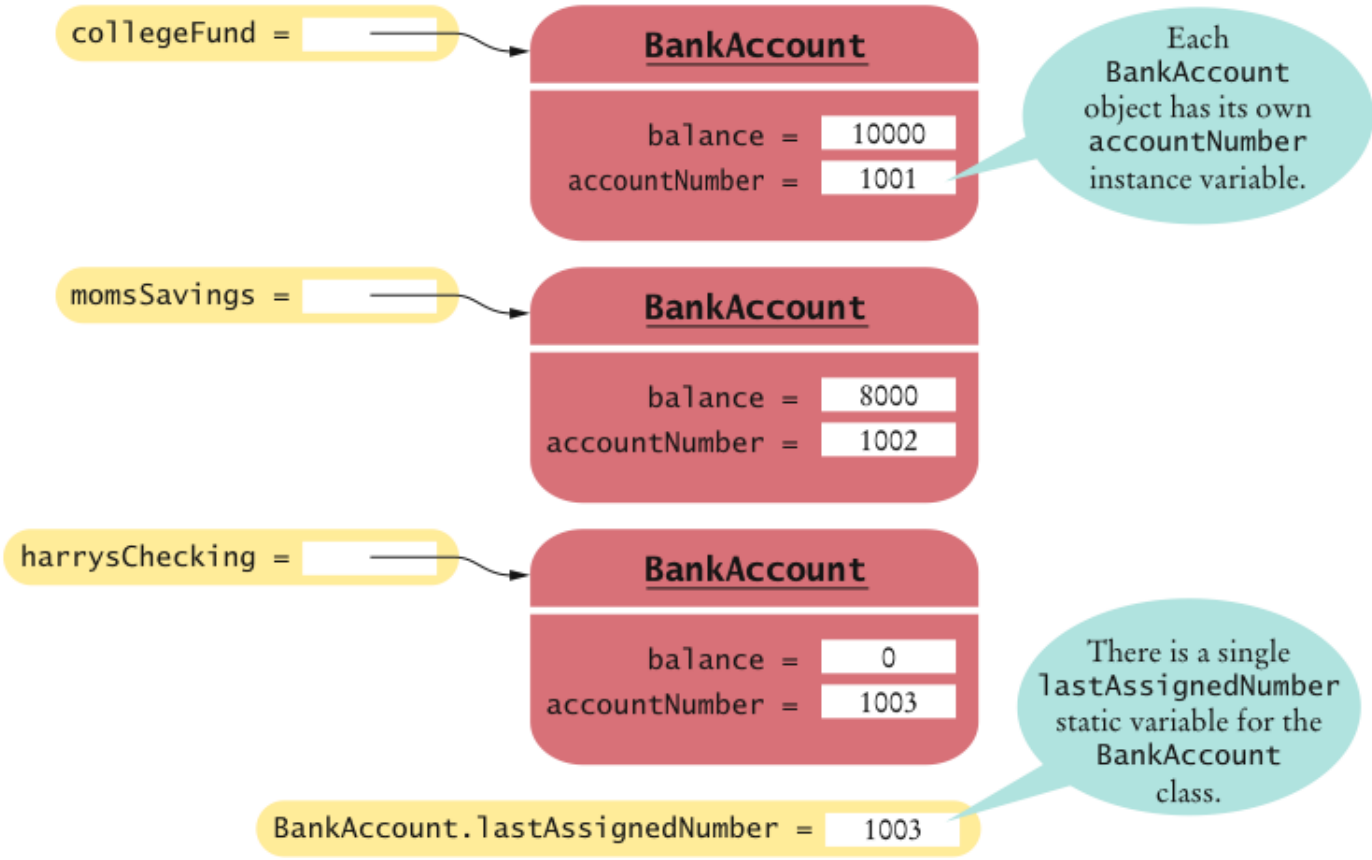
```
void read(Scanner in)
{
   while(in.hasNextDouble())
      add(in.nextDouble());
}
```

Does this method have a side effect other than mutating the data set?

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



Common Error: Trying to Modify Primitive Type Parameters

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

Common Error: Trying to Modify Primitive Type Parameters



Common Error: Trying to Modify Primitive Type Parameters

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

Common Error: Trying to Modify Primitive Type Parameters



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

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)
*/
```

- Typical use:
  1. *To restrict the parameters of a method*
  2. *To require that a method is only called when the object is in an appropriate state*

Preconditions

- If precondition is violated, method is not responsible for computing the correct result. It is free to do *anything*
- Method may throw exception if precondition violated:

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

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

  To enable assertion checking:

```
java -enableassertions MainClass
```

  You can turn assertions off after you have tested your program, so that it runs at maximum speed
- Many beginning programmers silently return to the caller

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

Syntax Assertion

Syntax `assert` *condition*;

Example

```
assert amount >= 0;
```

If the condition is false and assertion checking is enabled, an exception occurs.

Condition that is claimed to be true.

Postconditions

- **Postcondition:** Requirement 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:
  - *The return value is computed correctly*
  - *The object is in a certain state after the method call is completed*
-
```
/**
   Deposits money into this account.
   (Postcondition: getBalance() >= 0)
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

Postconditions

- Don’t document trivial postconditions that repeat the `@return` clause
- Formulate pre- and postconditions only in terms of the interface of the class:

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

- Contract: If caller fulfills preconditions, method must fulfill postconditions

Self Check

Why might you want to add a precondition to a method that you provide for other programmers?

Self Check

When you implement a method with a precondition and you notice that the caller did not fulfill the precondition, do you have to notify 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

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

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?

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



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

Name two static variables of the `System` class.

Self Check

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?

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
   ...
}
```

Overlapping Scope

- A local variable can *shadow* a variable with the same name
- Local scope wins over class scope:

```
public class Coin
{
   ...
   public double getExchangeValue(double exchangeRate)
   {
      double value; // Local variable
      ...
      return value;
   }
   private String name;
   private double value; // variable with the same
name
}
```

Overlapping Scope

- Access shadowed variables by qualifying them with the `this` reference:

```
value = this.value * exchangeRate;
```

- Generally, shadowing an instance variable is poor code - error-prone, hard to read
- Exception: when implementing constructors or setter methods, it can be awkward to come up with different names for instance variables and parameters
- OK:

```
public Coin(double value, String name)
{
   this.value = value;
   this.name = name;
}
```

Self Check

Consider the following program that uses two variables named `r`. Is this legal?

```
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);
   }
}
```

Self Check

What is the scope of the `balance` variable of the `BankAccount` class?

Packages

- **Package:** Set of related classes
- Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages

- To put classes in a package, you must place a line

```
package packageName;
```

  as the first instruction in the source file containing the classes
- Package name consists of one or more identifiers separated by periods

Organizing Related Classes into Packages

- For example, to put the `Financial` class introduced into a package named `com.horstmann.bigjava`, the `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;

public class Financial
{
   ...
}
```

- Default package has no name, no `package` statement

Syntax Package Specification

Syntax `package` *packageName*;

Example

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.

Importing Packages

- Can always use class without importing:

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

- Tedious to use fully qualified name
- Import lets you use shorter class name:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in)
```

- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import `java.lang`
- You don’t need to import other classes in the same package

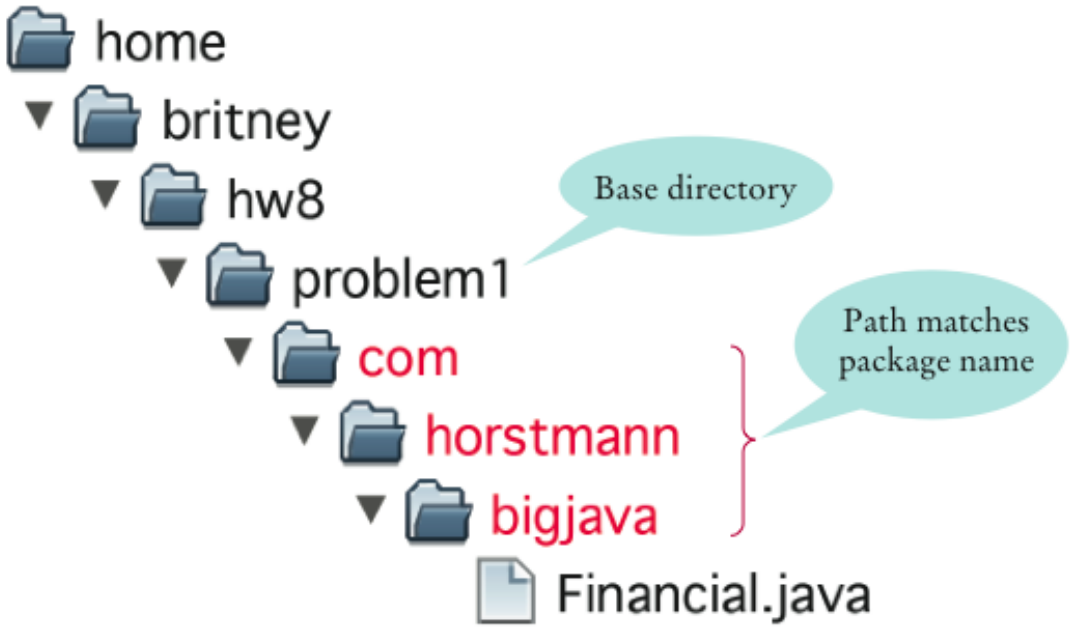
Package Names

- Use packages to avoid name clashes
  `java.util.Timer`
  vs.
  `javax.swing.Timer`
- Package names should be unambiguous
- Recommendation: start with reversed domain name:
  `com.horstmann.bigjava`
- `edu.sjsu.cs.walters`: for Britney Walters' classes (`walters@cs.sjsu.edu`)
- Path name should match package name:
  `com/horstmann/bigjava/Financial.java`

Package and Source Files

- **Base directory:** holds your program's Files
- Path name, relative to base directory, must match package name:

```
com/horstmann/bigjava/Financial.java
```



Self Check

Which of the following are packages?

a. `java`
b. `java.lang`
c. `java.util`
d. `java.lang.Math`

Answer:

a.No
b.Yes
c.Yes
d.No

Self Check

Is a Java program without `import` statements limited to using the default and `java.lang` packages?

Self Check

Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Answer: `/home/me/cs101/hw1/problem1` or, on Windows, `c:\Users\me\cs101\hw1\problem1`

Unit Testing Frameworks

- Unit test frameworks simplify the task of writing classes that contain many test cases
- JUnit: `http://junit.org`
  - *Built into some IDEs like BlueJ and Eclipse*
- Philosophy: whenever you implement a class, also make a companion test class. Run all tests whenever you change your code

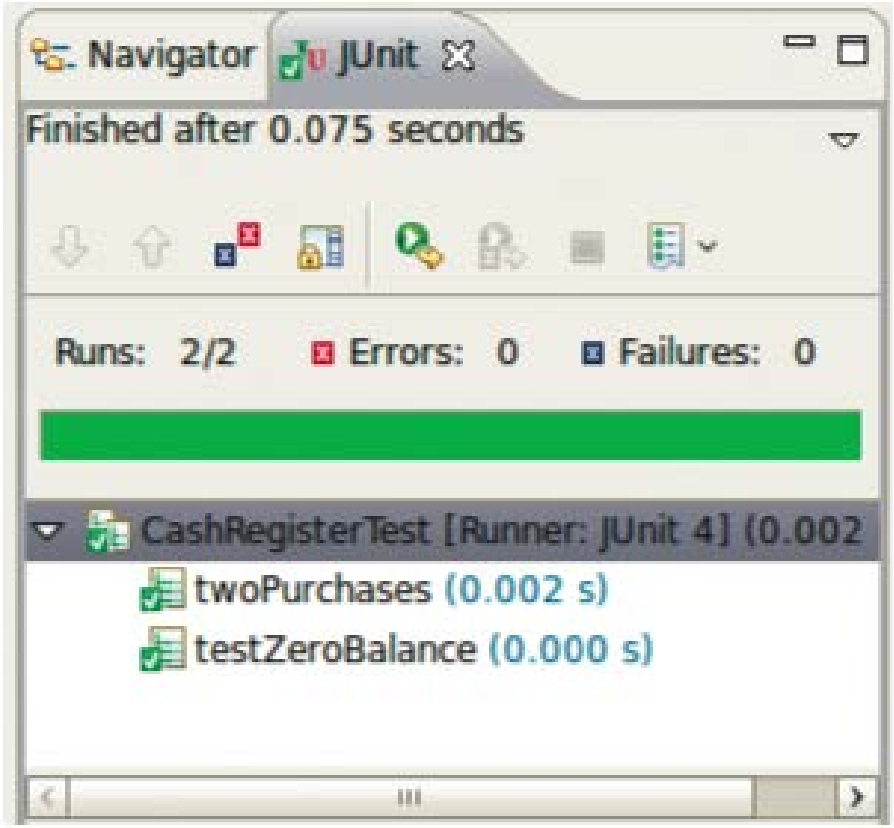
Unit Testing Frameworks

- Customary that name of the test class ends in `Test`:

```
import org.junit.Test;
import org.junit.Assert;
public class CashRegisterTest
{
   @Test public void twoPurchases()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assertEquals(expected, register.giveChange(),
         EPSILON);
   }
   // More test cases
   . . .
}
```

Unit Testing Frameworks

- If all test cases pass, the JUnit tool shows a green bar:



Self Check

Provide a JUnit test class with one test case for the `Earthquake` class in the previous example.

Answer: Here is one possible answer, using the JUnit 4 style.

```
public class EarthquakeTest
{
   @Test public void testLevel4()
   {
      Earthquake quake = new Earthquake(4);
      Assert.assertEquals("Felt by many people, no destruction”,
         quake.getDescription());
   }
}
```

Self Check

What is the significance of the `EPSILON` parameter in the `assertEquals` method?

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

PROGRAMMING IN JAVA

OBJECT-ORIENTED DESIGN

Chapter Goals

- To learn about the software life cycle
- To learn how to discover new classes and methods
- To understand the use of CRC cards for class discovery
- To be able to identify inheritance, aggregation, and dependency relationships between classes
- To master the use of UML class diagrams to describe class relationships
- To learn how to use object-oriented design to build complex programs

The Software Life Cycle

- Encompasses all activities from initial analysis until obsolescence
- Formal process for software development
  - *Describes phases of the development process*
  - *Gives guidelines for how to carry out the phases*
- Development process
  - *Analysis*
  - *Design*
  - *Implementation*
  - *Testing*
  - *Deployment*

Analysis

- Decide what the project is supposed to do
- Do not think about how the program will accomplish tasks
- Output: Requirements document
  - *Describes what program will do once completed*
  - *User manual: Tells how user will operate program*
  - *Performance criteria*

Design

- Plan how to implement the system
- Discover structures that underlie problem to be solved
- Decide what classes and methods you need
- Output:
  - *Description of classes and methods*
  - *Diagrams showing the relationships among the classes*

Implementation

- Write and compile the code
- Code implements classes and methods discovered in the design phase
- Program Run: Completed program

Testing

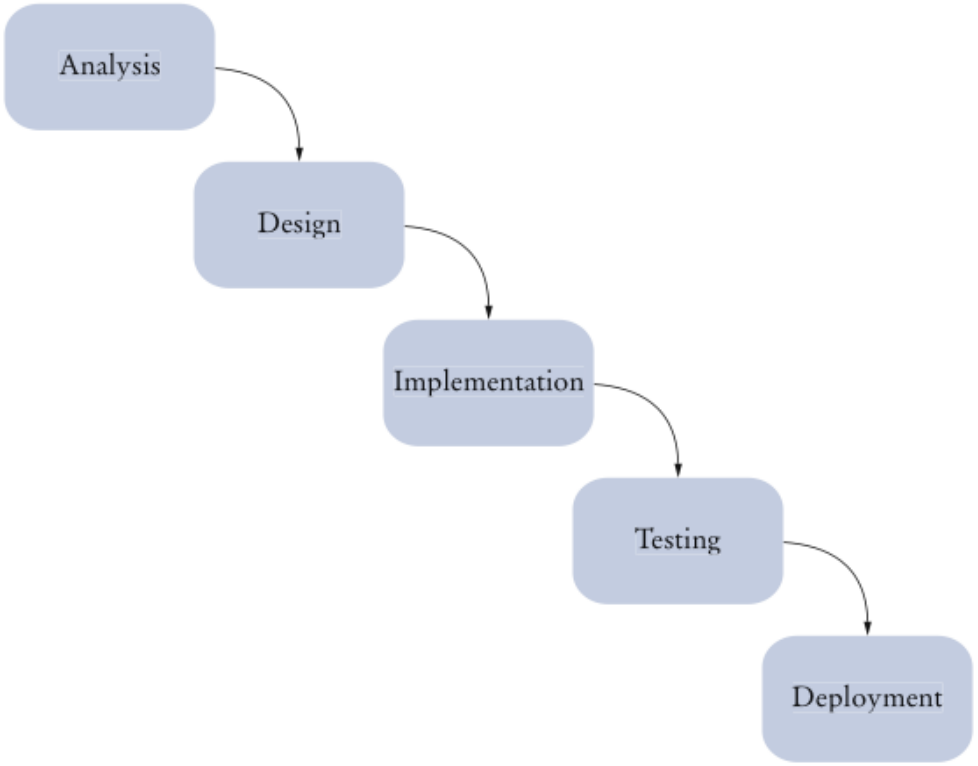
- Run tests to verify the program works correctly
- Program Run: A report of the tests and their results

Deployment

- Users install program
- Users use program for its intended purpose

The Waterfall Model

- Sequential process of analysis, design, implementation, testing, and deployment
- When rigidly applied, waterfall model did not work

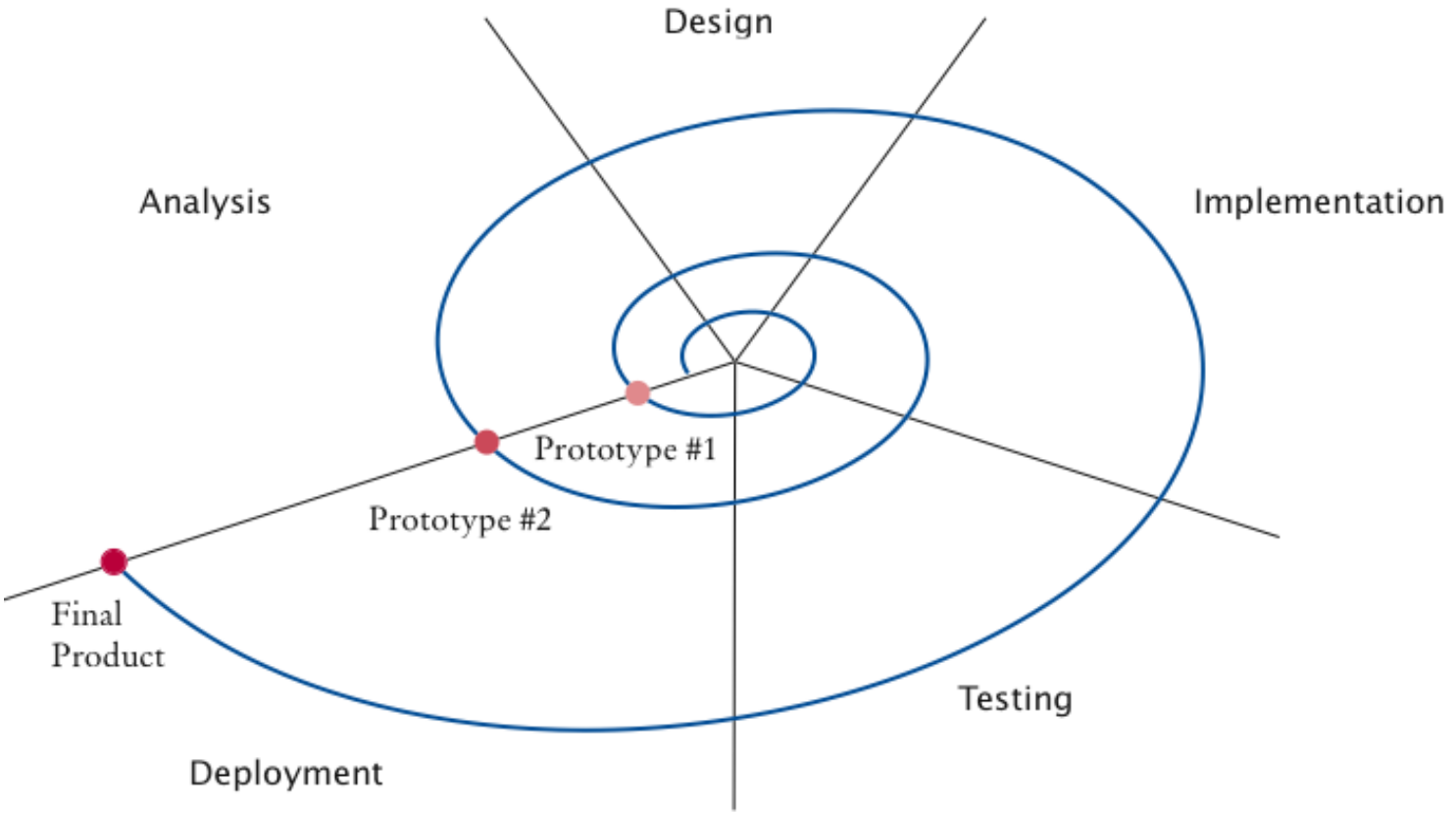


The Spiral Model

- Breaks development process down into multiple phases
- Early phases focus on the construction of *prototypes*
- Lessons learned from development of one prototype can be applied to the next iteration

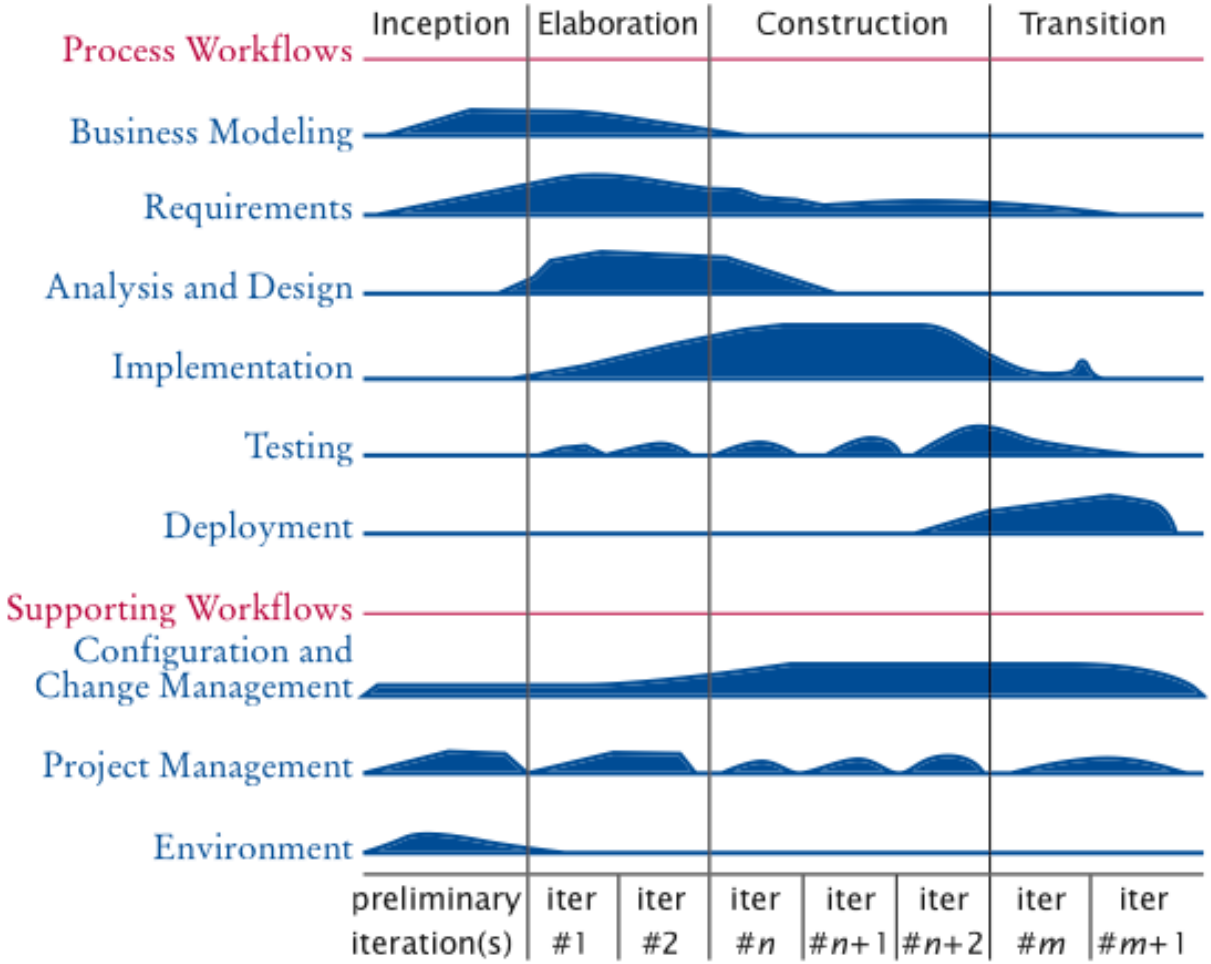
The Spiral Model

- Problem: Can lead to many iterations, and process can take too long to complete



Activity Levels in the Rational Unified Process

Development process methodology by the inventors of UML



Extreme Programming

- Realistic planning
  - *Customers make business decisions*
  - *Programmers make technical decisions*
  - *Update plan when it conflicts with reality*
- Small releases
  - *Release a useful system quickly*
  - *Release updates on a very short cycle*
- Pair programming
  - *Two programmers write code on the same computer*
- Simplicity
  - *Design as simply as possible instead of preparing for future complexities*
- Testing
  - *Programmers and customers write test cases*
  - *Test continuously*
- Coding standards
  - *Follow standards that emphasize self-documenting code*
- 40-hour week
  - *Don’t cover up unrealistic schedules with heroic effort*

Self Check

Suppose you sign a contract, promising that you will, for an agreed-upon price, design, implement, and test a software package exactly as it has been specified in a requirements document. What is the primary risk you and your customer are facing with this business arrangement?

Self Check

Does Extreme Programming follow a waterfall or a spiral model?

Self Check

What is the purpose of the “on-site customer” in Extreme Programming?

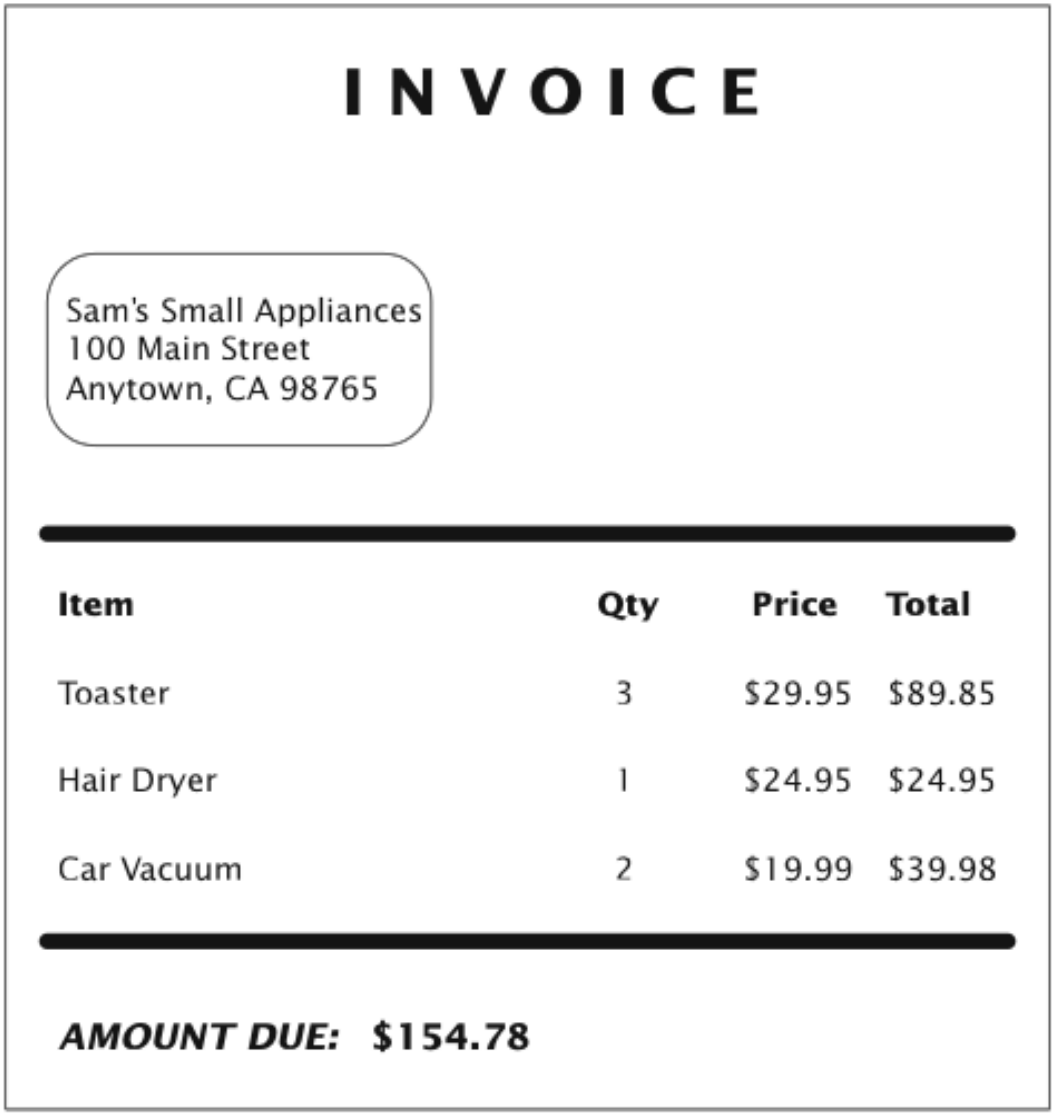
Object-Oriented Design

1. Discover classes
2. Determine responsibilities of each class
3. Describe relationships between the classes

Discovering Classes

- A class represents some useful concept
- Concrete entities: Bank accounts, ellipses, and products
- Abstract concepts: Streams and windows
- Find classes by looking for nouns in the task description
- Define the behavior for each class
- Find methods by looking for verbs in the task description

Example: Invoice



INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

AMOUNT DUE: **$154.78**

Example: Invoice

- Classes that come to mind: `Invoice`, `LineItem`, and `Customer`
- Good idea to keep a list of candidate classes
- Brainstorm, simply put all ideas for classes onto the list
- You can cross not useful ones later

Finding Classes

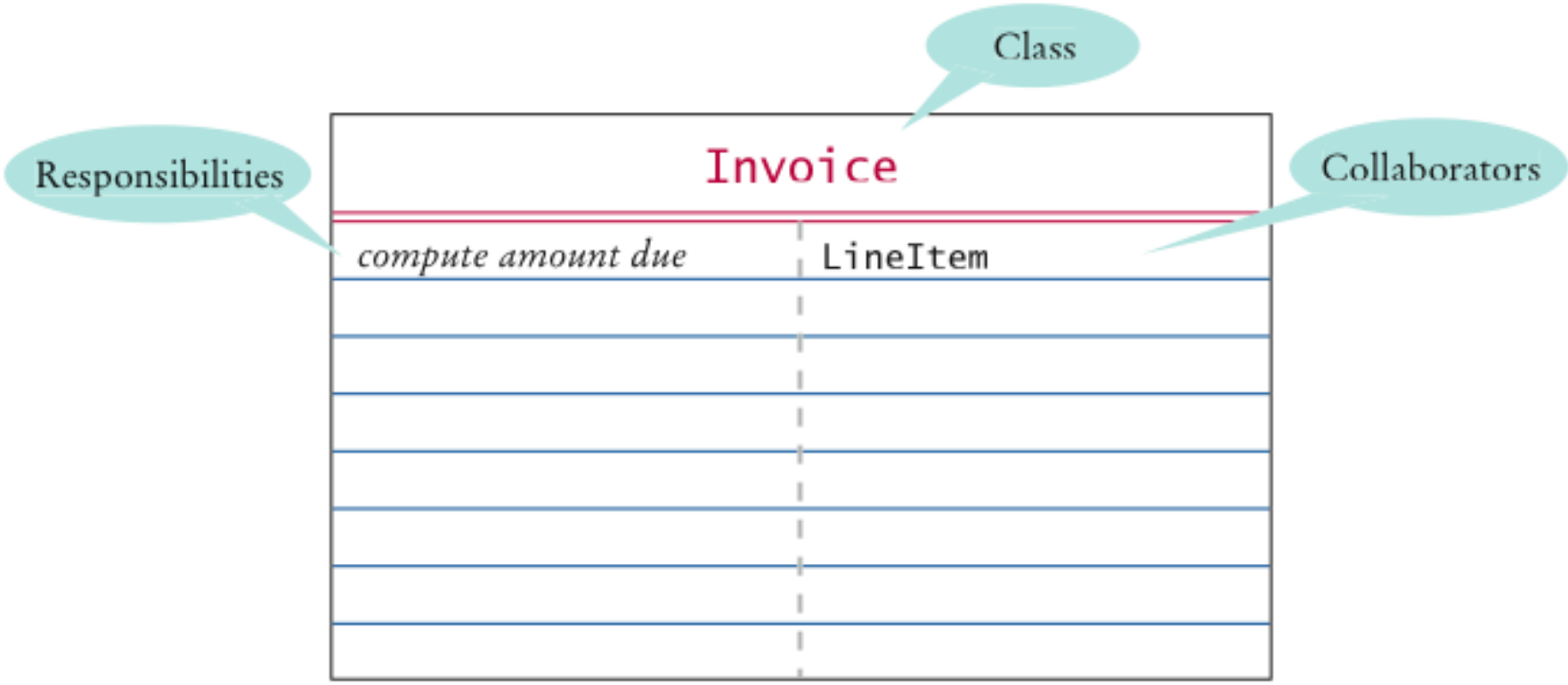
- Keep the following points in mind:
  - *Class represents set of objects with the same behavior*
    - *Entities with multiple occurrences in problem description are good candidates for objects*
    - *Find out what they have in common*
    - *Design classes to capture commonalities*
  - *Represent some entities as objects, others as primitive types*
    - *Should we make a class* `Address` *or use a* `String`*?*
  - *Not all classes can be discovered in analysis phase*
  - *Some classes may already exist*

CRC Card

- Describes a **c**lass, its **r**esponsibilities, and its **c**ollaborators
- Use an index card for each class
- Pick the class that should be responsible for each method (verb)
- Write the responsibility onto the class card

CRC Card

- Indicate what other classes are needed to fulfill responsibility (collaborators)



Self Check

Suppose the invoice is to be saved to a file. Name a likely collaborator.

Self Check

What do you do if a CRC card has ten responsibilities?

Relationships Between Classes

- Inheritance
- Aggregation
- Dependency

Inheritance

- *Is-a* relationship
- Relationship between a more general class (superclass) and a more specialized class (subclass)
- Every savings account is a bank account
- Every circle is an ellipse (with equal width and height)
- It is sometimes abused
  - *Should the class `Tire` be a subclass of a class `Circle`?*
    - *The has-a relationship would be more appropriate*

Aggregation

- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Use an instance variable
  - *A tire has a circle as its boundary:*

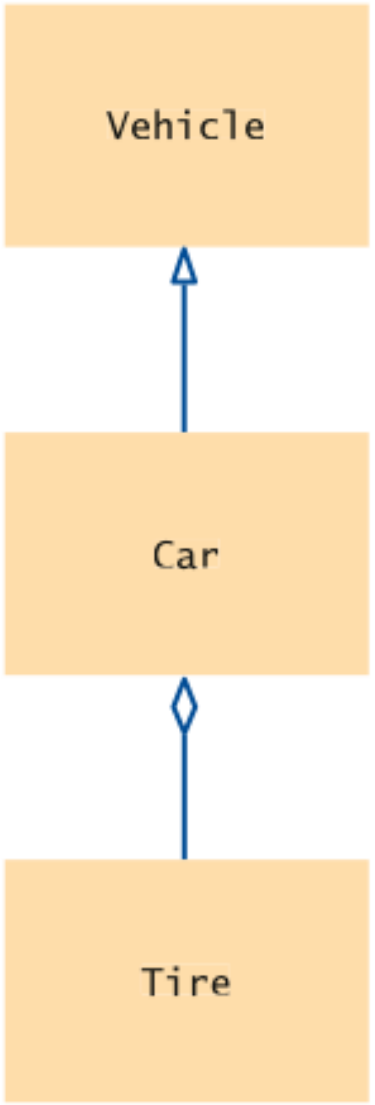
```
class Tire
{
   ...
   private String rating;
   private Circle boundary;
}
```

- Every car has a tire (in fact, it has four)

Example

```
class Car extends Vehicle
{
   ...
   private Tire[] tires;
}
```

- UML Notation for inheritance and aggregation



Dependency

- *Uses* relationship
- Example: Many of our applications depend on the `Scanner` class to read input
- Aggregation is a stronger form of dependency
- Use aggregation to remember another object between method calls

UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

Self Check

Consider the `Bank` and `BankAccount` classes in the previous example. How are they related?

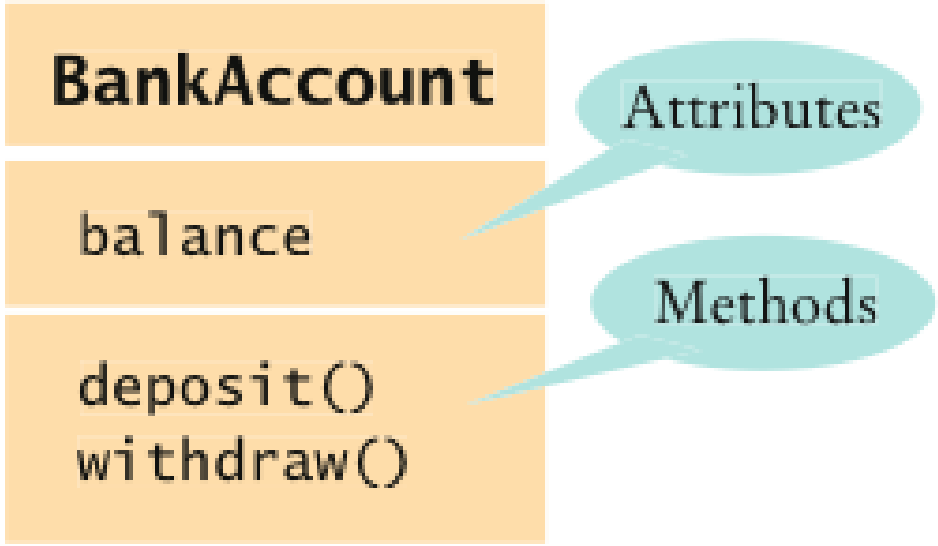
Self Check

Consider the `BankAccount` and `SavingsAccount` objects in the previous example. How are they related?

Self Check

Consider the `BankAccountTester` class in the previous example. Which classes does it depend on?

Attributes and Methods in UML Diagrams



Multiplicities

- any number (zero or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1

An Aggregation Relationship with Multiplicities

Aggregation and Association

- Association: More general relationship between classes
- Use early in the design phase
- A class is associated with another if you can navigate from objects of one class to objects of the other
- Given a `Bank` object, you can navigate to `Customer` objects

An Association Relationship

Five-Part Development Process

1. Gather requirements
2. Use CRC cards to find classes, responsibilities, and collaborators
3. Use UML diagrams to record class relationships
4. Use `javadoc` to document method behavior
5. Implement your program

Case Study: Printing an Invoice - Requirements

- Task: Print out an invoice
- Invoice: Describes the charges for a set of products in certain quantities
- Omit complexities
  - *Dates, taxes, and invoice and customer numbers*
- Print invoice
  - *Billing address, all line items, amount due*
- Line item
  - *Description, unit price, quantity ordered, total price*
- For simplicity, do not provide a user interface
- Test program: Adds line items to the invoice and then prints it

Case Study: Sample Invoice

I N V O I C E

Sam’s Small Appliances
100 Main Street
Anytown, CA 98765

Description	Price	Qty	Total
Toaster 29.95	3	89.85	
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98

AMOUNT DUE: $154.78

Case Study: Printing an Invoice - CRC Cards

- Discover classes
- Nouns are possible classes:

```
Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due
```

Case Study: Printing an Invoice - CRC Cards

- Analyze classes:

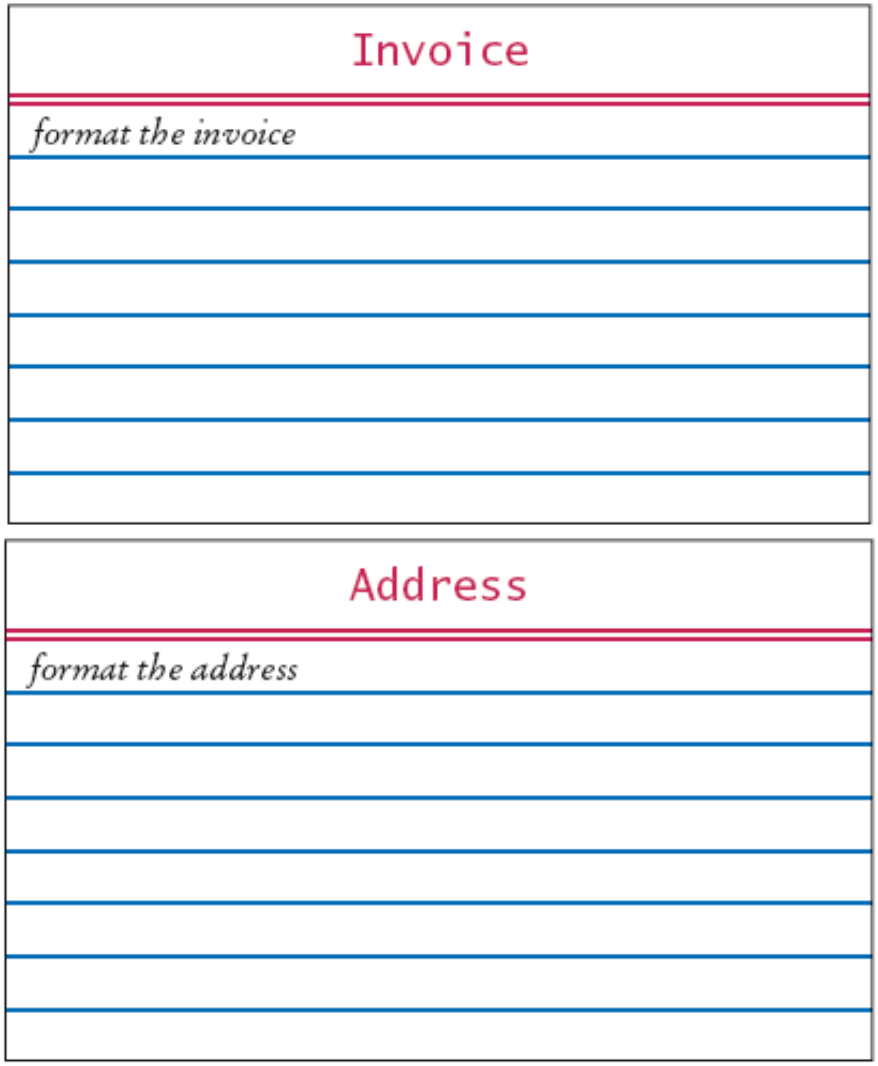
```
Invoice
Address
LineItem     // Records the product and the quantity
Product
Description  // variable of the Product class
Price        // variable of the Product class
Quantity     // Not an attribute of a Product
Total        // Computed - not stored anywhere
Amount Due   // Computed - not stored anywhere
```

- Classes after a process of elimination:

```
Invoice
Address
LineItem
Product
```

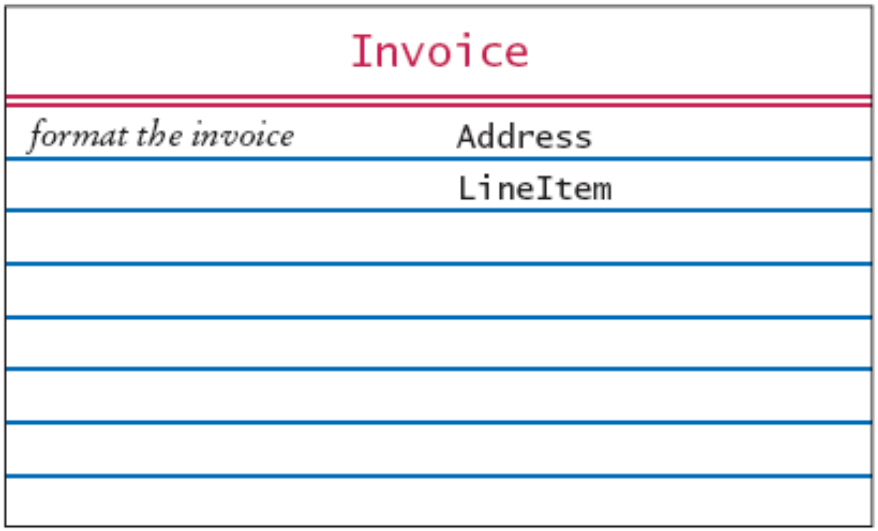
CRC Cards for Printing Invoice

`Invoice` and `Address` must be able to format themselves:



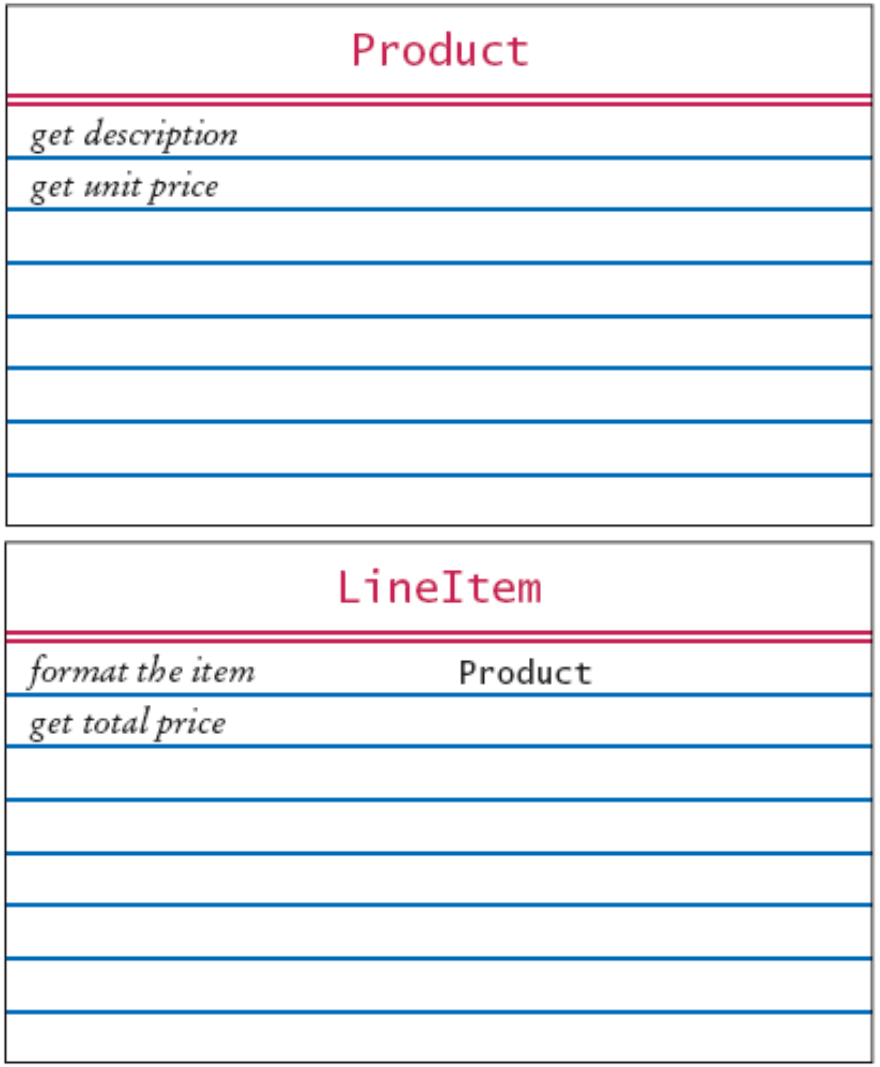
CRC Cards for Printing Invoice

Add collaborators to invoice card:



CRC Cards for Printing Invoice

`Product` and `LineItem` CRC cards:



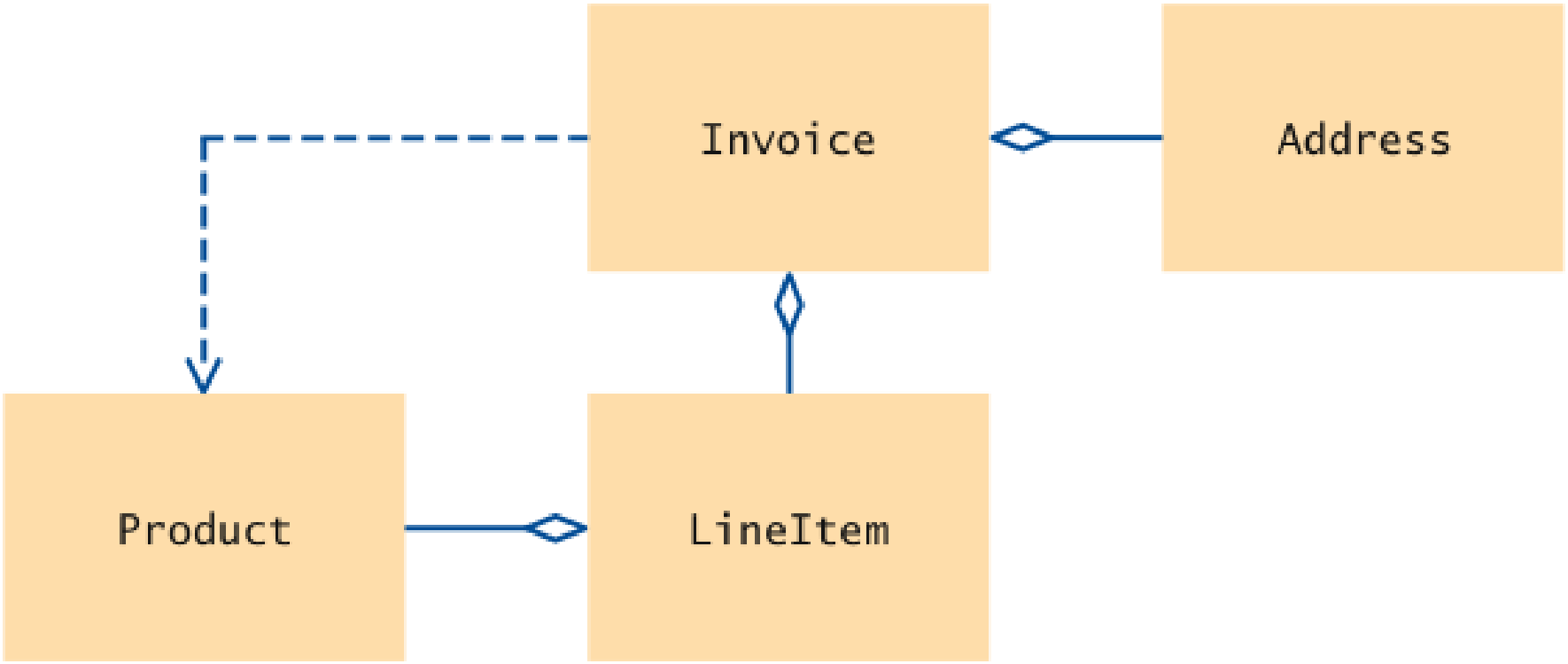
CRC Cards for Printing Invoice

Invoice must be populated with products and quantities:

Invoice	
format the invoice	Address
add a product and quantity	LineItem
	Product

Printing an Invoice - UML Diagrams

The relationships between the invoice classes



Printing an Invoice - Method Documentation

- Use `javadoc` documentation to record the behavior of the classes
- Leave the body of the methods blank
- Run `javadoc` to obtain formatted version of documentation in HTML format
- Advantages:
  - *Share HTML documentation with other team members*
  - *Format is immediately useful: Java source files*
  - *Supply the comments of the key methods*

Method Documentation - `Invoice` Class

```
/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   /**
      Adds a charge for a product to this invoice.
      @param aProduct the product that the customer
         ordered
      @param quantity the quantity of the product
   */
   public void add(Product aProduct, int quantity)
   {
   }
   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
   }
}
```

Method Documentation - `LineItem` Class

```
/**
   Describes a quantity of an article to purchase and its
   price.
*/
public class LineItem
{
   /**
      Computes the total cost of this line item.
      @return the total price
   */
   public double getTotalPrice()
   {
   }
   /**
      Formats this item.
      @return a formatted string of this line item
   */
   public String format()
   {
   }
}
```

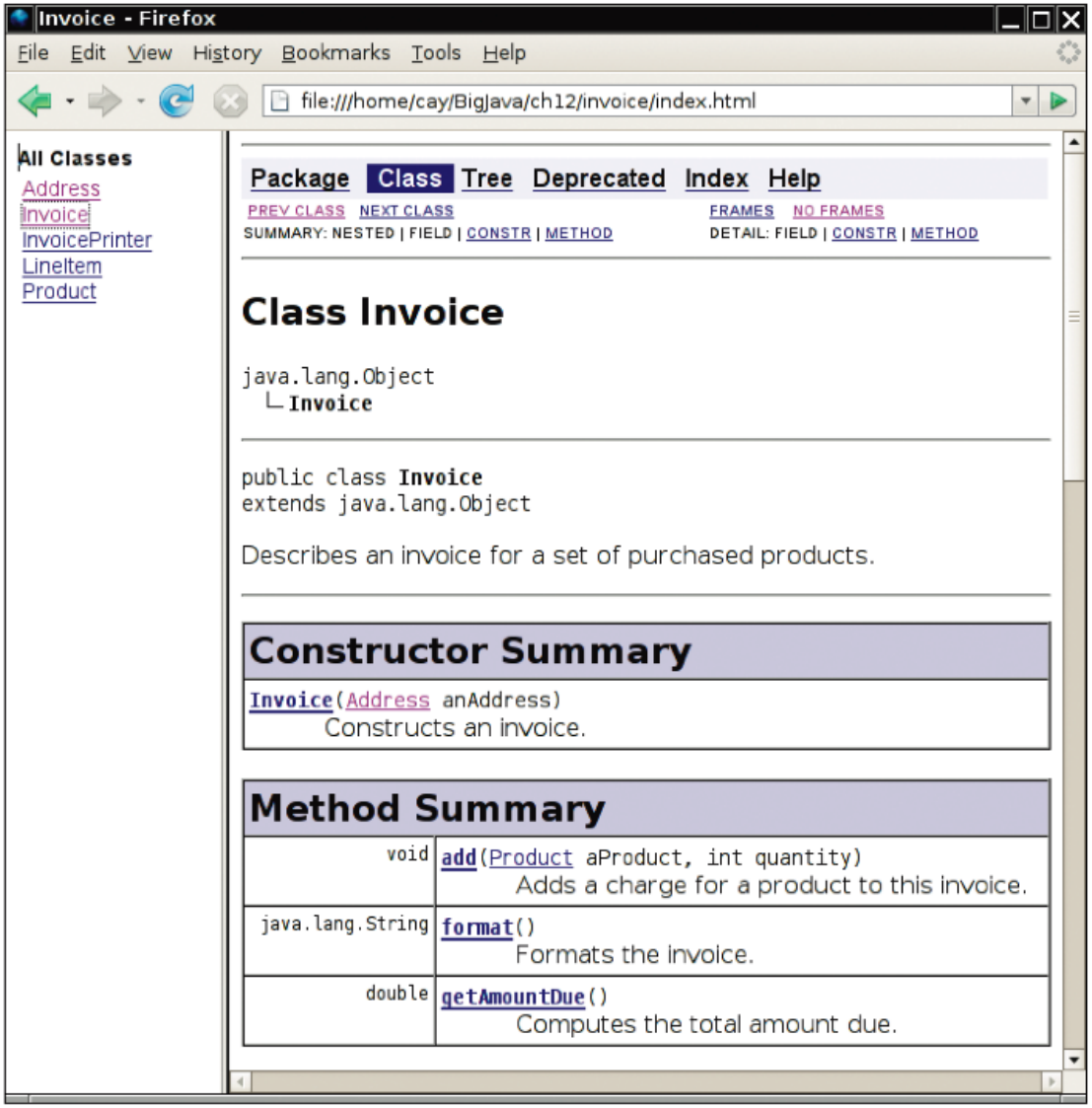
Method Documentation - `Product` Class

```
/**
   Describes a product with a description and a price.
*/
   public class Product
{
   /**
      Gets the product description.
      @return the description
   */
   public String getDescription()
   {
   }
   /**
      Gets the product price.
      @return the unit price
   */
   public double getPrice()
   {
   }
}
```

Method Documentation - `Address` Class

```
/**
Describes a mailing address.
*/
public class Address
{
   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
   }
}
```

The Class Documentation in the HTML Format



Printing an Invoice - Implementation

- The UML diagram will give instance variables
- Look for associated classes
  - *They yield instance variables*

Implementation

- `Invoice` aggregates `Address` and `LineItem`
- Every invoice has one billing address
- An invoice can have many line items:

```
public class Invoice
{
   ...
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

Implementation

A line item needs to store a `Product` object and quantity:

```
public class LineItem
{
   ...
   private int quantity;
   private Product theProduct;
}
```

Implementation

- The methods themselves are now very easy
- Example:
  - *`getTotalPrice` of `LineItem` gets the unit price of the product and multiplies it with the quantity:*

```
/**
   Computes the total cost of this line item.
   @return the total price
*/
public double getTotalPrice()
{
   return theProduct.getPrice() * quantity;
}
```

InvoicePrinter.java

```
1  /**
2     This program demonstrates the invoice classes by printing
3     a sample invoice.
4  */
5  public class InvoicePrinter
6  {
7     public static void main(String[] args)
8     {
9        Address samsAddress
10             = new Address("Sam&apos;s Small Appliances",
11                "100 Main Street", "Anytown", "CA", "98765");
12
13       Invoice samsInvoice = new Invoice(samsAddress);
14       samsInvoice.add(new Product("Toaster", 29.95), 3);
15       samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16       samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18       System.out.println(samsInvoice.format());
19    }
20 }
21
22
23
```

Invoice.java

```
 1  import java.util.ArrayList;
 2
 3  /**
 4     Describes an invoice for a set of purchased products.
 5  */
 6  public class Invoice
 7  {
 8     private Address billingAddress;
 9     private ArrayList<LineItem> items;
10
11     /**
12        Constructs an invoice.
13        @param anAddress the billing address
14     */
15     public Invoice(Address anAddress)
16     {
17        items = new ArrayList<LineItem>();
18        billingAddress = anAddress;
19     }
20
21     /**
22        Adds a charge for a product to this invoice.
23        @param aProduct the product that the customer ordered
24        @param quantity the quantity of the product
25     */
26     public void add(Product aProduct, int quantity)
27     {
28        LineItem anItem = new LineItem(aProduct, quantity);
29        items.add(anItem);
30     }
31
```

Invoice.java (cont.)

```
32      /**
33         Formats the invoice.
34         @return the formatted invoice
35      */
36      public String format()
37      {
38         String r =  "                        I N V O I C E\n\n"
39               + billingAddress.format()
40               + String.format("\n\n%-30s%8s%5s%8s\n",
41                  "Description", "Price", "Qty", "Total");
42
43         for (LineItem item : items)
44         {
45            r = r + item.format() + "\n";
46         }
47
48         r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
49
50         return r;
51      }
52
53      /**
54         Computes the total amount due.
55         @return the amount due
56      */
57      public double getAmountDue()
58      {
59         double amountDue = 0;
60         for (LineItem item : items)
61         {
62            amountDue = amountDue + item.getTotalPrice();
63         }
64         return amountDue;
65      }
66   }
```

LineItem.java

```
1  /**
2     Describes a quantity of an article to purchase.
3  */
4  public class LineItem
5  {
6     private int quantity;
7     private Product theProduct;
8
9     /**
10       Constructs an item from the product and quantity.
11       @param aProduct the product
12       @param aQuantity the item quantity
13    */
14    public LineItem(Product aProduct, int aQuantity)
15    {
16       theProduct = aProduct;
17       quantity = aQuantity;
18    }
19
20    /**
21       Computes the total cost of this line item.
22       @return the total price
23    */
24    public double getTotalPrice()
25    {
26       return theProduct.getPrice() * quantity;
27    }
28
29    /**
30       Formats this item.
31       @return a formatted string of this item
32    */
33    public String format()
34    {
35       return String.format("%-30s%8.2f%5d%8.2f",
36          theProduct.getDescription(), theProduct.getPrice(),
37          quantity, getTotalPrice());
38    }
39 }
```

Product.java

```
1  /**
2     Describes a product with a description and a price.
3  */
4  public class Product
5  {
6     private String description;
7     private double price;
8
9     /**
10       Constructs a product from a description and a price.
11       @param aDescription the product description
12       @param aPrice the product price
13    */
14    public Product(String aDescription, double aPrice)
15    {
16       description = aDescription;
17       price = aPrice;
18    }
19
20    /**
21       Gets the product description.
22       @return the description
23    */
24    public String getDescription()
25    {
26       return description;
27    }
28
29    /**
30       Gets the product price.
31       @return the unit price
32    */
33    public double getPrice()
34    {
35       return price;
36    }
37 }
38
```

Address.java

```
1  /**
2     Describes a mailing address.
3  */
4  public class Address
5  {
6     private String name;
7     private String street;
8     private String city;
9     private String state;
10    private String zip;
11
12    /**
13       Constructs a mailing address.
14       @param aName the recipient name
15       @param aStreet the street
16       @param aCity the city
17       @param aState the two-letter state code
18       @param aZip the ZIP postal code
19    */
20    public Address(String aName, String aStreet,
21          String aCity, String aState, String aZip)
22    {
23       name = aName;
24       street = aStreet;
25       city = aCity;
26       state = aState;
27       zip = aZip;
28    }
29
30    /**
31       Formats the address.
32       @return the address as a string with three lines
33    */
34    public String format()
35    {
36       return name + "\n" + street + "\n"
37             + city + ", " + state + " " + zip;
38    }
39 }
40
```

Self Check

Which class is responsible for computing the amount due? What are its collaborators for this task?

Self Check

Why do the format methods return `String` objects instead of directly printing to `System.out`?

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

PROGRAMMING IN JAVA

GENERIC PROGRAMMING

Chapter Goals

- To understand the objective of generic programming
- To be able to implement generic classes and methods
- To understand the execution of generic methods in the virtual machine
- To know the limitations of generic programming in Java

Generic Classes and Type Parameters

- **Generic programming:** creation of programming constructs that can be used with many different types
  - *In Java, achieved with type parameters or with inheritance*
  - *Type parameter example: Java's* `ArrayList` *(e.g.* `ArrayList<String>`*)*
  - *Inheritance example: e.g.* `LinkedList` *can store objects of any class*
- *Generic class*: declared with one or more type parameters
- A type parameter for `ArrayList` denotes the element type:

```
public class ArrayList<E>
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

Type Parameters

- Can be instantiated with class or interface type:

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

- Cannot use a primitive type as a type variable:

```
ArrayList<double> // Wrong!
```

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```

Type Parameters

- Supplied type replaces type variable in class interface
- Example: `add` in `ArrayList<BankAccount>` has type variable `E` replaced with `BankAccount`:

```
public void add(BankAccount element)
```

- Contrast with `LinkedList.add` in the previous example:

```
public void add(Object element)
```

Type Parameters Increase Safety

Type parameters make generic code safer and easier to read

Impossible to add a `String` into an `ArrayList<BankAccount>`

Can add a `String` into a `LinkedList` intended to hold bank accounts

```
ArrayList<BankAccount> accounts1 =
   new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList();
// Should hold BankAccount objects
accounts1.add("my savings");
// Compile-time error
accounts2.add("my savings");
// Not detected at compile time
. . .
BankAccount account = (BankAccount) accounts2.getFirst();
// Run-time error
```

Self Check

The standard library provides a class `HashMap<K, V>` with key type `K` and value type `V`. Declare a hash map that maps strings to integers.

Answer: `HashMap<String, Integer>`

Self Check

The binary search tree class is an example of generic programming because you can use it with any classes that implement the `Comparable` interface. Does it achieve genericity through inheritance or type variables?

Implementing Generic Classes

- Example: simple generic class that stores pairs of objects such as:

```
Pair<String, BankAccount> result =
   new Pair<String, BankAccount>("Harry Hacker",
      harrysChecking);
```

- Methods `getFirst` and `getSecond` retrieve first and second values of pair:

```
String name = result.getFirst();
BankAccount account = result.getSecond();
```

- Example of use: return two values at the same time (method returns a `Pair`)
- Generic `Pair` class requires two type parameters, one for each element type enclosed in angle brackets:

```
public class Pair<T, S>
```

Good Type Variable Names

Type Variable	Name Meaning
E	Element type in a collection
K	Key type in a map
V	Value type in a map
T	General type
S, U	Additional general types

Class Pair

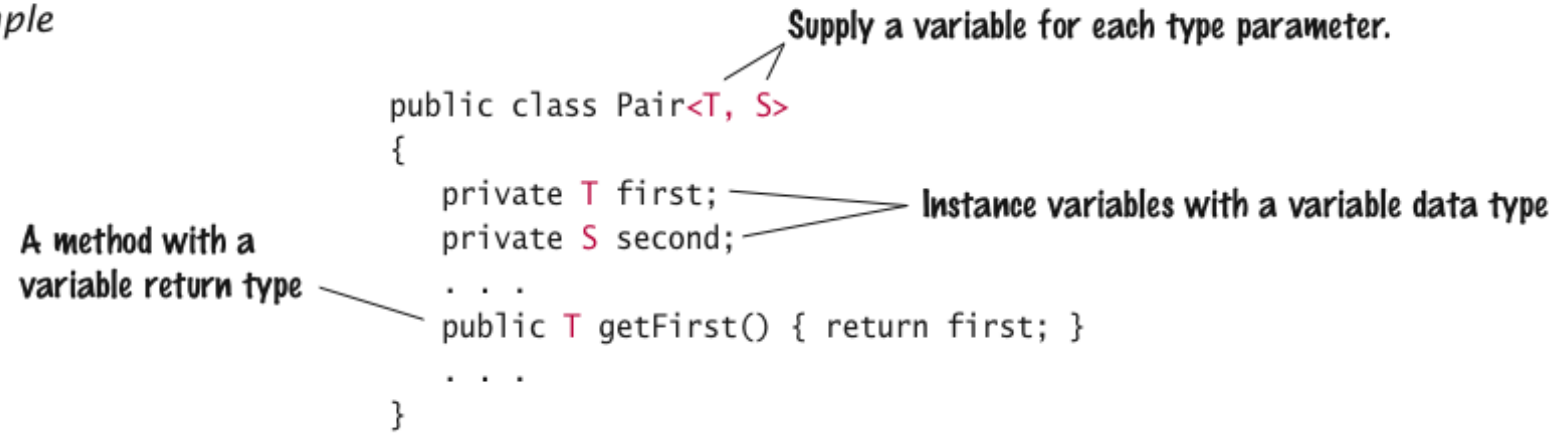
```
public class Pair<T, S>
{
   private T first;
   private S second;

   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
}
```

Syntax Declaring a Generic Class

Syntax

```
accessSpecifier class GenericClassName<TypeVariable1, TypeVariable2, . . .>
{
   instance variables
   constructors
   methods
}
```

Example



Pair.java

```
1  /**
2     This class collects a pair of elements of different types.
3  */
4  public class Pair<T, S>
5  {
6     private T first;
7     private S second;
8
9     /**
10       Constructs a pair containing two given elements.
11       @param firstElement the first element
12       @param secondElement the second element
13    */
14    public Pair(T firstElement, S secondElement)
15    {
16       first = firstElement;
17       second = secondElement;
18    }
19
20    /**
21       Gets the first element of this pair.
22       @return the first element
23    */
24    public T getFirst() { return first; }
25
26    /**
27       Gets the second element of this pair.
28       @return the second element
29    */
30    public S getSecond() { return second; }
31
32    public String toString() { return "(" + first + ", " + second + ")"; }
33 }
```

PairDemo.java

```
1  public class PairDemo
2  {
3     public static void main(String[] args)
4     {
5        String[] names = { "Tom", "Diana", "Harry" };
6        Pair<String, Integer> result = firstContaining(names, "a");
7        System.out.println(result.getFirst());
8        System.out.println("Expected: Diana");
9        System.out.println(result.getSecond());
10       System.out.println("Expected: 1");
11    }
12
13    /**
14       Gets the first String containing a given string, together
15       with its index.
16       @param strings an array of strings
17       @param sub a string
18       @return a pair (strings[i], i) where strings[i] is the first
19       strings[i] containing str, or a pair (null, -1) if there is no
20       match.
21    */
22    public static Pair<String, Integer> firstContaining(
23       String[] strings, String sub)
24    {
25       for (int i = 0; i < strings.length; i++)
26       {
27          if (strings[i].contains(sub))
28          {
29             return new Pair<String, Integer>(strings[i], i);
30          }
31       }
32       return new Pair<String, Integer>(null, -1);
33    }
34 }
```

Program Run:

```
Diana
Expected: Diana
1
Expected: 1
```

Self Check

How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?

Answer:

```
new Pair<String, String>("Hello", "World")
```

Self Check

What is the difference between an `ArrayList<Pair<String, Integer>>` and a `Pair<ArrayList<String>, Integer>`?

Answer: An `ArrayList<Pair<String, Integer>>` contains multiple pairs, for example `[(Tom, 1), (Harry, 3)]`. A `Pair<ArrayList<String>, Integer>` contains a list of strings and a single integer, such as `([Tom, Harry], 1)`.

Generic Methods

- **Generic method:** method with a type variable
- Can be defined inside non-generic classes
- Example: Want to declare a method that can print an array of any type:

```
public class ArrayUtil
{
   /**
      Prints all elements in an array.
      @param a the array to print
   */
   public <T> static void print(T[] a)
   {
      . . .
   }
   . . .
}
```

Generic Methods

Often easier to see how to implement a generic method by starting with a concrete example; e.g. print the elements in an array of *strings*:

```
public class ArrayUtil
{
   public static void print(String[] a)
   {
      for (String e : a)
         System.out.print(e + " ");
         System.out.println();
   }
   . . .
}
```

Generic Methods

- In order to make the method into a generic method:
  - *Replace* `String` *with a type parameter, say* `E`*, to denote the element type*
  - *Supply the type parameters between the method's modifiers and return type*

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Generic Methods

- When calling a generic method, you need not instantiate the type variables:

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

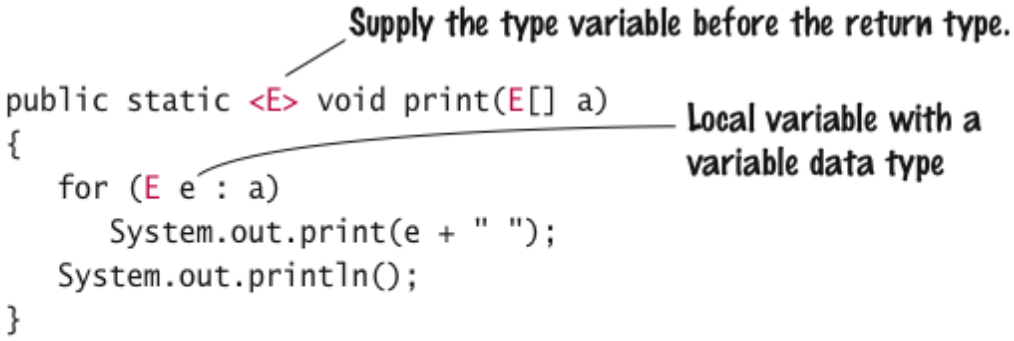
- The compiler deduces that `E` is `Rectangle`
- You can also define generic methods that are not static
- You can even have generic methods in generic classes
- Cannot replace type variables with primitive types
  e.g.: cannot use the generic `print` method to print an array of type `int[]`

Syntax Defining a Generic Method

Syntax

modifiers <*TypeVariable*$_1$, *TypeVariable*$_2$, . . . > *returnType methodName*(*parameters*)
{
   body
}

Example



Self Check

Exactly what does the generic print method `print` when you pass an array of `BankAccount` objects containing two bank accounts with zero balances?

Self Check

Is the `getFirst` method of the `Pair` class a generic method?

Constraining Type Variables

- Type variables can be constrained with bounds:

```
public static <E extends Comparable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

- Can call `min` with a `String[]` array but not with a `Rectangle[]` array
- `Comparable` bound necessary for calling `compareTo`
- Otherwise, `min` method would not have compiled

Constraining Type Variables

- Very occasionally, you need to supply two or more type bounds:
  ```
  <E extends Comparable & Cloneable>
  ```
- `extends`, when applied to type variables, actually means "extends or implements"
- The bounds can be either classes or interfaces
- Type variable can be replaced with a class or interface type

Self Check

How would you constrain the type parameter for a generic `BinarySearchTree` class?

Answer:

```
public class BinarySearchTree<E extends Comparable>
```

Self Check

Modify the min method to compute the minimum of an array of elements that implements the `Measurable` interface.

Answer:

```
public static <E extends Measurable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
   if (a[i].getMeasure() < smallest.getMeasure()) < 0)
      smallest = a[i];
   return smallest;
}
```

Wildcard Types

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with higher bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Wildcard Types

- `public void addAll(LinkedList<? extends E> other)`
  ```
  {
     ListIterator<E> iter = other.listIterator();
     while (iter.hasNext()) add(iter.next());
  }
  ```
- `public static <E extends Comparable<E>> E min(E[] a)`
- `public static <E extends Comparable<? super E>> E min(E[] a)`
- `static void reverse(List<?> list)`

  You can think of that declaration as a shorthand for

  ```
  static void <T> reverse(List<T> list)
  ```

Type Erasure

- The virtual machine erases type parameters, replacing them with their bounds or `Object`s
- For example, generic class `Pair<T, S>` turns into the following raw class:

```
public class Pair
{
   private Object first;
   private Object second;

   public Pair(Object firstElement, Object secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }
}
```

Type Erasure

- Same process is applied to generic methods:

```
public static Comparable min(Comparable[] a)
{
   Comparable smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

Type Erasure

- Knowing about raw types helps you understand limitations of Java generics
- For example, trying to fill an array with copies of default objects would be wrong:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERROR
}
```

- Type erasure yields:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Not useful
}
```

Type Erasure

- To solve this particular problem, you can supply a default type:

```
public static <E> void fillWithDefaults(E[] a,
   E defaultValue)
{
   for (int i = 0; i < a.length; i++)
      a[i] = defaultValue;
}
```

Type Erasure

- You cannot construct an array of a generic type:

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = new E[MAX_SIZE]; // Error
   }
}
```

- Because the array construction expression `new E[]` would be erased to `new Object[]`

Type Erasure

- One remedy is to use an array list instead:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   . . .
   public Stack()
   {
      elements = new ArrayList<E>(); // Ok
   }
}
```

Self Check

What is the erasure of the `print` method?

Answer:

```
public static void print(Object[] a)
{
   for (Object e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Self Check

Could the `Stack` example be implemented as follows?

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = (E[]) new Object[MAX_SIZE];
   }
   . . .
}
```

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

AGENDA

- Designing Classes
- Object-Oriented Design
- Generic Programming
- Stream & Binary Input / Output

PROGRAMMING IN JAVA

STREAM & BINARY INPUT / OUTPUT

Chapter Goals

- To become familiar with the concepts of text and binary formats
- To learn about encryption
- To understand when to use sequential and random file access
- To be able to read and write objects using serialization

Text and Binary Formats

- Two ways to store data:
  - *Text format*
  - *Binary format*

Text Format

- Human-readable form
- Sequence of characters
  - *Integer 12,345 stored as characters* `'1' '2' '3' '4' '5'`
- Use `Reader` and `Writer` and their subclasses to process input and output
- To read:

```
FileReader reader = new FileReader("input.txt");
```

- To write:

```
FileWriter writer = new FileWriter("output.txt");
```

Binary Format

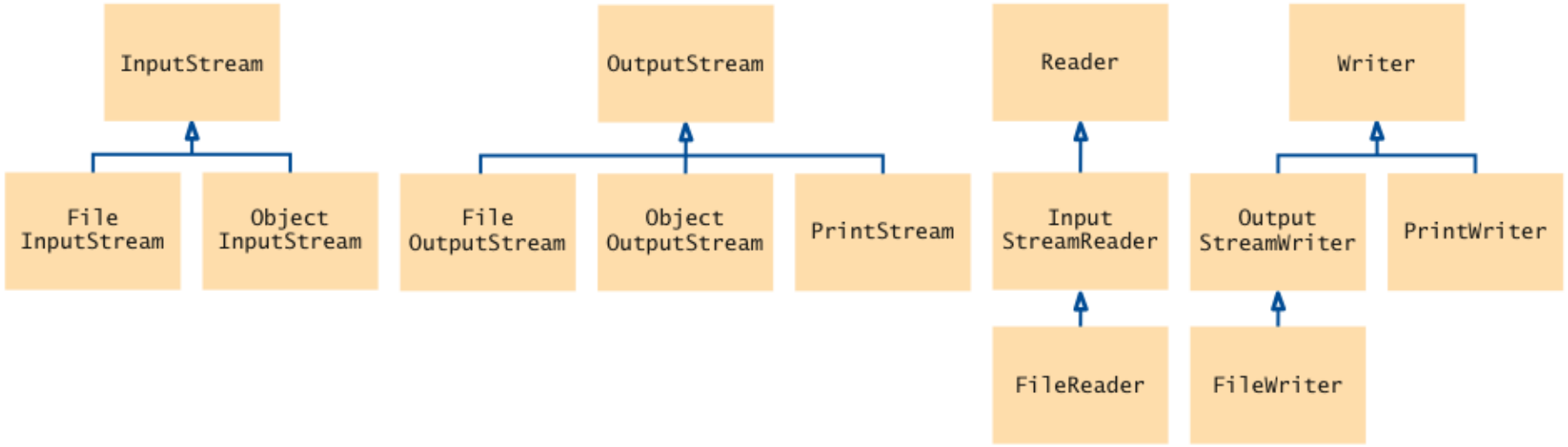
- Data items are represented in *bytes*
- Integer 12,345 stored as a sequence of four bytes `0 0 48 57`
- Use `InputStream` and `OutputStream` and their subclasses
- More compact and more efficient
- To read:

```
FileInputStream inputStream =
    new FileInputStream("input.bin");
```

- To write:

```
FileOutputStream outputStream =
    new FileOutputStream("output.bin");
```

Classes for Input/Output



`InputStream` and `OutputStream` Classes

- `InputStream` and `OutputStream` classes are responsible for input and output of bytes
- When constructing a `Scanner` from a `File` object, the `Scanner` automatically constructs a `FileReader`
- `System.out` is a `PrintStream` object

`Reader` and `Writer` Classes

- `Reader` and `Writer` classes are responsible for converting between bytes and characters
- Variation in how characters are represented as bytes
- Example Unicode encodings:

Character	UTF-8	UTF-16
'e'	69	0 69
'é'	195 169	0 223

Self Check

Suppose you need to read an image file that contains color values for each pixel in the image. Will you use a `Reader` or an `InputStream`?

Binary Input

- Use `read` method of `InputStream` class to read a single byte
  - *returns the next byte as an `int`*
  - *or the integer `-1` at end of file*

```
InputStream in = . . .;
int next = in.read();
byte b;
if (next != -1)
   b = (byte) next;
```

Binary Output

- Use `write` method of `OutputStream` class to write a single byte:

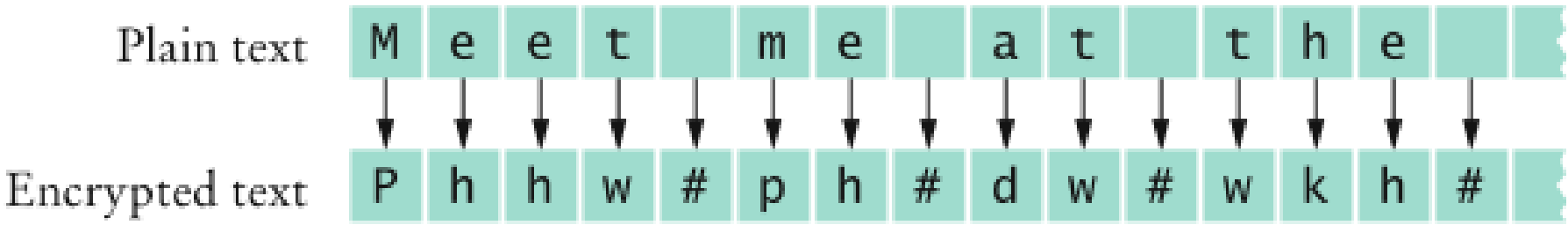
```
OutputStream out = . . .;
byte b = . . .;
out.write(b);
```

- When you are done writing to the file, you should close it:

```
out.close();
```

An Encryption Program

- File encryption
  - *To scramble it so that it is readable only to those who know the encryption method and secret keyword*
- To use Caesar cipher
  - *Choose an encryption key - a number between 1 and 25 that indicates the shift to be used in encrypting each byte*
  - *Example: If the key is 3, replace A with D, B with E, ...*



  - *To decrypt, use the negative of the encryption key*

To Encrypt Binary Data

```
int next = in.read();
if (next == -1)
   done = true;
else
{
   byte b = (byte) next;
   byte c = encrypt(b);
   out.write(c);
}
```

CaesarCipher.java

```
 1  import java.io.InputStream;
 2  import java.io.OutputStream;
 3  import java.io.IOException;
 4
 5  /**
 6     This class encrypts files using the Caesar cipher.
 7     For decryption, use an encryptor whose key is the
 8     negative of the encryption key.
 9  */
10  public class CaesarCipher
11  {
12     private int key;
13
14     /**
15        Constructs a cipher object with a given key.
16        @param aKey the encryption key
17     */
18     public CaesarCipher(int aKey)
19     {
20        key = aKey;
21     }
22
```

CaesarCipher.java (cont.)

```
23      /**
24         Encrypts the contents of a stream.
25         @param in the input stream
26         @param out the output stream
27      */
28      public void encryptStream(InputStream in, OutputStream out)
29            throws IOException
30      {
31         boolean done = false;
32         while (!done)
33         {
34            int next = in.read();
35            if (next == -1) done = true;
36            else
37            {
38               byte b = (byte) next;
39               byte c = encrypt(b);
40               out.write(c);
41            }
42         }
43      }
44
45      /**
46         Encrypts a byte.
47         @param b the byte to encrypt
48         @return the encrypted byte
49      */
50      public byte encrypt(byte b)
51      {
52         return (byte) (b + key);
53      }
54   }
```

JAVA

CaesarEncryptor.java

```
1  import java.io.File;
2  import java.io.FileInputStream;
3  import java.io.FileOutputStream;
4  import java.io.InputStream;
5  import java.io.IOException;
6  import java.io.OutputStream;
7  import java.util.Scanner;
8
9  /**
10    This program encrypts a file, using the Caesar cipher.
11 */
12 public class CaesarEncryptor
13 {
14    public static void main(String[] args)
15    {
16       Scanner in = new Scanner(System.in);
17       try
18       {
19          System.out.print("Input file: ");
20          String inFile = in.next();
21          System.out.print("Output file: ");
22          String outFile = in.next();\
23          System.out.print("Encryption key: ");
24          int key = in.nextInt();
25
26          InputStream inStream = new FileInputStream(inFile);
27          OutputStream outStream = new FileOutputStream(outFile);
28
29          CaesarCipher cipher = new CaesarCipher(key);
30          cipher.encryptStream(inStream, outStream);
31
32          inStream.close();
33          outStream.close();
34       }
35       catch (IOException exception)
36       {
37          System.out.println("Error processing file: " + exception);
38       }
39    }
40 }
```

Self Check

Why does the `read` method of the `InputStream` class return an `int` and not a `byte`?

Self Check

Decrypt the following message: `Khoor/#Zruog$`.

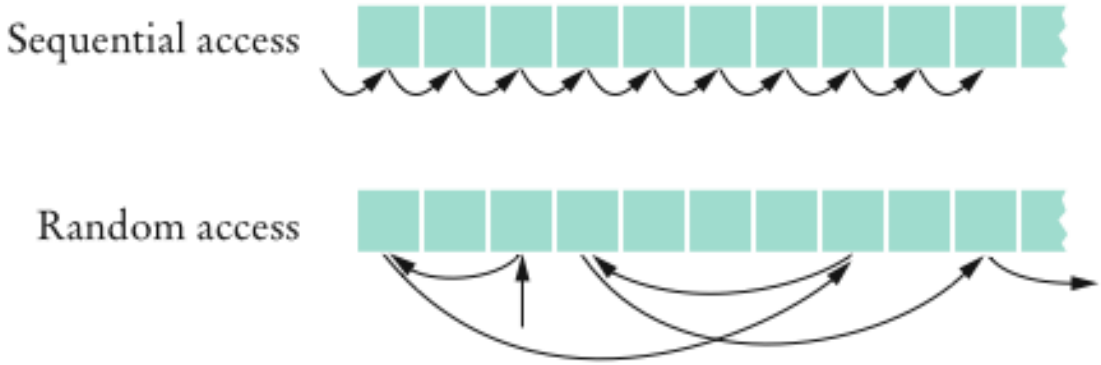
Answer: It is `"Hello, World!"`, encrypted with a key of `3`.

Self Check

Can you use the sample program from this section to encrypt a binary file, for example, an image file?

Random Access vs. Sequential Access

- Sequential access
  - *A file is processed a byte at a time*
  - *It can be inefficient*
- Random access
  - *Allows access at arbitrary locations in the file*
  - *Only disk files support random access*
    - `System.in` and `System.out` do not
  - *Each disk file has a special **file pointer** position*
    - You can read or write at the position where the pointer is



RandomAccessFile

- You can open a file either for
  - *Reading only (“r”)*
  - *Reading and writing (“rw”)*

```
RandomAccessFile f = new
RandomAcessFile("bank.dat","rw");
```

- To move the file pointer to a specific byte:

```
f.seek(n);
```

- To get the current position of the file pointer:

```
long n = f.getFilePointer();
// of type "long" because files can be very large
```

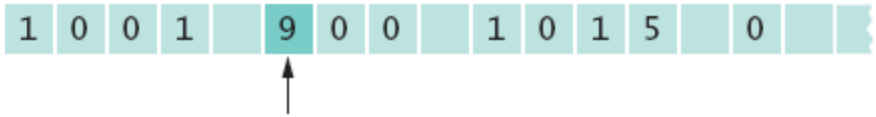
- To find the number of bytes in a file:

```
long fileLength = f.length();
```

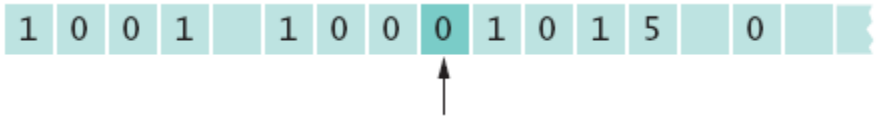
A Sample Program

- Use a random access file to store a set of bank accounts
- Program lets you pick an account and deposit money into it
- To manipulate a data set in a file, pay special attention to data formatting
  - *Suppose we store the data as text*
    Say account 1001 has a balance of $900, and account 1015 has a balance of 0

We want to deposit $100 into account 1001



If we now simply write out the new value, the result is



A Sample Program

- Better way to manipulate a data set in a file:
  - *Give each value a fixed size that is sufficiently large*
  - *Every record has the same size*
  - *Easy to skip quickly to a given record*
  - *To store numbers, it is easier to store them in binary format*

A Sample Program

- `RandomAccessFile` class stores binary data
- `readInt` and `writeInt` read/write integers as four-byte quantities
- `readDouble` and `writeDouble` use 8 bytes:

```
double x = f.readDouble();
f.writeDouble(x);
```

- To find out how many bank accounts are in the file:

```
public int size() throws IOException
{
   return (int) (file.length() / RECORD_SIZE);
         // RECORD_SIZE is 12 bytes:
         // 4 bytes for the account number and
         // 8 bytes for the balance
}
```

A Sample Program

- To read the n^{th} account in the file:

```
public BankAccount read(int n) throws IOException
{
   file.seek(n * RECORD_SIZE);
   int accountNumber = file.readInt();
   double balance = file.readDouble();
   return new BankAccount(accountNumber, balance);
}
```

A Sample Program

- To write the n^{th} account in the file:

```
public void write(int n, BankAccount account)
      throws IOException
{
   file.seek(n * RECORD_SIZE);
   file.writeInt(account.getAccountNumber());
   file.writeDouble(account.getBalance());
}
```

BankSimulator.java

```
1  import java.io.IOException;
2  import java.util.Scanner;
3
4  /**
5     This program demonstrates random access. You can access existing
6     accounts and deposit money, or create new accounts. The
7     accounts are saved in a random access file.
8  */
9  public class BankSimulator
10 {
11    public static void main(String[] args) throws IOException
12    {
13       Scanner in = new Scanner(System.in);
14       BankData data = new BankData();
15       try
16       {
17          data.open("bank.dat");
18
19          boolean done = false;
20          while (!done)
21          {
22             System.out.print("Account number: ");
23             int accountNumber = in.nextInt();
24             System.out.print("Amount to deposit: ");
25             double amount = in.nextDouble();
26
27             int position = data.find(accountNumber);
28             BankAccount account;
29             if (position >= 0)
30             {
31                account = data.read(position);
32                account.deposit(amount);
33                System.out.println("New balance: " + account.getBalance());
34             }
```

BankSimulator.java (cont.)

```
35                else // Add account
36                {
37                   account = new BankAccount(accountNumber, amount);
38                   position = data.size();
39                   System.out.println("Adding new account.");
40                }
41                data.write(position, account);
42
43                System.out.print("Done? (Y/N) ");
44                String input = in.next();
45                if (input.equalsIgnoreCase("Y")) done = true;
46             }
47          }
48          finally
49          {
50             data.close();
51          }
52       }
53    }
```

BankData.java

```
1  import java.io.IOException;
2  import java.io.RandomAccessFile;
3
4  /**
5     This class is a conduit to a random access file
6     containing savings account data.
7  */
8  public class BankData
9  {
10    private RandomAccessFile file;
11
12    public static final int INT_SIZE = 4;
13    public static final int DOUBLE_SIZE = 8;
14    public static final int RECORD_SIZE = INT_SIZE + DOUBLE_SIZE;
15
16    /**
17       Constructs a BankData object that is not associated with a file.
18    */
19    public BankData()
20    {
21       file = null;
22    }
23
24    /**
25       Opens the data file.
26       @param filename the name of the file containing savings
27       account information
28    */
29    public void open(String filename)
30          throws IOException
31    {
32       if (file != null) file.close();
33       file = new RandomAccessFile(filename, "rw");
34    }
35
```

BankData.java (cont.)

```
36      /**
37         Gets the number of accounts in the file.
38         @return the number of accounts
39      */
40      public int size()
41            throws IOException
42      {
43         return (int) (file.length() / RECORD_SIZE);
44      }
45
46      /**
47         Closes the data file.
48      */
49      public void close()
50            throws IOException
51      {
52         if (file != null) file.close();
53         file = null;
54      }
55
56      /**
57         Reads a savings account record.
58         @param n the index of the account in the data file
59         @return a savings account object initialized with the file data
60      */
61      public BankAccount read(int n)
62            throws IOException
63      {
64         file.seek(n * RECORD_SIZE);
65         int accountNumber = file.readInt();
66         double balance = file.readDouble();
67         return new BankAccount(accountNumber, balance);
68      }
69
```

BankData.java (cont.)

```
70      /**
71         Finds the position of a bank account with a given number
72         @param accountNumber the number to find
73         @return the position of the account with the given number,
74         or -1 if there is no such account
75      */
76      public int find(int accountNumber)
77            throws IOException
78      {
79         for (int i = 0; i < size(); i++)
80         {
81            file.seek(i * RECORD_SIZE);
82            int a = file.readInt();
83            if (a == accountNumber) // Found a match
84               return i;
85         }
86         return -1; // No match in the entire file
87      }
88
89      /**
90         Writes a savings account record to the data file
91         @param n the index of the account in the data file
92         @param account the account to write
93      */
94      public void write(int n, BankAccount account)
95            throws IOException
96      {
97         file.seek(n * RECORD_SIZE);
98         file.writeInt(account.getAccountNumber());
99         file.writeDouble(account.getBalance());
100     }
101  }
```

BankData.java (cont.)

Program Run:

```
Account number: 1001
Amount to deposit: 100
Adding new account.
Done? (Y/N) N
Account number: 1018
Amount to deposit: 200
Adding new account.
Done? (Y/N) N
Account number: 1001
Amount to deposit: 1000
New balance: 1100.0
Done? (Y/N) Y
```

Self Check

Why doesn't `System.out` support random access?

Self Check

What is the advantage of the binary format for storing numbers?
What is the disadvantage?

Object Streams

- `ObjectOutputStream` class can save a entire objects to disk
- `ObjectInputStream` class can read objects back in from disk
- Objects are saved in binary format; hence, you use streams

Writing a `BankAccount` Object to a File

The object output stream saves all instance variables:

```
BankAccount b = ...;
ObjectOutputStream out =
   new ObjectOutputStream(new
      FileOutputStream("bank.dat"));
out.writeObject(b);
```

Reading a `BankAccount` Object from a File

- `readObject` returns an `Object` reference
- Need to remember the types of the objects that you saved and use a cast:

```
ObjectInputStream in = new ObjectInputStream(new
   FileInputStream("bank.dat"));
BankAccount b =(BankAccount) in.readObject();
```

- `readObject` method can throw a `ClassNotFoundException`
- It is a checked exception - you must catch or declare it

Write and Read an `ArrayList` to a File

- Write:

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();
// Now add many BankAccount objects into a
out.writeObject(a);
```

- Read:

```
ArrayList<BankAccount> a =
   (ArrayList<BankAccount>) in.readObject();
```

Serializable

- Objects that are written to an object stream must belong to a class that implements the `Serializable` interface:

```
class BankAccount implements Serializable
{
   ...
}
```

- `Serializable` interface has no methods
- **Serialization:** Process of saving objects to a stream
  - *Each object is assigned a serial number on the stream*
  - *If the same object is saved twice, only serial number is written out the second time*
  - *When reading, duplicate serial numbers are restored as references to the same object*

SerialDemo.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.io.FileInputStream;
4  import java.io.FileOutputStream;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7
8  /**
9      This program demonstrates serialization of a Bank object.
10     If a file with serialized data exists, then it is
11     loaded. Otherwise the program starts with a new bank.
12     Bank accounts are added to the bank. Then the bank
13     object is saved.
14 */
15 public class SerialDemo
16 {
17    public static void main(String[] args)
18          throws IOException, ClassNotFoundException
19    {
20       Bank firstBankOfJava;
21
22       File f = new File("bank.dat");
23       if (f.exists())
24       {
25          ObjectInputStream in = new ObjectInputStream
26                (new FileInputStream(f));
27          firstBankOfJava = (Bank) in.readObject();
28          in.close();
29       }
30       else
31       {
32          firstBankOfJava = new Bank();
33          firstBankOfJava.addAccount(new BankAccount(1001, 20000));
34          firstBankOfJava.addAccount(new BankAccount(1015, 10000));
35       }
36
```

SerialDemo.java (cont.)

```
37          // Deposit some money
38          BankAccount a = firstBankOfJava.find(1001);
39          a.deposit(100);
40          System.out.println(a.getAccountNumber() + ": " + a.getBalance());
41          a = firstBankOfJava.find(1015);
42          System.out.println(a.getAccountNumber() + ": " + a.getBalance());
43
44          ObjectOutputStream out = new ObjectOutputStream
45                (new FileOutputStream(f));
46          out.writeObject(firstBankOfJava);
47          out.close();
48      }
49  }
```

Program Run:

First Program Run:

```
1001:20100.0
1015:10000.0
```

Second Program Run:

```
1001:20200.0
1015:10000.0
```

Self Check

Why is it easier to save an object with an `ObjectOutputStream` than a `RandomAccessFile`?

Self Check

What do you have to do to the `Coin` class so that its objects can be saved in an `ObjectOutputStream`?

THANK YOU FOR YOUR ATTENTION !