Designing Classes

- A class represents a single concept in the problem domain.
- Name for a class should be a noun that describes a concept, e.g. `BankAccount`, `Scanner`, etc.
- Utility classes – these classes only contain static methods and constants, e.g. `Math`. Do not create object instances of these classes.
- Do not turn actions into classes, e.g. deposit is an action, so it should correspond to a method.

Public interface of a class

- The public interface specifies what a programmer using the class can do with its objects.
- Headers of the `public` methods constitute the public interface of a class.
- Every method header contains the following parts:
  - An access specifier (`public, protected, private`), specifying which other methods can call this method. All methods in a program can call a `public` method, but `private` methods can only be called by other methods of the same class. A `protected` method can be called by other methods of classes in the same `package`.
  - The return type (remark: return type of a method is not part of the method signature)
  - The name of the method
  - A list of (explicit) parameter variables of the method, enclosed in parentheses.
  - You can have more than 1 method with the same name, but the list of explicit parameters in methods having the same name must be distinct.

- 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;
   ...
}
```

- The `CashRegister` class involves 2 concepts: cash register and coin. A better design is to make 2 classes:

```
public class Coin
{
   private double value;
   private String type;
   public Coin(double aValue, String name)
   {
      value = aValue;
      type = name;
   }

   public double getValue() { ...  }
   ...
}

public class CashRegister
{
   public void enterPayment(int count, Coin coin)
   {
      ...
   }
   ...
}
```

Instance variables

- An instance variable declaration consists of the following parts:
  - An access specifier (`public, private, protected`)
  - The type of the instance variable (e.g. `int, double, String`, etc.)
  - The name of the instance variable
- Each object of a class has its own set of instance variables.

- Instance variables are generally declared **`private`**. That specifier means that the variable can be accessed only by the methods of the same class, not by any method in other classes.

- For example, you can obtain a `Date` object and manipulate the object by calling the public methods in the `Date` class. You do not have direct access to the private instance variables of the `Date` object. Hence, the user of the `Date` object does not know the internal implementation details of the `Date` class.

- The process of hiding implementation details while providing a set of methods for working with objects is called encapsulation.

- Encapsulation is useful for
  - avoiding unnecessary dependencies among software modules
  - diagnosing errors
  - enhance software maintenance (you can modify the implementation of a class without affecting the other parts of the system as long as the public interface is not changed, or remains compatible).

Static variables

- A `static` variable belongs to the class (a global variable within the scope of the class), not belong to an object instance.
- A `static` variable can be declared `public` if it is a constant (declared final).

Example

```
public class MyClock
{
   public static final int MINUTES_PER_HOUR = 60;
   public static final int HOURS_PER_DAY = 24;
   ...

}
```

Class diagram in the Unified Modeling Language (UML)

Classes are represented with boxes that contain 3 compartments:

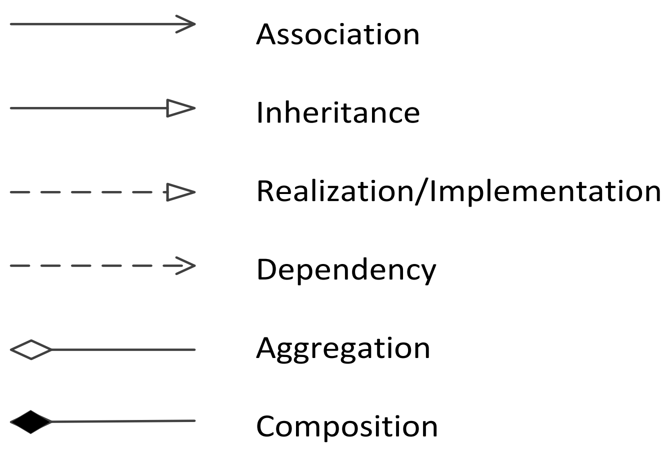
- Top compartment : name of the class
- Middle compartment : attributes of the class
- Bottom compartment : methods in the class

Visibility of a class attribute or method

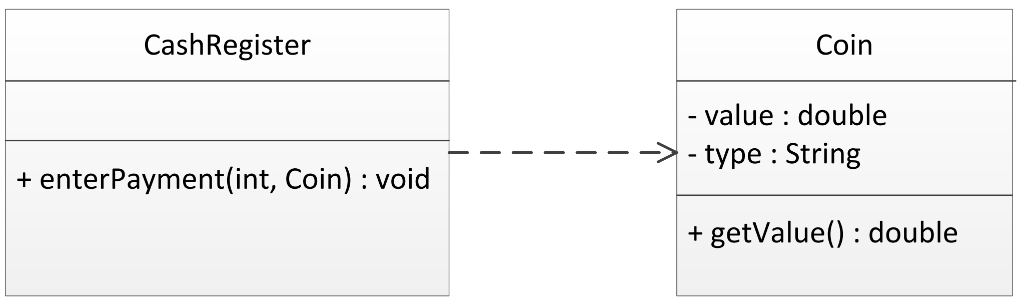
Symbol	Visibility
+	public
-	private
#	protected
/	derived (can be combined with one of the others)
~	package

Static member is represented by underlined name.

Relationship:



Example:



More details: https://en.wikipedia.org/wiki/Class_diagram

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`

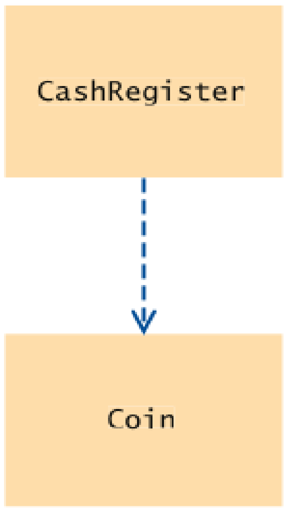


Figure 1
Dependency Relationship Between the CashRegister and Coin Classes

- High coupling = many class dependencies
- We can visualize relationships by class diagrams
- Minimize coupling to minimize the impact of interface changes

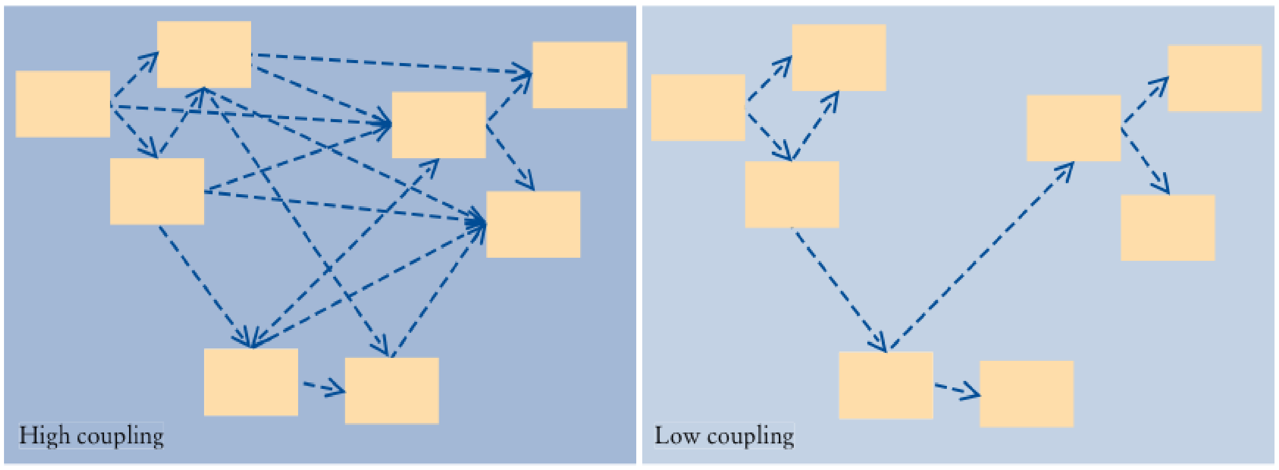


Figure 2 High and Low Coupling Between Classes

Implementing methods

Side Effects

- A side effect of a method is any kind of modification of data that is observable outside the method.
- Mutator methods have a side effect, namely the modification of the implicit parameter, i.e. changing the state value of the referenced object. Such a side effect is expected.
- If the method may modify any of the explicit parameters, it should be documented.

```
public class BankAccount
{
   ...

   /**
      Transfer money from this account to other account.
      @param amount the amount of money to transfer
      @param other the account into which to transfer the
             money
   */
   public void transfer(double amount,
                        BankAccount other)
   {
      this.balance -= amount;
      other.deposit(amount); // documented side effect

      // The object reference "this" refers to the implicit
      // parameter.
      // It is optional to include it in a statement.

      // Alternative implementation.
      // this.balance -= amount;
      // other.balance += amount;
   }
}
```

* Unexpected side effects should be avoided.

```
//Don't do that in your program

public class GradeBook
{
   ...

   /**
      Add student names to this grade book.
      @param studentNames a list of student names
   */
   public void addStudents(ArrayList<String>
                                      studentNames)
   {
      while (studentNames.size() > 0)
      {
         String name = studentNames.remove(0);
         Add name to gradebook;
      }
   }
   //This method modifies both the implicit parameter and
   //the explicit parameter studentNames.
   //Changes to studentNames are not documented, and
   //should be avoided.

}
```

Parameter passing in method calls

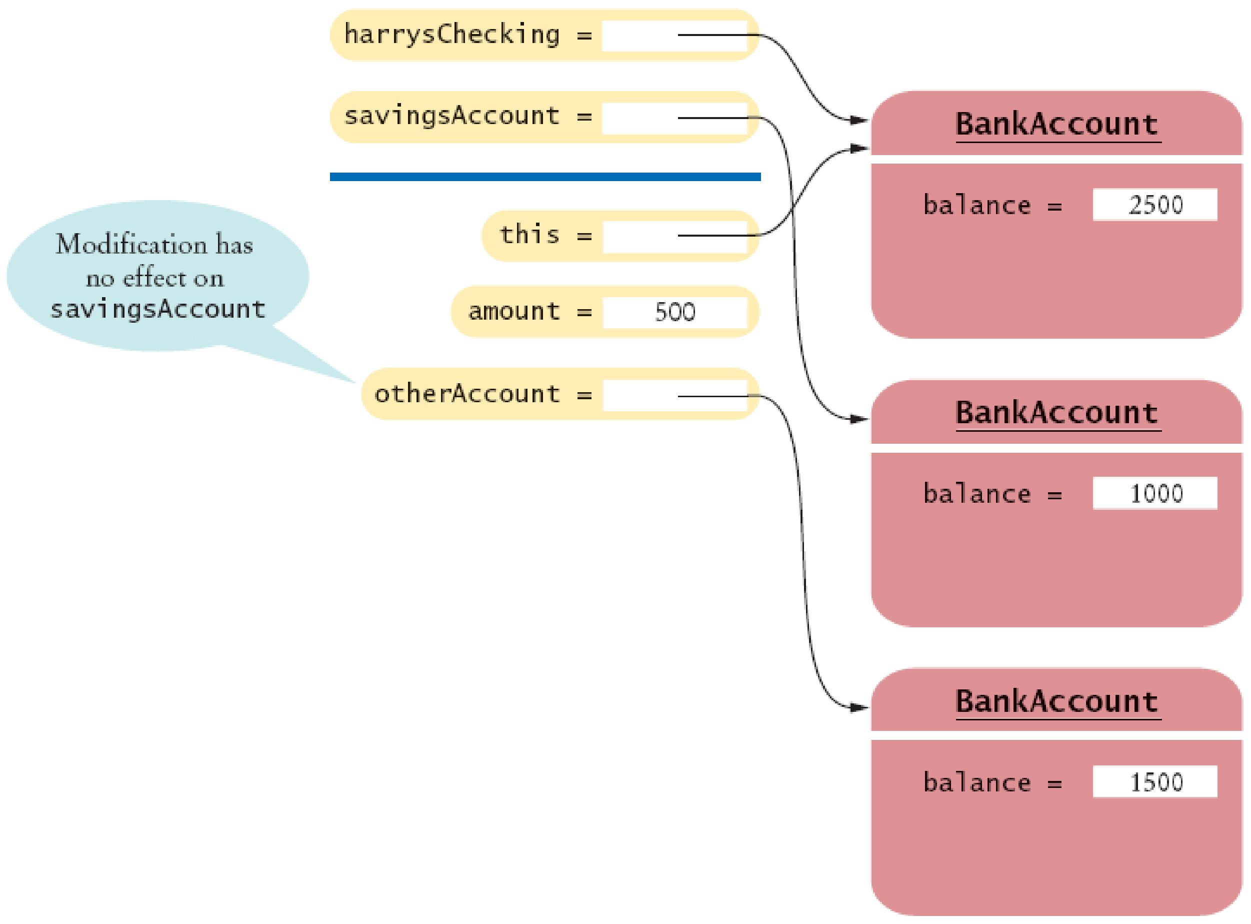
- In Java, explicit parameters are passed by value in method calls.

- General rules for using the explicit parameters in the body of a method
  - For primitive type parameter – do not modify its value inside the method.
  - For object reference parameter – you can update the state of the associated object via the object reference, but you cannot replace the contents (value) of an object reference.

```
//implementation that doesn't work
public class BankAccount
{
   private double balance;
   ...

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

//-----------------------------------------------------
//statements in method of other classes
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

Precondition and Postcondition

- A precondition is a requirement that the caller of a method must obey.
- For example, preconditions are typically provided for
  - restricting the parameters of a method
  - requiring that a method is only called when the object is in the appropriate state
- When a method is called in accordance with its preconditions, then the method promises to produce the postconditions, e.g. the return value is computed correctly, or the object is in a certain state after the method call is completed.

Example:

```
public class MyMath
{
   /**
      Compute the greatest common divisor of m and n.
      (Postcondition: return value is the gcd of m and n)
      @param m first input integer parameter
      @param n second input integer parameter
      (Precondition: n >= 0 and m > 0)
   */
   public static int gcd(int n, int m)
   {
      int r = n % m; //divide-by-zero exception if m == 0

      if (r == 0)
         return m;
      else
         return gcd(m, r);
   }
}
```

- The method is free to do anything if a precondition is not fulfilled.
- In the above `gcd()` method, if the input value of `m` is zero, the program will be terminated because of a Divide-by-zero exception.

What should a method actually do when it is called with inappropriate inputs?

There are three choices:

1. A method can check for the violation and throw an exception. Then the method does not return to its caller; instead, control is transferred to an exception handler. If no handler is present, then the program terminates. The exception handling mechanism will be discussed later.

2. A method can skip the check and work under the assumption that the preconditions are fulfilled. If they aren't, then any data corruption or other failures are the caller's fault. Debugging of the program may be difficult.

3. Make use of the assertion checking mechanism.

```
public static int gcd(int n, int m)
{
   assert m > 0;
   assert n >= 0;

   int r = n % m;

   if (r == 0)
      return m;
   else
      return GCD(m, r);
}
```

- An assertion is a condition that you believe to be true at all times in a particular program location.
- An assertion-check tests whether an assertion is true.
- When the assertion is correct, no harm is done, and the program works in the normal way. If the assertion fails, and assertion checking is enabled, then the program terminates with an `AssertionError`. This can be useful for diagnosing errors.

Class Invariants

- A class invariant is a statement about an object that is true after every constructor, and that is preserved by every mutator.
- We can use the class invariant to reason about the correctness of our program.

Example: Set of integers

- Suppose we represent a set of integers using an array.
- Requirements of the physical representation (invariant properties)
  1. If there are `n` elements in the set, then the elements are stored in the front portion of the array occupying index locations `0` to `n-1`.
  2. All the elements in the set must be distinct, i.e. no duplication is allowed.
  3. To facilitate efficient implementation of various set operations, elements in the array are arranged in ascending order.

```
public class IntSet
{
   private int[] element;
   private int size;  // logical size, no. of elements
   private final int DEFAULT_SIZE = 10;

   public IntSet()  //constructor
   {
      element = new int[DEFAULT_SIZE];
      size = 0;
   }

   private IntSet(int[] x, int s)  // private constructor
   {  // precondition: x[] is sorted in ascending order
      element = Arrays.copyOf(x, x.length);
      size = s;
   }

   public int size()  //accessor
   {  return size;
   }

   public boolean isEmpty()  //accessor
   {  return size == 0;
   }
```

```
public boolean contains(int e) //accessor
{
   int k = Arrays.binarySearch(element, 0, size, e);
   return k >= 0;
}

public void add(int e) //mutator
{
   if (this.contains(e))
      return;  //e is already a member of the set

   if (element.length == size)
      element = Arrays.copyOf(element, 2*element.length);

   //needs to maintain the elements in ascending order
   int i;
   for (i = size-1; i >= 0 && element[i] > e; i--)
      element[i+1] = element[i];

   element[i+1] = e;
   size++;
}
```

```
   public IntSet union(IntSet other) //accessor
   {
      //compute this union other
      int[] c = new int[this.size + other.size];
      int i = 0, j = 0, k = 0;

      while (i < this.size && j < other.size)
      {
         if (this.element[i] <= other.element[j])
         {
            if (this.element[i] == other.element[j])
               j++;

            c[k++] = this.element[i++];
         }
         else
            c[k++] = other.element[j++];
      }

      while (i < this.size)
         c[k++] = this.element[i++];

      while (j < other.size)
         c[k++] = other.element[j++];

      if (k > 0)
         return new IntSet(c, k);
      else
         return new IntSet();
   }

   //other methods
}
```

Remark:
The constructor `IntSet(int[] x, int s)` is defined `private` because it is difficult to validate the precondition if it can be invoked by other methods.

Packages

- A Java program consists of a collection of classes.
- A Java package is a set of related classes.
- Packages can also help to avoid name conflicts.

Package	**Purpose**	**Sample Class**
java.lang	Language support	Math
java.util	Utilities	Arrays
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

- In addition to the named packages, there is a special package, called the *default package*, which has no name.
- If you did not include any package statement at the top of your source file, its classes are placed in the default package.

- If you want to use a class from a package, you can
    - refers to it by its full name
      ```
      java.util.Scanner in = new java.util.Scanner(System.in);
      ```
    - Alternatively, you can import the name with an import statement
      ```
      import java.util.Scanner;
      Scanner in = new Scanner(System.in);
      ```

- A source file must be located in a subdirectory that matches the package name.
- For example, the source files for classes in the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`.
- You place the subdirectory inside the base directory bolding your program's file.
- If you working directory is `/home/Britney/hw8/problem1`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/Britney/hw8/problem1/com/horstmann/bigjava`

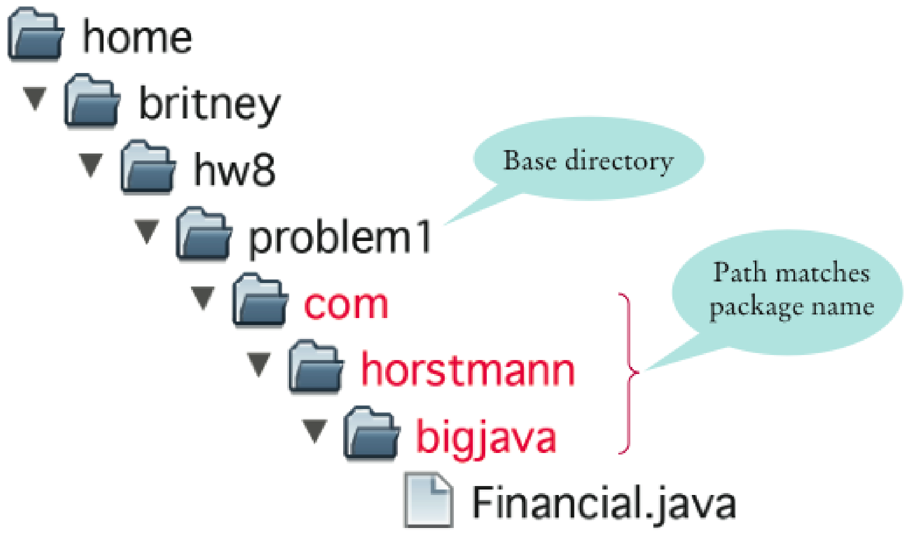


Figure 5
Base Directories and Subdirectories for Packages

- If a class, field, or method has no `public` or `private` modifier, then all methods of classes in the same package can access the feature.
- Package access is reasonable default for classes, but it is a potential security risk for methods and instance variables.

Inheritance

- In the real world, concepts are often grouped into hierarchies.
- In Java it is equally common to group classes in inheritance hierarchies.
- The classes representing the most general concepts are near the root, more specialized classes towards the branches.

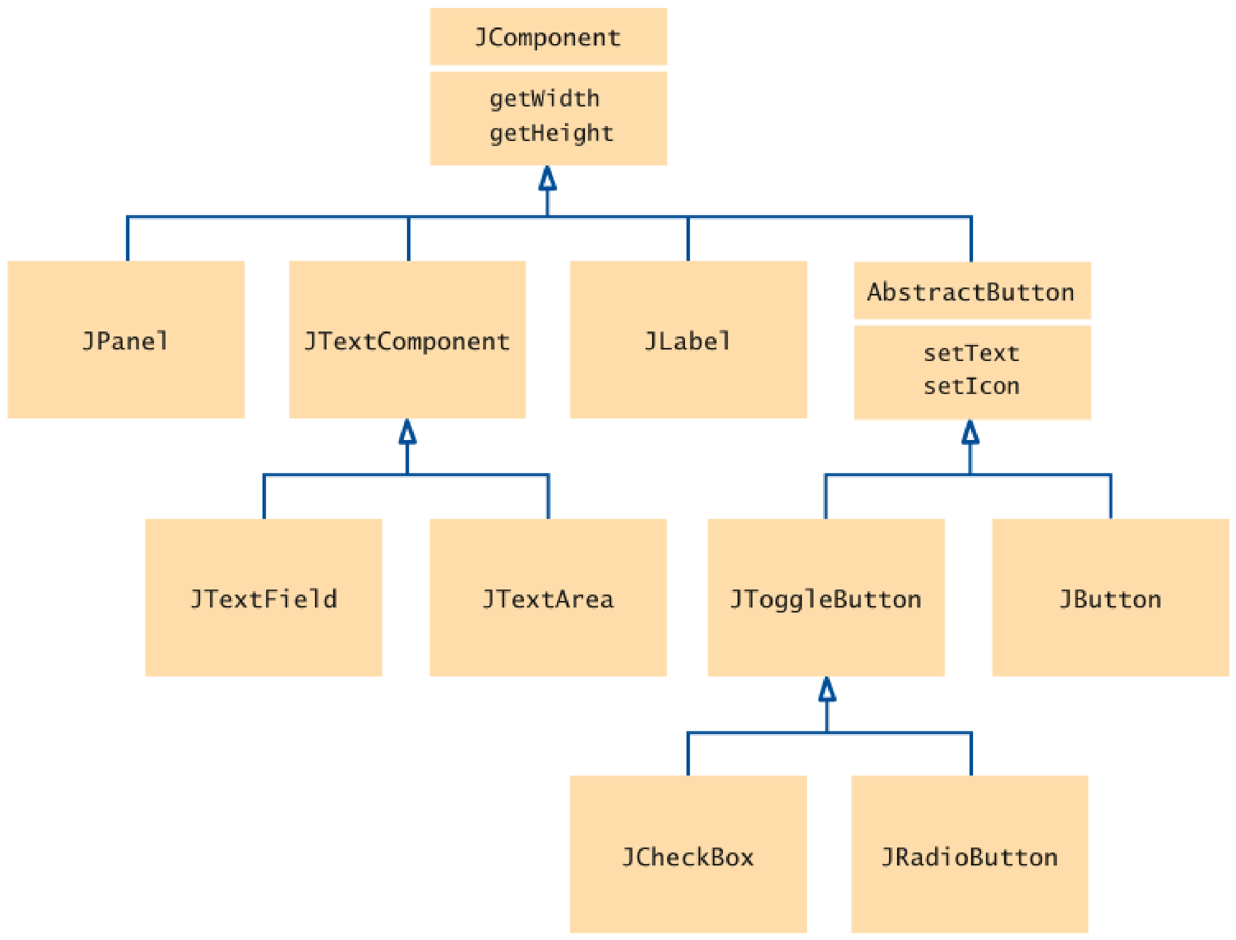


Figure 2 A Part of the Hierarchy of Swing User Interface Components

- The more general class is called the superclass (or base class)
- The more specialized class that inherits from the superclass is called the subclass (or derived class)
- In the above inheritance hierarchy, `JPanel` is a subclass of `JComponent`.
- In Java, all classes (except class `Object`) are subclass of class `Object`.

Example: Bank accounts

There can be different account types:

1. *Checking account:*
    - No interest
    - Small number of free transactions per month
    - Charges transaction fee for additional transactions
2. *Savings account:*
    - Earns interest that compounds monthly

We can have a more general class (Superclass): `BankAccount`, and more specialized classes (Subclasses): `CheckingAccount` & `SavingsAccount`

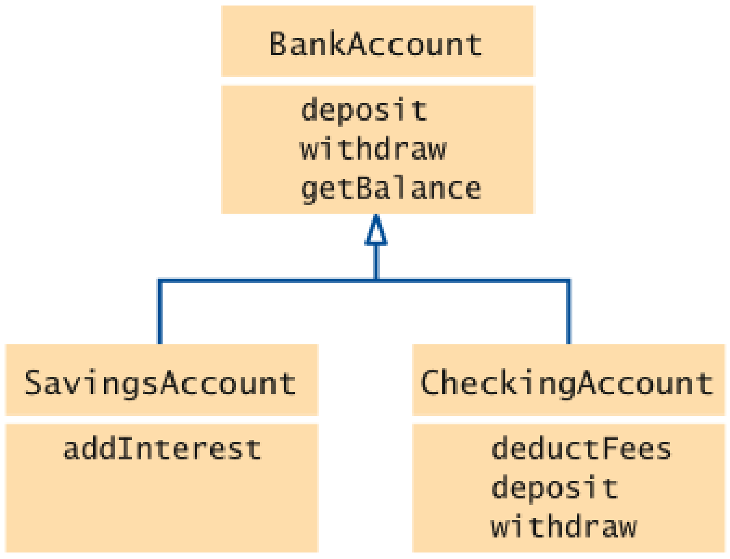


Figure 3 Inheritance Hierarchy for Bank Account Classes

Behavior of account classes:

- All support `getBalance` method
- Also support `deposit` and `withdraw` methods, but implementation details differ
- Checking account needs a method `deductFees` to deduct the monthly fees and to reset the transaction counter
- Checking account must override `deposit` and `withdraw` methods to count the transactions

Recall the implementation details of the `BankAccount` class

```
public class BankAccount
{
   private double balance;

   public BankAccount() //default constructor
   {  balance = 0;
   }

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

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

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

   public void transfer(double amount, BankAccount other)
   {
      this.withdraw(amount);
      other.deposit(amount);
   }

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

Inheritance is a mechanism for extending existing classes by adding instance variables and methods.

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;

   public SavingsAccount(double rate)  //constructor
   {
      //the default constructor of the superclass is
      //invoked to initialize the balance to zero
      super();
      interestRate = rate;
   }

   public SavingsAccount(double rate, double initialBalance)
   {
      //call the superclass constructor to set the balance
      //subclass has no access to private instance variables
      //of its superclass
      super(initialBalance);
      interestRate = rate;
   }


   public void addInterest() //additional method
   {
      //call the getBalance method of the superclass
      //to obtain the account balance
      double interest = getBalance() * interestRate / 100;

      //call the deposit method of superclass to update
      //the balance
      deposit(interest);
   }
}
```

```
public class CheckingAccount extends BankAccount
{
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
   private int transactionCount;

   public CheckingAccount(double initialBalance)
   {
      super(initialBalance);
      transactionCount = 0;
   }

   //override the withdraw method
   @Override
   public void withdraw(double amount)
   {
      transactionCount++;

      //call superclass withdraw method to update balance
      super.withdraw(amount);
   }

   //override the deposit method
   @Override
   public void deposit(double amount)
   {
      transactionCount++;
      super.deposit(amount);
   }

   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE *
            (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
}
```

Polymorphism

- "Polymorphism" comes from the Greek words for "many shapes".

- In Java, the term "Polymorphism" is used to refer to the fact that the same pieces of codes can have different behaviors depending on the actual object type during run-time.

Consider the codes below:

```
CheckingAccount myAccount = new CheckingAccount(0);
SavingsAccount momsSavings = new SavingsAccount(5000.0, 0.01);

double transferAmount = 200.0;

monsSavings.transfer(transferAmount, myAccount);


/* Method in BankAccount

   public void transfer(double amount, BankAccount other)
   {
      this.withdraw(amount);   // which version of withdraw
      other.deposit(amount);   // & deposit is executed ??
   }
*/
```

- The Java virtual machine locates the correct method by first looking at the class of the actual object, and then calling the method with the given name in that class. This mechanism is called *dynamic method lookup* (or *dynamic binding*).

Converting between subclass and superclass types

```
SavingsAccount collegeFund = new SavingsAccount(10);
collegeFund.deposit(10000);

BankAccount anAccount = collegeFund;
Object anObject = collegeFund;

SavingsAccount acc2 = anAccount; // error, not compatible
```

Variable of superclass can reference an object instance of subclass.
Variable of subclass cannot be used to reference an object instance of superclass.

The three object references stored in `collegeFund`, `anAccount`, and `anObject` all refer to the same object of type `SavingsAccount`

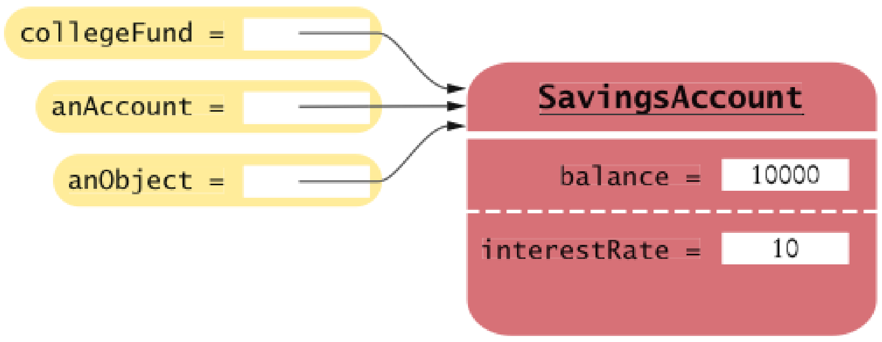


Figure 6
Variables of Different Types Can Refer to the Same Object

You can only reference the features in the public interface of `BankAccount` through the object reference of type `BankAccount`

```
anAccount.deposit(1000); // OK
anAccount.addInterest(); // Error--not a method of the class
                         // to which anAccount belongs
```

Now consider the statements

```
CheckingAccount myAccount = new CheckingAccount(1000);
myAccount.transfer(500, collegeFund);
/*
  The transfer() method is defined in class BankAccount.
  When the statement this.withdraw(500) in transfer() method
  is executed, which version of the withdraw() method (the
  one in BankAccount, or the one in CheckingAccount) will be
  used?
*/
```

Occasionally you need to convert from a superclass reference to a subclass reference

```
BankAccount anAccount = (BankAccount) anObject;
```

This cast is dangerous: If you are wrong, an exception is thrown

Use the **`instanceof`** operator to tests whether an object belongs to a particular type.

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    ...
}
else
{
   //take other actions
}
```

Multiple inheritances

- Java does not allow multiple inheritances.
- A class can only extend one superclass.
- But a class can implements multiple interfaces.

Abstract Classes

- When you extend an existing class, you have the choice whether or not to override the methods of the superclass.
- Sometimes, it is desirable to force programmers to override a method. That happens when there is no good default implementation in the superclass, and only the subclass programmer can know how to implement the method properly.
- You can define an **`abstract`** method in a class. An `abstract` method has no implementation (not the same as an empty implementation).
- A class with `abstract` method is called an `abstract` class. You cannot construct objects of `abstract` class.
- A class that declares an `abstract` method, or that inherits an `abstract` method without overriding it, must be declared as `abstract`.

Example:

```
public abstract class AbsBankAccount
{
   public abstract void deductFees();
   ...
}


public class SavingsAccount extends AbsBankAccount
{
   public void deductFees()
   {
      //must provide an implementation of the method
   }

   ...
}
```

Protected Access

- `Protected` features can be accessed by all subclasses and by all classes in the same package
- Solves the problem that `CheckingAccount` methods need access to the balance instance variable of the superclass `BankAccount`:

```
public class BankAccount
{
    . . .
   protected double balance;

    . . .
}
```

- The designer of the superclass has no control over the authors of subclasses:
  - Any of the subclass methods can corrupt the superclass data
  - Classes with protected instance variables are hard to modify — the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them

- `Protected` data can be accessed by all methods of classes in the same package

- It is best to leave all data `private` and provide accessor methods for the data

The `clone` method

The `clone` method is defined in the `class Object` with **`protected`** access

The `clone` method is used to make a copy of the given object.

```
BankAccount clonedAccount = (BankAccount)account.clone();
```

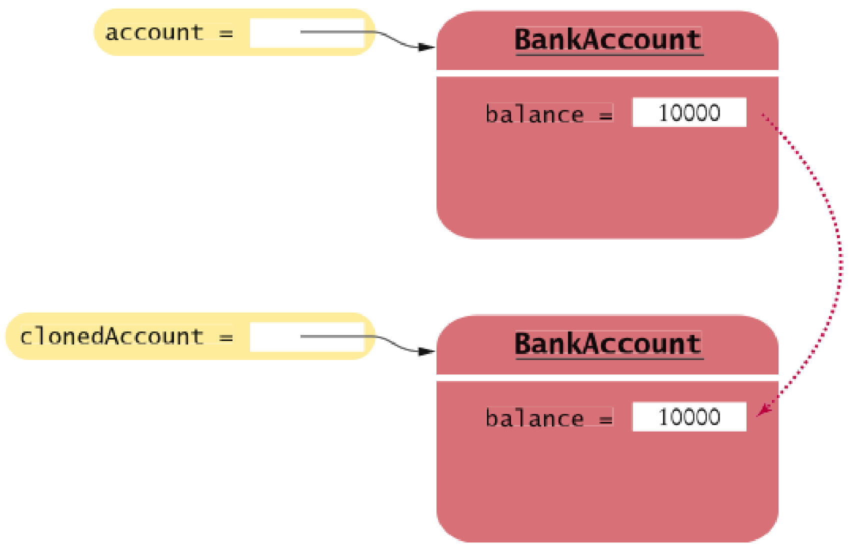


Figure 10
Cloning Objects

- Does not systematically clone all sub-objects
- Must be used with caution
- It is declared as `protected`; prevents from accidentally calling `x.clone()` if the class to which `x` belongs hasn't redefined clone to be `public`
- You should override the clone method (telling the system how to clone the object) with care