Chapter 9

Programming to Interfaces

When we write a class that matches a specification, we say that the class implements *the specification. In this chapter, we learn four Java constructions for designing classes and connecting them to their collaborating classes:*

- *Java* `interface`*s, which define specifications that a coded class must* implement*.*
- *Java inheritance (*`extends`*), which defines a new class by adding methods onto an already written class;*

- *Java* `abstract class`*es, which are "incomplete classes," part specification and part implementation.*
- *Java* `package`*s, which group a collection of classes under one name.*

This chapter's title comes from a slogan that practicing programmers follow when they build an application:

Program to the interface, not the implementation!

That is, when you write a class that depends on another, collaborator class, you should rely on the collaborator's interface—its specification—and not its coding (its implementation). With this approach, you can design and implement classes separately yet ensure that the assembled collection of classes collaborate successfully.

9.1 Why We Need Specifications

A program is assembled from a collection of classes that must "work together" or "fit together." What does it mean for two classes to fit together? For some insight, consider the following situation.

Say that you receive a portable disc player as a gift. When you try to operate the player, nothing happens — the player requires batteries. What batteries fit into the player? Fortunately, on the back of the player is the specification, "This player requires two AA batteries." With this information, you can obtain the correctly sized components (the batteries) and fit them into the player. The completed "assembly" operates.

The specification of the disc player's batteries served several useful purposes:

- The specification told the user which component must be fitted to the player to ensure correct operation.
- The specification told the manufacturer of the disc player what size to build the player's battery chamber and what voltage and amperage to use within the player's electronics.
- The specification told the battery manufacturer what size, voltage, and amperage to build batteries so that others can use them.

These three facts are important in themselves, but they also imply that *the user, the disc manufacturer, and the battery manufacturer need not communicate directly with each other* — the specification of the battery is all that is needed for each party to perform its own task independently of the other two.

Without size specifications of items like batteries, clothing, and auto parts, everyday life would would be a disaster.

When we assemble a program from components, the components must fit together. When a class, `A`, invokes methods from an object constructed from class `B`, class `A` assumes that `B` possesses the methods invoked and that the methods behave in some expected way — in the way they are *specified*. Just like batteries have specifications (e.g., sizes AAA, AA, C,...), classes have specifications also. This explains why we have been writing specifications for the Java classes we code.

For example, in Chapter 7, we encountered a case study of a simulation of a ball bouncing in a box. The specifications of the ball and the box proved crucial to both writing the simulation's classes and fitting the classes together in the program.

The Java language and the Java compiler can help us write specifications of classes and check that a class correctly matches (implements) its specification. In this chapter we study several Java constructions for designing programs in separate classes:

1. the `interface` construction, which lets us code in Java the information we specify in a class diagram;

2. the `extends` construction, which lets us code a class by adding methods to a class that already exists;

3. the `abstract class` construction, which lets us code an incomplete class that can be finished by another class.

Finally, to help us group together a collection of related classes into the same folder, we use Java's `package` construction.

We will study each of the constructions in turn in this chapter.

9.2 Java Interfaces

In the previous chapters, we used informal specifications to design classes and to connect them to their collaborator classes. But the Java language provides a construct, called an `interface`, that lets us include a specification as an actual Java component of an application—we type the interface into a file and compile it, just like a class. Then, we use the compiled `interface` in two ways:

- We write classes that match or `implement` the `interface`, and the Java compiler verifies this is so.

- We write classes that rely upon the `interface`, and the Java compiler verifies this is so.

These ideas are best explained with a small example: Say that you and a friend must write two classes—one class models a bank account and the other class uses the bank account to make monthly payments on a mortgage. You will model the bank account,

Figure 9.1: Java interface

```
/** BankAccountSpecification specifies the behavior of a bank account.  */
public interface BankAccountSpecification
{ /** deposit adds money to the account
    * @param amount - the amount of the deposit, a nonnegative integer  */
  public void deposit(int amount);

  /** withdraw deducts money from the account, if possible
    * @param amount - the amount of the withdrawal, a nonnegative integer
    * @return true, if the the withdrawal was successful;
    *  return false, otherwise.  */
  public boolean withdraw(int amount);
}
```

your friend will write the monthly-payment class, and the two of you will work simultaneously. But how can your friend write his class without yours? To do this, the two of you agree on the Java interface stated in Figure 1, which specifies, *in the Java language*, the format of the yet-to-be written bank-account class. The `interface` states that, whatever class is finally written to implement a `BankAccountSpecification`, the class must contain two methods, `deposit` and `withdraw`, which behave as stated. Compare Figure 1 to Table 10 of Chapter 6, which presented a similar, but informal, specification.

A Java `interface` is a collection of header lines of methods for a class that is not yet written. The `interface` is not itself a class—it is a listing of methods that some class might have.

The syntax of a Java interface is simply

```
public interface NAME
{ METHOD_HEADER_LINES }
```

where `METHOD_HEADER_LINES` is a sequence of header lines of methods, terminated by semicolons. As a matter of policy, we insert a comment with each header line that describes the intended behavior of the method.

The `BankAccountSpecification` interface is placed in its own file, `BankAccountSpecification.java`, and is compiled like any other component. Once the interface is compiled, other classes can use it in their codings, as we now see.

Your friend starts work on the mortgage-payment class. Although your friend does not have the coding of the bank-account class, it does not matter — whatever the coding will be, it will possess the methods listed in `interface BankAccountSpecification`. Therefore, your friend writes the class in Figure 2, which uses the `BankAccountSpecification` as the data type for the yet-to-be-written bank-account class. The interface name, `BankAccountSpecification`, is used as a data type, just like class names are used as

Figure 9.2: class that references a Java interface

```
/** MortgagePaymentCalculator makes mortgage payments */
public class MortgagePaymentCalculator
{ private BankAccountSpecification bank_account;  // holds the address of
                // an object that implements the BankAccountSpecification

  /** Constructor MortgagePaymentCalculator initializes the calculator.
    * @param account - the address of the bank account from which we
    *   make deposits and withdrawals  */
  public MortgagePaymentCalculator(BankAccountSpecification account)
  { bank_account = account; }

  /** makeMortgagePayment makes a mortgage payment from the bank account.
    * @param amount - the amount of the mortgage payment  */
  public void makeMortgagePayment(int amount)
  { boolean ok = bank_account.withdraw(amount);
    if ( ok )
       { System.out.println("Payment made: " + amount); }
    else { ... error ... }
  }

  ...
}
```

data types. This lets us write a constructor method that accepts (the address of) an object that has the behavior specified in `interface BankAccountSpecification` and it lets us use this object's `withdraw` method in the method, `makeMortgagePayment`.

`Class MortgagePaymentCalculator` can now be compiled; the Java compiler validates that the class is using correctly the methods listed in `interface BankAccountSpecification`. (That is, the methods are spelled correctly and are receiving the correct forms of arguments, and the results that the methods return are used correctly.) In this way, your friend completes the class that makes mortgage payments.

Meanwhile, you are writing the class that implements `interface BankAccountSpecification`; this might look like Figure 3. This is essentially Figure 11 of Chapter 6, but notice in the class's header line the phrase, `implements BankAccountSpecification`. This tells the Java compiler that `class BankAccount` can be connected to those classes that use `BankAccountSpecifications`. When you compile `class BankAccount`, the Java compiler verifies, for each method named in `interface BankAccountSpecification`, that `class BankAccount` contains a matching method. (By "matching method," we mean that the class's method's header line is the same as the header line in the interface—the number of parameters is the same, the types of the parameters are the same, and the result type is the same. But the names of the formal parameters need not be

Figure 9.3: a class that implements a Java interface

```
/** BankAccount manages a single bank account; as stated in its
  * header line, it _implements_ the BankAccountSpecification:  */
public class BankAccount implements BankAccountSpecification
{ private int balance;  // the account's balance

  /** Constructor BankAccount initializes the account */
  public BankAccount()
  { balance = 0; }

  // notice that methods  deposit  and  withdraw  match the same-named
  //  methods in interface  BankAccountSpecification:

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

  public boolean withdraw(int amount)
  { boolean result = false;
    if ( amount <= balance )
       { balance = balance - amount;
         result = true;
       }
    return result;
  }

  /** getBalance reports the current account balance
    * @return the balance */
  public int getBalance()
  { return balance; }
}
```

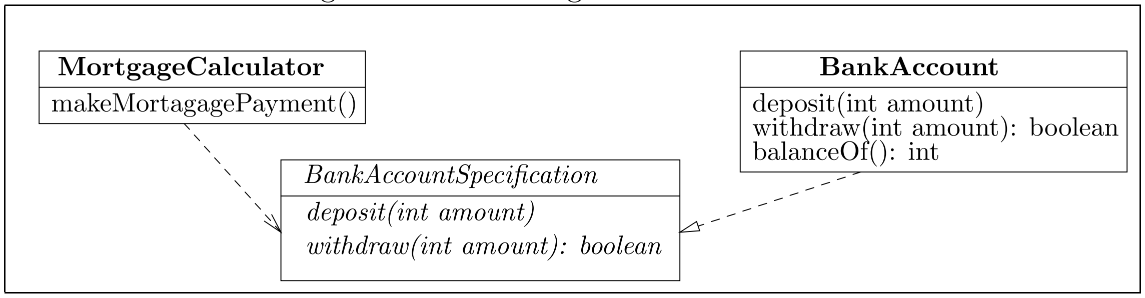
exactly the same, and the order of the methods in the class need not be the same as the order of the methods in the `interface`.)

Notice that `class BankAccount` has an additional method that is not mentioned in the `interface`; this is acceptable.

To connect together the two classes, we write a start-up method with statements like the following:

```
BankAccount my_account = new BankAccount();
MortgageCalculator calc = new MortgageCalculator(my_account);
  ...
calc.makeMortgagePayment(500);
```

Figure 9.4: class diagram of Java interface



Since the Java compiler verified that `BankAccount implements BankAccountSpecification`, the `my_account` object can be an argument to the constructor method of `MortgageCalculator`.

A major advantage of using Java interfaces is this:

> `Class MortgageCalculator` *is not rewritten or changed to refer to* `BankAccount` — once `class MortgageCalculator` is correctly compiled with `interface BankAccountSpecification`, it is ready for use.

In this way, components can be separately designed, written, compiled, and later connected together into an application, just like the disc player and batteries were separately manufactured and later connected together.

Figure 4 shows the class diagram for the example in Figures 1 to 3. To distinguish it from the classes, the Java interface is written in italics. Since `class MortageCalculator` depends on (references) the interface, a dotted arrow is drawn from it to the interface. Since `class BankAccount` implements the interface, a dotted arrow with a large arrowhead is drawn from it to the interface.

The class diagram in Figure 4 shows that `MortgageCalculator` and `BankAccount` do *not* couple-to/depend-on each other (in the sense of Chapter 6)—they both depend on `interface BankAccountSpecification`, which is the "connection point" for the two classes. Indeed, `MortgageCalculator` and `BankAccount` are simple examples of "subassemblies" that connect together through the `BankAccountSpecification` interface. This makes it easy to remove `BankAccount` from the picture and readily replace it by some other class that also `implements BankAccountSpecification`. (See the Exercises that follow this section.)

It is exactly this use of Java interfaces that allows teams of programmers build large applications: Say that a team of programmers must design and build a complex application, which will require dozens of classes. Perhaps the programmers are divided into three groups, each group agrees to write one-third of the application, and the three subassemblies will be connected together. If all the programmers first agree on the interfaces where the three subassemblies connect, then each group can independently develop their own subassembly so that it properly fits into the final product

Exercises

1. Given this interface,

   ```
   public interface Convertable
   { public double convert(int i); }
   ```

   which of the following classes will the Java compiler accept as correctly implementing the interface? Justify your answers.

   (a)
   ```
   public class C1 implements Convertable
   { private int x;
     public C1(int a) { x = a; }
     public double convert(int j)
     { return x + j; }
   }
   ```
   (b)
   ```
   public class C2 implements Convertable
   { private int x;
     public C1(int a) { x = a; }
     public int convert(int i)
     { return i; }
   }
   ```
   (c)
   ```
   public class C3
   { private int x;
     public C3(int a) { x = a; }
     public double convert(int i)
     { return (double)x; }
   }
   ```

2. Given this interface,

   ```
   public interface Convertable
   { public double convert(int i); }
   ```

   which of the following classes use the interface correctly? Justify your answers.

   (a)
   ```
   public class Compute1
   { private Convertable convertor;

     public Compute1(Convertable c)
     { convertor = c; }

     public void printConversion(int x)
     { System.out.println(convertor.convert(x)); }
   }
   ```

(b)
```
public class Compute2 uses Convertable
{ public Compute2() { }

  public void printIt(double x)
  { System.out.println(Convertable.convert(x)); }
}
```

(c)
```
public class Compute3
{ Convertable c;

  public Compute3()
  { c = new Convertable(); }

  public void printIt(int v)
  { System.out.println(c.compute(v)); }
}
```

3. After you have answered the previous two questions, do the following:

   - Place `interface Convertable` in the file, `Convertable.java`, and compile it. Next, copy `class C1` to the file, `C1.java`, in the same folder as `Convertable.java`, and compile it.
   - Place `class Compute1` in the file, `Compute1.java`, in the same folder as `Convertable.java`, and compile it.
   - Now, place the following code in the file, `Start.java`, in the same folder as `Convertable.java`, and compile it:

     ```
     public class Start
     { public static void main(String[] args)
       { C1 c = new C1(1);
         Compute1 computer = new Computer(c);
         computer.printConversion(3);
       }
     }
     ```

     Execute `Start`.

   Notice that both `C1.java` as well as `Compute1.java` use the same compiled instance of `Convertable.java`. This arrangement is a bit awkward when two different people write the two classes — they must share the same folder. We will repair this difficulty when we study Java `package`s later in this chapter.

4. Reconsider interface `BankAccountSpecification` from Figure 1. We will use it to improve the bank-accounts manager program from Section 8 of Chapter 6.

(a) Revise `class BankAccount` from Figure 11 of Chapter 6 so that it implements `BankAccountSpecification`. Next, revise `class BankWriter` (Figure 15, Chapter 6) and `class AccountController` (Figure 16, Chapter 6) so that they invoke the methods of an object of type `BankAccountSpecification` (and *not* `BankAccount`). Why does `AccountController` compile correctly but `BankWriter` does not?

Repair `interface BankAccountSpecification` so that `class BankWriter` compiles without errors. Does `class AccountManager` of Figure 16 require any changes?

(b) Next, discard `class BankAccount` from Figure 11 of Chapter 6 and replace it by this class:

```
public class SillyAccount implements BankAccountSpecification
{ public SillyAccount() { }
  public void deposit(int amount) { }
  public boolean withdraw(int amount) { return true; }
  public int getBalance() { return 0; }
}
```

Change `class AccountManager` in Figure 16 so that it declares,

```
BankAccountSpecification account = new SillyAccount();
```

Does `AccountManager` compile without error? Do any of the other classes require changes to compile?

This exercise demonstrates that Java `interfaces` make it simple to replace one class in an assembly without rewriting the other classes.

5. Review the moving-ball animation in Chapter 7, Section 9.

(a) First, write a Java `interface` that describes `MovingObjectBehavior` as defined in Table 10 of Chapter 7. Then make `class MovingBall` in Figure 12, Chapter 7, `implement MovingObjectBehavior`, and change classes `BounceController` and `BallWriter` so that they mention only `MovingObjectBehavior` (and not `MovingBall`).

(b) Based on the changes you made, redraw the class diagram in Figure 9, Chapter 7.

(c) Next, replace `class MovingBall` by this class:

```
/** ThrobbingBall models a stationary ball that changes size */
public class ThrobbingBall implements MovingObjectBehavior
{ private int max_radius = 60;
  private int increment = 10;
  private int current_radius; // invariant: 0 <= current_radius < max_radius
  private int position = 100;
  public ThrobbingBall() { current_radius = 0; }
  public int xPosition() { return position; }
  public int yPosition() { return position; }
  public int radiusOf() { return current_radius; }
  public void move()
  { current_radius = (current_radius + increment) % max_radius; }
}
```

and in Figure 16, Chapter 7, replace the `MovingBall` object constructed within `class BounceTheBall` by

```
ThrobbingBall ball = new ThrobbingBall();
```

Recompile `class BounceTheBall` and execute the animation.

9.2.1 Case Study: Databases

In Chapter 8, Section 6, we designed a database, named `class Database`, to hold a collection of "record" objects, each of which possessed a unique "key" object to identify it. The database was designed to be general purpose, in the sense that records might be library-book objects or bank-account objects, or tax records. As stated in Chapter 8, the crucial behaviors were

1. The `Database` holds a collection of `Record` objects, where each `Record` holds a `Key` object. The remaining structure of the `Records` is unimportant and unknown to the database.
2. The `Database` will possess `insert`, `find`, and `delete` methods.
3. `Records`, regardless of their internal structure, will possess a `getKey` method that returns the `Record`'s `Key` object when asked.
4. `Key` objects, regardless of their internal structure, will have an `equals` method that compares two `Keys` for equality and returns true or false as the answer.

Based on these ideas, informal specifications of `Record` and `Key` were written (see Table 3 of Chapter 8), and `class Database` was written to use the methods defined in the specifications; see Figure 4 of Chapter 8.

Clearly, the types `Record` and `Key` are not meant to be specific classes; they should be the names of two Java `interfaces`, so that we can compile `class Database` now and decide later how to implement the two interfaces.

Figure 5 shows how to transform the informal specifications of `Record` and `Key` from Table 3 of Chapter 8 into `interfaces`. These interfaces are compiled first, then `class Database` from Figure 4, Chapter 8, can be compiled — please review that Figure, now. Note how `class Database` refers to `Record` and `Key` in its coding. In

Figure 9.5: interfaces for Record and Key

```
/** Record is a data item that can be stored in a database  */
public interface Record
{ /** getKey  returns the key that uniquely identifies the record
    * @return the key  */
  public Key getKey();
}

/** Key is an identification, or ``key,'' value   */
public interface Key
{ /** equals  compares itself to another key, m, for equality
    * @param m - the other key
    * @return true, if this key and  m  have the same key value;
    *  return false, otherwise  */
  public boolean equals(Key m);
}
```

particular, `Record`'s `keyOf` method and `Key`'s `equals` method are used in crucial ways to insert and find records in the database.

As noted in Section 8.6.5, we might use the database to hold information about bank accounts that are identified by integer keys. Figure 5 of Chapter 8, which defines bank accounts and integer keys, should be rewritten — Figure 6 shows these two classes revised so that they implement the `Record` and `Key` interfaces, respectively. Note that both classes are properly named and `implement` their respective interfaces.

The first class, `BankAccount`, keeps its key as an attribute and gives it away with its `getKeyOf` method; the class knows nothing about its key's implementation. The second class, `IntegerKey`, uses an integer attribute as its internal state. Its `equals` method must compare its internal integer to the integer held in its argument, `c`. To do this, object `c` must be *cast* into its underlying type, `IntegerKey`, so that the `getInt` method can be queried for `c`'s integer. (From the perspective of the Java compiler, an object whose data type is `Key` does not necessarily possess a `getInt` method; the cast is necessary to tell the compiler that `c` is actually an `IntegerKey`, which does possess a `getInt` method.)

Unfortunately, we cannot avoid the cast by writing `equals`'s header line as

```
public boolean equals(IntegerKey c)
```

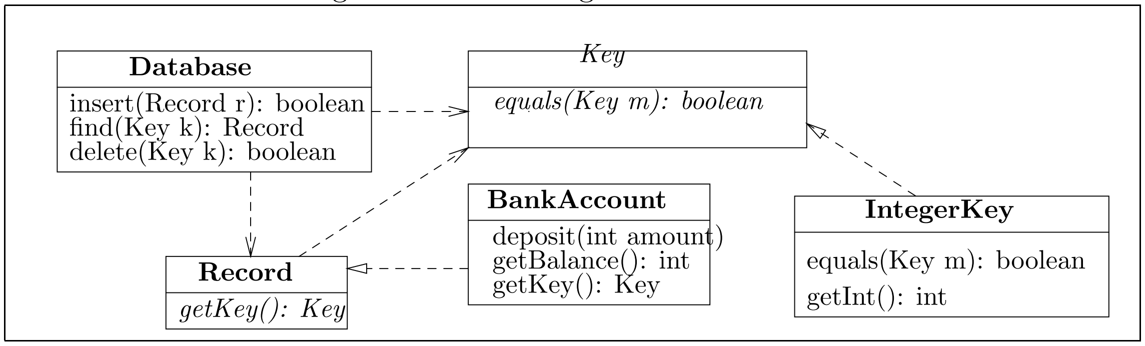
because the parameter's data type would not match the data type of the parameter of `equals` in `interface Key`. We must live with this clumsiness.

Now, we can build a database that holds `BankAccount` records; study carefully the following, which shows how to insert and retrieve bank acccounts:

Figure 9.6: implementing the database interfaces

```
/** BankAccount  models a bank account with an identification key */
public class BankAccount implements Record
{ private int balance;  // the account's balance
  private Key id;       // the identification key

  /** Constructor  BankAccount  initializes the account
    * @param initial_amount - the starting account balance, a nonnegative.
    * @param id - the account's identification key  */
  public BankAccount(int initial_amount, Key id)
  { balance = initial_amount;
    key = id;
  }

  /** deposit adds money to the account.
    * @param amount - the amount of money to be added, a nonnegative int */
  public void deposit(int amount)
  { balance = balance + amount; }

  /** getBalance reports the current account balance
    * @return the balance */
  public int getBalance() { return balance; }

  /** getKey returns the account's key
    * @return the key */
  public int getKey() { return key; }
}


/** IntegerKey models an integer key */
public class IntegerKey implements Key
{ private int k;  // the integer key

  /** Constructor  IntegerKey  constructs the key
    * @param i - the integer that uniquely defines the key */
  public IntegerKey(int i) {  k = i; }

  /** equals  compares this Key to another for equality
    * @param c - the other key
    * @return true, if this key equals k's; return false, otherwise */
  public boolean equals(Key c)
  { return  ( k == ((IntegerKey)c).getInt() ); }

  /** getInt returns the integer value held within this key */
  public int getInt() { return k; }
}
```

Figure 9.7: class diagram with interfaces



```
Database db = new Database(4);  // see Figure 4, Chapter 8

BankAccount a = new BankAccount(500, new IntegerKey(1234));
boolean result1 = db.insert(a);

IntegerKey k = new IntegerKey(567);
BankAccount b = new BankAccount(1000, k);
boolean result2 = db.insert(b);

Record r = db.find(k);  // retrieve object indexed by Key  k
System.out.println(((BankAccount)r).getBalance());  // why is the cast needed?
```

Since `Database`'s `find` method returns an object that is known to be only a `Record`, a cast to `BankAccount` is required by the Java compiler in the last statement of the above example.

Figure 7 shows the database example's class diagram. The diagram shows that `Database` is coupled only to the two interfaces and not to the classes that implement the interfaces. This is a clear signal that other classes of records and keys can be used with `class Database`. (For example, Figure 6 of Chapter 8 shows codings of classes of library books and catalog numbers; by recoding the two classes so that they implement the `Record` and `Key` interfaces, the two classes can be readily used with the database.)

Also, note how `BankAccount` and `IntegerKey` are not coupled to each other, making it easy to "unplug" and replace both classes. Finally, by transitivity, the dotted arrows from `BankAccount` to `Record` to `Key` let us infer correctly that `BankAccount` depends on interface `Key` as well as interface `Record`.

Exercises

Return to Section 8.6, "Case Study: Databases," in Chapter 8. Rework Figure 6 so that its classes implement `Record` and `Key`.

9.3 Inheritance

In the previous chapters, we built many applications that paint onto a panel within a graphics window, like this:

```
import java.awt.*;
import javax.swing.*;
public class MyPanel extends JPanel
{  ...

  public void paintComponent(Graphics g)
  { ...  instructions for painting on a panel ... }
}
```

This tactic worked because, within the package `javax.swing`, there is a prewritten class, `class JPanel`, which contains the instructions for constructing a blank graphics panel that can be displayed on a monitor. We exploit this already written class by

- writing a new class that `extends JPanel`
- writing a method, `paintComponent`, that contains instructions for painting on the panel.

When we construct an object from our class, say,

```
MyPanel p = new MyPanel(...);
```

the object we construct has all the private fields and methods within `class JPanel` plus the new methods and fields within `class MyPanel`. This gives object `p` the ability to display a panel on the display as well as paint shapes, colors, and text onto it.

This style of "connecting" to an already written class and adding new methods is called *inheritance*. We say that `MyPanel` is a *subclass* of `JPanel` (and `JPanel` is a *superclass* of `MyPanel`).

To understand inheritance further, study this small example developed from scratch: Say that a friend has written `class Person`:

```
public class Person
{ private String name;

  public Person(String n)
  { name = n; }

  public String getName()
  { return name; }

  ... // Other clever methods are here.
}
```

Pretend this class has proven popular, and many programs use it.

Next, say that you must write a new application that models persons with their addresses. You would like to "connect" an address to a person and reuse the coding within `class Person` to save you time writing your new class. You can use inheritance to do this:

```
public class PersonAddress extends Person
{ private String address;

  public PersonAddress(String the_name, String the_addr)
  { super(the_name);  // this gives  the_name  to  Person's  constructor
    address = the_addr;
  }

  public String getAddress()
  { return address; }

  ...
}
```

Because its title line states, `extends Person`, class `PersonAddress` inherits the fields and methods of `class Person`. When we construct an object from the new class, say,

```
PersonAddress x = new PersonAddress("fred", "new york");
```

the object constructed in computer storage contains an `address` variable and also a `name` variable. Indeed, the first statement in the constructor method for `PersonAddress`,

```
super(the_name);
```

invokes the constructor method within `Person` (the superclass), so that the person's name is inserted into the `name` field. (When this tactic is used, the `super` instruction *must be the first statement within the subclass's constructor method.*)

When we use the object we constructed, e.g.,

```
System.out.println("Name: " + x.getName());
System.out.println("Address: " + x.getAddress());
```

the methods from `class Person` *can be used alongside the methods from* `class PersonAddress`. It is striking that we can say, `x.getName()`, even though there is no `getName` method within `class PersonAddress`; the inheritance technique "connects" the methods of `Person` to `PersonAddress`.

We use a large arrowhead in class-diagram notation to denote inheritance:

```
FIGURE HERE:   PersonAddress  ---|>  Person
```

Inheritance is fundamentally different from Java interfaces, because inheritance *builds upon* classes that are already written, whereas interfaces *specify* classes that are not yet written (or unavailable). Inheritance is often used when someone has written a basic class that contains clever methods, and other people wish to use the clever methods in their own classes without copying the code — they write subclasses. A good example is our extension of `class JPanel`, at the beginning of this section.

Here is another situation where inheritance can be useful: Again, say that `class Person` has proved popular, and someone has written a useful class that writes information about persons:

```
public class WritePerson
{ ...

  public void writeName(Person p)
  { ... clever instructions to write  p's  name ... }
}
```

If we write an application that manages persons plus their addresses, we can exploit both `class Person` and `class WritePerson` by writing `class PersonAddress extends Person` as shown above. Then, we can do this:

```
PersonAddress x = new PersonAddress("fred", "new york");
WritePerson writer = new WritePerson(...);
writer.writeName(x);
System.out.println("Address: " + x.getAddress());
```

The statement, `writer.writeName(x)` is crucial, because the `writeName` method expects an argument that has data type `Person`. Because `x` has data type `PersonAddress` and because `PersonAddress` extends `Person`, `x` is acceptable to `writeName`. This behavior is based on *subtyping* and is explored in the next section.

Exercises

1. Given these classes,

```
public class Person
{ private String name;

  public Person(String n) { name = n; }

  public String getName() { return name; }

  public boolean sameName(Person other)
  { return  getName().equals(other.getName(); }
}
```

```
public class PersonAddress extends Person
{ private String address;

  public PersonAddress(String the_name, String the_addr)
  { super(the_name);
    address = the_addr;
  }

  public String getAddress() { return address; }

  public boolean same(PersonAddress other)
  { return   sameName(other) && address.equals(other.getAddress()); }
}
```

and these declarations:

```
Person p = new Person("fred");
Person q = new PersonAddress("ethel", "new york");
```

Which of the following statement sequences are acceptable to the Java compiler? If a statement sequence is acceptable, what does it print when executed?

(a) `System.out.println(p.getAddress());`

(b) `System.out.println(q.getName());`

(c) `System.out.println(p.sameName(p));`

(d) `System.out.println(q.sameName(p));`

(e) `System.out.println(q.same(p));`

This example is studied further in the next Section.

9.4 Reference Types, Subtypes, and `instanceof`

When we first encountered data types in Chapter 3, we treated them as "species" of values—`int`, `double`, and `boolean` were examples of such species. The classifying values into species prevents inappropriate combinations of values, such as `true && 3` — the Java compiler does data-type checking to spot such errors. Data-type checking also spots bad actual-formal parameter combinations, such as `Math.sqrt(true)`.

Numerical, boolean, and character values are *primitive* (non-object), and their data types are called *primitive types.* The numeric primitive types are related by *subtyping*: `int` is a subtype of `double`, written `int <= double`, because an integer can be used in any situation where a double is required. For example, the integer `2` can be used within

```
double d = 4.5 / 2;
```

because a double answer can be produced by dividing the double, `4.5`, by `2`. Similarly, if a method expects an argument that is a double, as in

```
public double inverseOf(double d)
{ return  1.0 / d; }
```

it is acceptable to send the method an actual parameter that is an integer, e.g., `inverseOf(3)`. The Java compiler uses the subtyping relationship, `int <= double`, to check the well formedness of these examples.

Subtyping relationships simplify our programming; in particular, cumbersome cast expressions are not required. For example, it is technically correct but ugly to write `double d = 4.5 / ((double)2)`, and thanks to subtyping, we can omit the cast.

Begin footnote: Here is a listing of the subtyping relationships between the numeric types:

```
byte  <=  int  <=  long  <=  float  <=  double
```

Thus, `byte <= int`, `int <= long`, `byte <= long`, etc. *End footnote*

In addition to the primitive data types, there are object or *reference* data types: Every `class C` defines a reference data type, named `C` — its values are the objects constructed from the class. This explains why we write declarations like

```
Person p = new Person("fred");
```

class `Person` defines data type `Person`, and variable `p` may hold only addresses of objects that have data type `Person`.

Java interfaces and inheritance generate subtyping relationships between reference types as well. If we write the class,

```
public class MyPanel extends JPanel
{ ... }
```

then this subtyping relationship is generated: `MyPanel <= JPanel`. This means that the object, `new MyPanel()`, can be used in any situation where a value of data type `JPanel` is expected. We have taken for granted this fact, but it proves crucial when we construct, a new `MyPanel` object and insert it into a `JFrame` object:

```
// This example comes from Figure 12, Chapter 4:
import java.awt.*;
import javax.swing.*;
public class MyPanel extends JPanel
{ ... }

public class FrameTest3
{ public static void main(String[] args)
```

```
  { JFrame my_frame = new JFrame();
    // insert a new panel into the frame:
    my_frame.getContentPane().add(new MyPanel());
    ...
  }
}
```

The `add` method invoked within `main` expects a `JPanel` object as its argument. But since `MyPanel <= JPanel`, the newly constructed `MyPanel` object is acceptable.

This same phenomenon appeared at the end of the previous section when we used a `PersonAddress` object as an argument to the `writeName` method of `PersonWriter`.

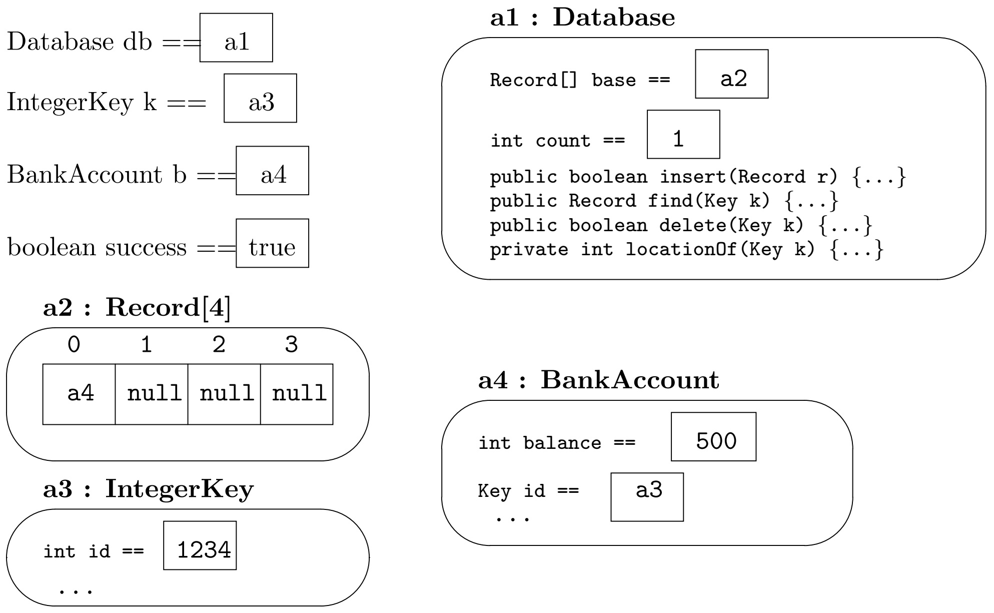
Similar subtyping principles also apply to Java interfaces: The Java compiler uses `interface` names as data type names, and the compiler enforces a subtyping relationship when an interface is implemented: if `class C implements I`, then `C <= I`. This proves crucial when connecting together classes:

Reconsider the database example in Figures 5-7; we might write

```
Database db = new Database(4);
IntegerKey k = new IntegerKey(1234);
BankAccount b = new BankAccount(500, k);
boolean success = db.insert(b);
```

The database method, `insert`, expects an arguments with data type `Record`, but it operates properly with one of type `BankAccount`, because `BankAccount <= Record`. This subtyping can be justified by noting that a `BankAccount` has all the methods expected of a `Record`, so `insert` executes as expected.

If we peek inside computer storage, we see that the above statements constructed these objects:



The diagram indicates that every object in storage is labelled with the name of the class from which the object was constructed. This is the *run-time data type* of the object. The diagram also illustrates that run-time data types are distinct from the data types that appear in assignments. For example, the object at address `a4` retains its run-time data type, `BankAccount`, even though it was assigned into an element of an array declared to hold `Records`. (Look at `a2`'s run-time data type.) The situation is acceptable because of the subtyping relationship.

Consider the following statements, which build on the above:

```
Record r = db.find(k);
System.out.println( ((BankAccount)r).getBalance() );
```

The first statement extracts from the data base the record matching `k`, that is, `a4` is assigned to `r`. But we *cannot* say, immediately thereafter, `r.getBalance()`, because variable `r` was declared to have data type `Record`, and there is no `getBalance` method listed in `interface Record`. The problem is that the data type in the statement, `Record r = db.find(k)`, is distinct from the run-time data type attached to the object that is assigned to `r`.

If we try to repair the situation with the assignment, `BankAccount r = db.find(k)`, the Java compiler complains again, because the `find` method was declared to return a result of data type `Record`, and `Record` is not a subtype of `BankAccount`!

This is frustrating to the programmer, who knows that `db` is holding `BankAccount` objects, but Java's compiler and interpreter are not intelligent enough to deduce this fact. Therefore, the programmer must write an explicit cast upon `r`, namely, `(BankAccount)r`, to tell the Java compiler that `r` holds an address of an object whose run-time type is `BankAccount`. Only then, can the `getBalance` message be sent.

If the programmer encounters a situation where she is not certain herself what is extracted from the database, then the `instanceof` operation can be used to ask the extracted record its data type, e.g.,

```
Record mystery_record = db.find(mystery_key);
if ( mystery_record instanceof BankAccount)
     { System.out.println( ((BankAccount)mystery_record).getBalance() ); }
else { System.out.println("unknown record type"); }
```

Stated precisely, the phrase, `EXPRESSION instanceof TYPE`, returns `true` exactly when the run-time data type attached to the object computed by `EXPRESSION` is a *subtype* of `TYPE`.

We can use the `instanceof` method to repair a small problem in the database example in the previous section. In addition to `class IntegerKey implements Key`, say that the programmer writes this new class:

```
/** StringKey models a key that is a string */
public class StringKey implements Key
{ private String s;
```

```
  public StringKey(String j)
  { s = j; }

  public String getString()
  { return s; }

  public boolean equals(Key m) { ... }
}
```

and say that she intends to construct some records that use `IntegerKey`s and some that use `StringKey`s. This seems ill-advised, because we see a problem when an `IntegerKey` object is asked to check equality against a `StringKey` object:

```
IntegerKey k1 = new IntegerKey(2);
StringKey k2 = new StringKey("two");
boolean answer = k1.equals(k2);
```

Surprisingly, the Java compiler will accept these statements as well written, and it is only when the program executes that the problem is spotted—execution stops in the middle of `IntegerKey`'s `equals` method at the statement, `int m = ((IntegerKey)another_key).getInt()`, and this exception message appears:

```
Exception in thread "main" java.lang.ClassCastException: StringKey
        at IntegerKey.equals(...)
```

Despite the declaration in its header line, `IntegerKey`'s `equals` method is unprepared to deal with all possible actual parameters whose data types are subtypes of `Key`!

If an application will be constructing both `IntegerKey`s and `StringKey`s, then we should improve `IntegerKey`'s `equals` method to protect itself against alien keys. We use the `instanceof` operation:

```
public boolean equals(Key another_key)
{ boolean answer;
  // ask if  another_key's  run-time data type is  IntegerKey:
  if ( another_key instanceof IntegerKey )
       { int m = ((IntegerKey)another_key).getInt();
         answer = (id == m);
       }
  else // another_key  is not an  IntegerKey, so don't compare:
       { answer = false; }
  return answer;
}
```

The phrase,

```
if ( another_key instanceof IntegerKey )
```

determines the address of the object named by `another_key`, locates the object in storage, extracts the run-time data type stored in the object, and determines whether that data type is a subtype of `IntegerKey`. If it is, `true` is the answer. If not, `false` is the result.

Exercises

1. Given these classes,

```
public class Person
{ private String name;

  public Person(String n) { name = n; }

  public String getName() { return name; }

  public boolean sameName(Person other)
  { return  getName().equals(other.getName(); }
}

public class PersonAddress extends Person
{ private String address;

  public PersonAddress(String the_name, String the_addr)
  { super(the_name);
    address = the_addr;
  }

  public String getAddress() { return address; }

  public boolean same(PersonAddress other)
  { return   sameName(other) && address.equals(other.getAddress()); }
}
```

and these declarations:

```
Person p = new Person("fred");
Person q = new PersonAddress("ethel", "new york");
```

Which of the following statement sequences are acceptable to the Java compiler? If a statement sequence is acceptable, what does it print when executed?

(a) `System.out.println(p.sameName(q));`

(b) `Person x = q; System.out.println(x.getName());`

   (c) `PersonAddress x = p; System.out.println(x.getAddress());`
   (d) `Person x = q; System.out.println(x.getAddress());`
   (e) `System.out.println(q.same(p));`

2. Explain why this example fails to compile:

```
public class C
{ private int x;
  public C() { x = 0; }
}

public class D extends C
{ public D() { super(); }
  public void increment() { x = x + 1; }
}
```

   If you are interested in repairing the example, read the section, "Subclasses and Method Overriding," at the end of the Chapter.

3. Use the `instanceof` operation to improve the existing implementations of `interface Key`:
   (a) Insert the above coding of `equals` into `class IntegerKey` in Figure 6.
   (b) Finish writing `class StringKey` so that it has `equals` method like the one you wrote in the previous exercise. (Hint: Use the `compareTo` method, in Table 5, Chapter 3, to write the `lessthan` method.)
   (c) Write a test class that executes these statements:

```
Database db = new Database(4);  // see Figure 3, Chapter 8

BankAccount b = new BankAccount(500, new IntegerKey(1234));
IntegerKey k = new StringKey("lucy");
BankAccount lucy = new BankAccount(1000, k);
boolean result1 = db.insert(b);
boolean result2 = db.insert(lucy);

Record p = db.find(k);
BankAccount q = (BankAccount)p;
System.out.println(q.getBalance());

Key k = q.getKey();
if ( k instanceof IntegerKey )
     { System.out.println( ((IntegerKey)k).getInt() ); }
else if ( k instance of StringKey )
```

```
     { System.out.println( ((StringKey)k).getString() ); }
else { System.out.println("unknown key value"); }
```

What appears on the display?

4. Given these interfaces and classes,

```
public interface I
{ public int f(int i); }

public class C implements I
{ public C() { }
  public int f(int i) { return i + 1; }
  public void g() { }
}

public class D
{ public D() { }
  public int f(int i) { return i + 1; }
}
```

   (a) Which of the following subtyping relations hold true? `C <= I`; `D <= I`; `C <= D`.

   (b) Given these initializations,

   ```
   I x = new C();
   C y = new C();
   C z = (C)x;
   ```

   Which of the following expressions evaluate to `true`? `x instanceof I`; `x instanceof C`; `x instanceof D`; `y instanceof I`; `y instanceof C`; `y instanceof D`; `z instanceof I`; `z instanceof C`.

   (c) Add casts, where necessary, so that the following statement sequence passes the scrutiny of the Java compiler:

   ```
   I x = new C();
   x.f(21);
   x.g();
   C y = x;
   y.g();
   if ( x instanceof C )
      { x.g(); }
   ```

   (d) Explain why the Java compiler complains about these statements:

```
        D a = new D();
        I b = a;
        if ( a instanceof C )
           { System.out.println("!");
```

9.5 Abstract Classes

Simply stated, an *abstract class* is a class with missing method bodies. The missing bodies are supplied by subclasses that extend the abstract class. Why would we use such a construction? Here is a small example:

In the previous section, we pretended that a `class Person` was an important component of many programs. Perhaps the author of `class Person` intended that `Person` objects *must* have addresses, but the author did not wish to code the address part. (The address might be a string or an integer or a color or ....) An abstract class can state exactly the author's wishes:

```
public abstract class Person  // note the keyword,  abstract
{ private String name;

  public Person(String n)
  { name = n; }

  public String getName()
  { return name; }

  public abstract String getAddress();  // method will be written later

  ...
}
```

This variant of `class Person` has two additions from the one seen earlier:

1. In the title line, the keyword, `abstract`, announces that we cannot construct `Person` objects, because there are missing method bodies.
2. The title line for method `getAddress` contains the keyword, `abstract`, and the method is missing its body. The title line is terminated by a semicolon.

Because the class is abstract, we *cannot* construct `new Person(...)` objects. Instead, we can only `extend` the class with its missing method body. This extension, seen earlier, works fine:

```
public class PersonAddress extends Person
{ private String address;
```

```
  public PersonAddress(String the_name, String the_addr)
  { super(the_name);  // this gives  the_name  to  Person's  constructor
    address = the_addr;
  }

  public String getAddress()
  { return address; }

  ...
}
```

We can construct `new PersonAddress(...)` objects, as before. And, we can construct other variants, e.g.,

```
public class PersonWithInt extends Person
{ private int address;

  public PersonWithInt(String the_name, int the_addr)
  { super(the_name);
    address = the_addr;
  }

  public String getAddress()
  { return  "" + address; }  // make the int into a string

  ...
}
```

Note that the `getAddress` method must have the same title line as the one in the superclass; this forces the integer to be converted into a string when it is returned.

9.5.1 Case Study: Card Players

An abstract class is "half interface" and "half superclass," and it is most useful for grouping, under a single data type name, classes that share methods.

Here is an example that illustrates good use of interfaces with abstract classes: In Chapter 8, we saw how to design cards and card decks for a card game. (See Tables 7 and 8 and Figures 9 and 10 from that Chapter.) If we continue developing the card game, we will design a `class Dealer`, which collaborates with cards and card decks as well as with objects that are card players. At this point, we do not know exactly how the card players will behave (this is dependent on the rules of the card game), but `class Dealer` would expect that a card player has a method to receive a card and has a method that replies whether a player wants to receive additional cards. The obvious step is to write an interface; Figure 8 shows it. Now we can write a

Figure 9.8: interface for card playing

```
/** CardPlayerBehavior defines expected behaviors of card players */
public interface CardPlayerBehavior
{ /** wantsACard replies whether the player wants one more new card
    * @return whether a card is wanted */
  public boolean wantsACard();

  /** receiveCard accepts a card and adds it to the player's hand
    * @param c - the card  */
  public void receiveCard(Card c);
}
```

`class Dealer` whose coding uses the interface to deal cards to players. (See Exercise 1, below.)

Next, we consider implementations of the `CardPlayerBehavior` interface. Perhaps there are two formats of card players—computerized players and human players. (A computerized player is an object that does card playing all by itself; a human player is an object that helps the human user join in the play.) The two classes of player receive cards in the same way, but when a human-player object is asked if it wants a card, it asks the user for a decision. In contrast, a computerized-player object will make the decision based on an algorithm — that is, the two players share the same implementation of the `receiveCard` method but use different implementations of the `wantsACard` method.

To accommodate the situation, we invent the abstract class, `CardPlayer`, that acts as the superclass of both computerized and human card players. Figure 9 shows `abstract class CardPlayer`; its subclasses, `ComputerPlayer` and `HumanPlayer`, appear in Figure 10.

`CardPlayer` is labelled `abstract` because it is incomplete: It is missing its `wantsACard` method (which it needs to implement `CardPlayerBehavior`). Only the header line for `wantsACard` appears; it contains the keyword, `abstract`, and is terminated by a semicolon. In contrast, `receiveCard` and another method, `showCards`, are written in entirety.

The two classes in Figure 8, `HumanPlayer` and `ComputerPlayer`, extend the abstract class, meaning the classes get the attributes and methods of a `CardPlayer`. The two classes also supply their own versions of the missing `wantsACard` method.

At this point, there are no more missing methods, and we say that the classes `HumanPlayer` and `ComputerPlayer` are *concrete classes*. Objects can be constructed from concrete classes. (Recall that they cannot be constructed from abstract classes.) Say that we construct two players:

```
ComputerPlayer p = new ComputerPlayer(3);
```

Figure 9.9: abstract class for card players

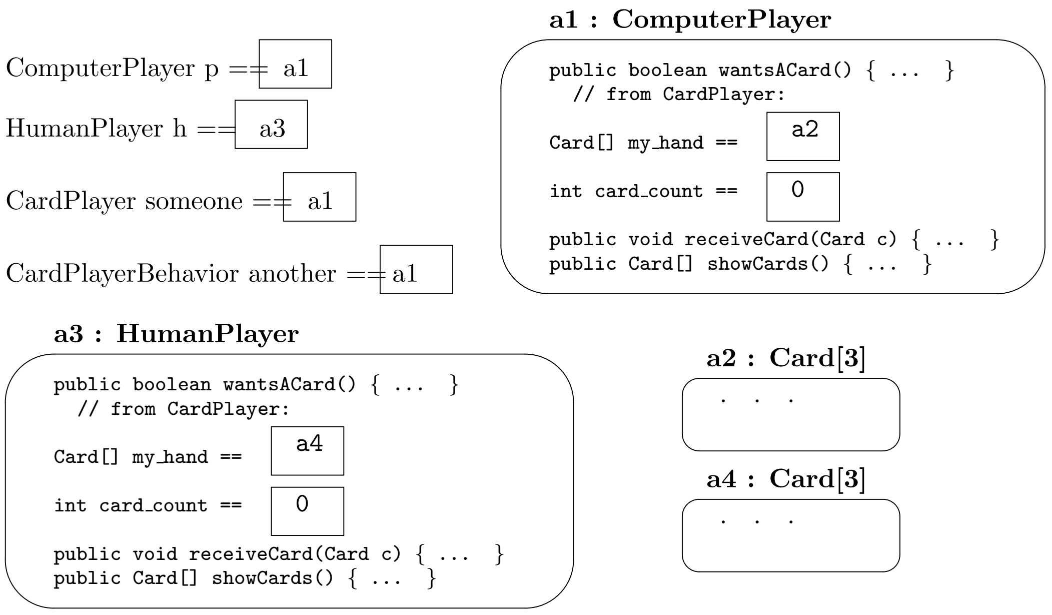
```
/** CardPlayer models an abstract form of card player */
public abstract class CardPlayer implements CardPlayerBehavior
{ private Card[] my_hand;  // the player's cards
  private int card_count;  // how many cards are held in the hand

  /** CardPlayer builds the player
    * @param max_cards - the maximum cards the player can hold.  */
  public CardPlayer(int max_cards)
  { my_hand = new Card[max_cards];
    card_count = 0;
  }

  /** wantsACard replies whether the player wants one more new card
    * @return whether a card is wanted */
  public abstract boolean wantsACard();    // method will be written later

  public void receiveCard(Card c)
  { my_hand[card_count] = c;
    card_count = card_count + 1;
  }

  /** showCards displays the player's hand
    * @return an array holding the cards in the hand  */
  public Card[] showCards()
  { Card[] answer = new Card[card_count];
    for ( int i = 0;  i != card_count;  i = i + 1 )
        { answer[i] = my_hand[i]; }
    return answer;
  }
}
```

Figure 9.10: subclasses of CardPlayer

```
/** HumanPlayer models a human who plays cards */
public class HumanPlayer extends CardPlayer
{ /** HumanPlayer builds the player
    * @param max_cards - the maximum cards the player can hold  */
  public HumanPlayer(int max_cards)
  { super(max_cards); }   // invoke constructor in superclass

  public boolean wantsACard()
  { String response = JOptionPane.showInputDialog
                        ("Do you want another card (Y or N)?");
    return  response.equals("Y");
  }
}


/** ComputerPlayer  models a computerized card player  */
public class ComputerPlayer extends CardPlayer
{ /** ComputerPlayer builds the player
    * @param max_cards - the maximum cards the player can hold.  */
  public ComputerPlayer(int max_cards)
  { super(max_cards); }  // invoke constructor in superclass

  public boolean wantsACard()
  { boolean decision;
    Card[] what_i_have = showCards();
    ... statements go here that examine  what_i_have  and
            calculate a decision ...
    return decision;
  }
}
```

```
HumanPlayer h = new HumanPlayer(3);
CardPlayer someone = p;
CardPlayerBehavior another = someone;
```

The following situation appears in computer storage:



The fields (and methods) of `class CardPlayer` are adjoined to those of `class ComputerPlayer` to make a `ComputerPlayer` object. The same happens for the `HumanPlayer` object.

The assignment, `CardPlayer someone = p`, reminds us that the data types `ComputerPlayer` and `HumanPlayer` are subtypes of `CardPlayer`; objects of either of the first two types can be used in situations where a `CardPlayer` object is required. This means we can send the message, `boolean ask = someone.wantsACard()`, because this is a behavior expected of a `CardPlayer` (even though no method was written for `wantsACard` in the abstract class!).

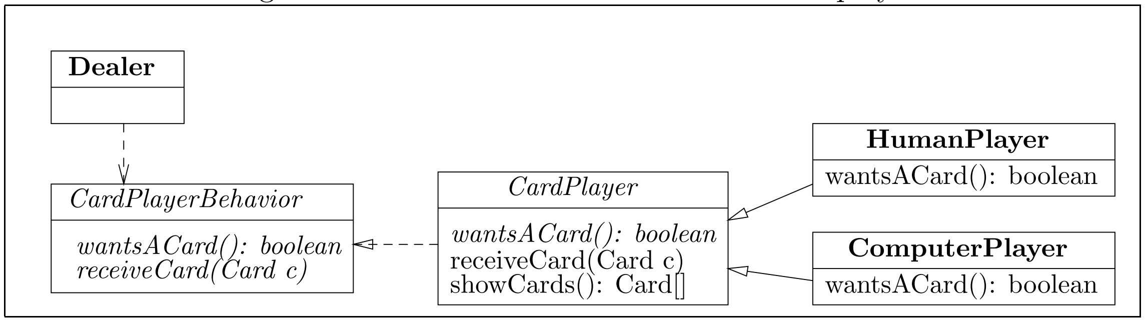
The last assignment, `CardPlayerBehavior another = someone`, is also acceptable, because interface names are data types. Although it is legal to say, `boolean b = another.wantsACard()`, the Java compiler will complain about `another.showCards()`, because the `showCards` method is not listed in interface `CardPlayerBehavior`; a cast would be necessary: e.g.,

```
if ( another instanceof CardPlayer )
   { ((CardPlayer)another).showCards(); }
```

The `instanceof` operation examines the data type stored within the object named by `another` and verifies that the type is a subtype of `CardPlayer`.

The above example has created these subtyping relationships:

Figure 9.11: architecture of dealer and card players



```
ComputerPlayer <=
                  CardPlayer  <=  CardPlayerBehavior
HumanPlayer    <=
```

Although the assignment, `CardPlayer someone = p`, was legal, this initialization is not—`CardPlayer someone = new CardPlayer(3)`—because objects cannot be constructed from abstract classes.

Figure 11 shows the architecture we have assembled with the example. The diagram tells us that the `Dealer` collaborates through the `CardPlayerBehavior` interface, which is implemented by the abstract class, `CardPlayer`. (The abstract parts are stated in italics.)

Exercises

1. Write `class Dealer` based on this specification:

`class Dealer`	models a dealer of cards
Responsibilities (methods)	
`dealTo(CardPlayerBehavior p)`	gives cards, one by one, to player `p`, until `p` no longer wants a card
Collaborators:	`CardPlayerBehavior, CardDeck, Card`

2. In the above specification, replace all occurrences of `CardPlayerBehavior` by `CardPlayer`. Rewrite `class Dealer` accordingly. Compare the advantages and disadvantages of the two variants of `class Dealer`.

3. One abstract class can extend another; here is an example:

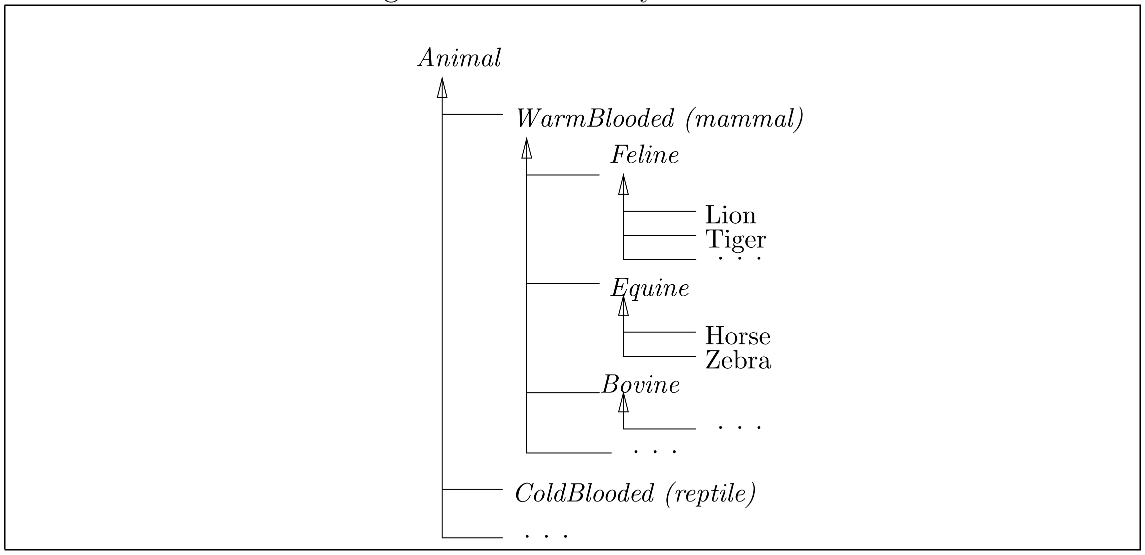
```
/** Point models a geometric point */
public class Point
{ private int x;
```

```
  private int y;

  public Point(int a, int b)
  { x = a;
    y = b;
  }

  public int xPosition() { return x; }
  public int yPosition() { return y; }
}

/** Shape models a two-dimensional geometric shape */
public abstract class Shape
{ private Point upper_left_corner;

  public Shape(Point location)
  { upper_left_corner = location; }

  /** locationOf returns the location of the shape's upper left corner */
  public Point locationOf()
  { return upper_left_corner; }

  /** widthOf returns the width of the shape, starting from its left corner */
  public abstract int widthOf();

  /** depthOf returns the depth of the shape, starting from its left corner */
  public abstract int depthOf();
}

/** Polygon models a polygon shape */
public abstract class Polygon extends Shape
{ private Point[] end_points;  // the nodes (corners) of the polygon

  public Polygon(Point upper_left_corner)
  { super(upper_left_corner); }

  /** setCorners remembers the nodes (corners) of the polygon */
  public void setCorners(Point[] corners)
  { end_points = corners; }
}
```

Now, write a concrete class, `Rectangle`, that extends `Polygon`. The constructor method for `class Rectangle` can look like this:

```
public Rectangle(Point location, int width, int height)
```

Figure 9.12: hierarchy of animals



Next, write a concrete class, `Circle`, that extends `Shape`.

9.5.2 Class Hierarchies

When we write programs that manipulate many different types of objects, we will find it helpful to draw the classes as a "hierarchy" or "taxonomy" based on their subclass relationships.

This approach comes from real-life taxonomies, like the one in Figure 12, which simplistically models the zoology of part of the animal kingdom.

The characteristics of animals appear at the internal positions of the hierarchy, and actual animals appear at the end positions—the "leaves" of the "tree." (`Animal` is the "root" of the "tree.") When one characteristic is listed beneath another, it means that the first is more specific or a "customization" of the second, in the sense that the first has all the behaviors and characteristics of the second. For example, being Equine means having all the characteristics of a WarmBlooded entity.

The zoological taxonomy helps us make sense of the variety of animals and reduces the amount of analysis we apply to the animal kingdom. In a similar way, in a taxonomy of classes, when a class, `A` is a superclass of class `B`, we place `B` underneath `A` in the hierarchy.

For example, if we designed a computer program that worked with the animal-kingdom taxonomy, the program would create objects for the classes at the leaves, for example,

```
Horse black_beauty = new Horse(...);
```

and it might assign,

```
Equine a_good_specimen = black_beauty;
```

But the program would *not* create objects from the non-leaf classes, such as `new Equine()`, because there are no such animals.

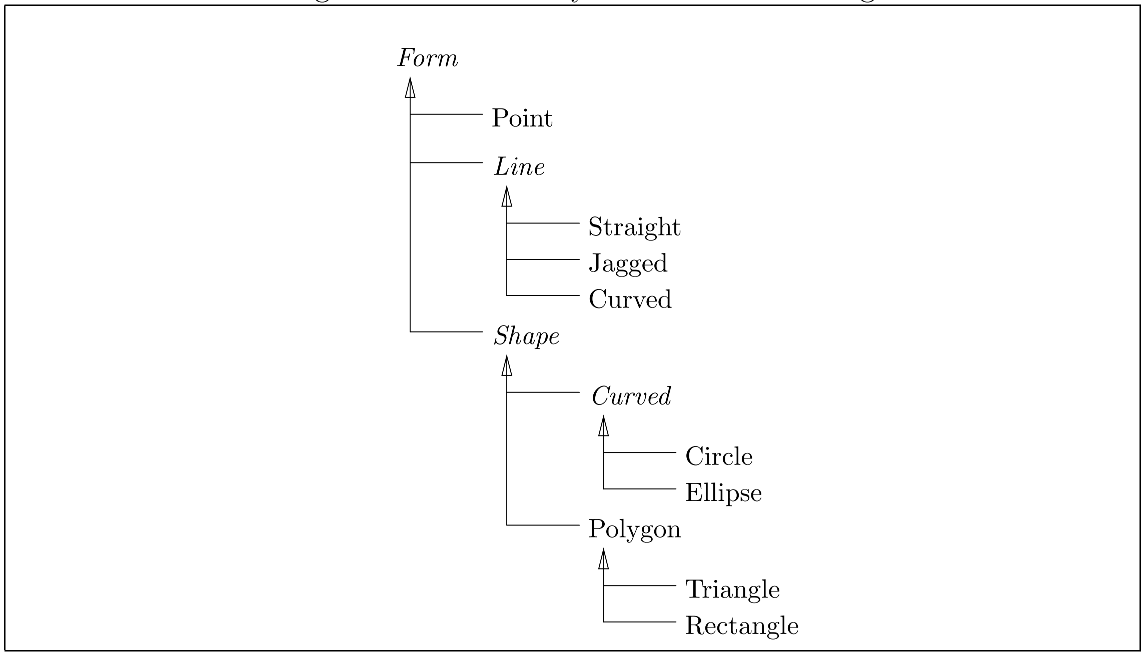
This example suggests that the non-leaf entries of a class hierarchy represent typically (but not always!) abstract classes, and the leaves must be concrete classes from which objects are constructed. The hierarchy can be converted into a collection of classes, that extend one another, e.g.,

```
public abstract class Animal
{
  // fields and methods that describe an animal
}

public abstract class WarmBlooded extends Animal
{
  // additional fields and methods specific to warm-blooded animals
}

public abstract class Equine extends WarmBlooded
{
  // fields and methods specific to equines
}

public class Horse extends Equine
{
  // fields and completed methods specific to horses
}
```

Here is a more computing-oriented example: Figure 13 shows a hierarchy that might arise from listing the forms one would draw in a graphics window. The forms that one actually draws reside (primarily) at the leaves of the tree; the non-leaf entries are adjectives that list aspects or partial behaviors. (Although this rule is not hard and fast—arbitrary `Polygon` objects are possible.)

The primary benefit of working with class hierarchies is that a hierarchy collects together related classes, meaning that commonly used attributes and methods can be written within abstract classes and shared by concrete classes. The major negative aspect of a class hierarchy is its size—one actual object might be constructed by extending many classes, which might be located in many different files. For example, a `Triangle` object constructed from Figure 13 is a composite of classes `Triangle`, `Polygon`, `Shape`, and `Form`; a programmer will quickly grow weary from reading four (or more) distinct classes to understand the interface and internal structure of just one object! If possible, limit the hierarchies you write to a depth of two or three classes.

Figure 9.13: hierarchy of forms for drawing



A documentation tool, like `javadoc`, can generate helpful interface documentation for a class hierarchy; see the section, "Generating Package APIs with `javadoc`," which follows.

Exercises

1. Reread the Exercise from the section, "Abstract Classes," that displayed several geometric forms. Use the classes in that exercise to write classes for the hierarchy in Figure 13. Say that `class Form` goes as follows:

   ```
   /** Form  is the root of the geometric forms hierarchy */
   public abstract class Form { }
   ```

   Alter classes `Point` and `Shape` so that they fit into the hierarchy. Next, write the classes for `Line` and `Straight`. (Remember that a line has two end points. A straight line has no additional points, whereas jagged lines require additional points and curves require additional information.)

2. Write taxonomies of the following:

   (a) the people—teachers, students, and staff—who work in a school;

   (b) the different forms of motorized vehicles (cars, trucks, cycles);

   (c) forms of fruits and vegetables;

(d) forms of music;

(e) forms of dance;

(f) computer input/output devices;

(g) people who work for a candy company (managers, chefs, assembly-line workers, tasters, salespeople)

9.5.3 Frameworks and Abstract Classes

An abstract class is an "incomplete program," and we might use a collection of abstract classes to write a complex, powerful, but incomplete program that another programmer finishes by writing one or two concrete classes.

A *framework* is such an incomplete program—it is a collection of classes and interfaces organized into an architecture for a particular application area, such as building graphics windows, or generating animations, or building spreadsheets, or writing music, or creating card games. Some of the framework's classes are left abstract—incomplete. This is deliberate—a programmer uses the framework to build a specific graphics window (or a specific animation, or spreadsheet, or song, or game) by writing concrete classes that extend the abstract ones. The result is a complete application that creates what the programmer desires with little effort.

For example, say that someone wrote for us a framework for building graphics windows. Such a framework would contain classes that do the hard work of calculating colors, shapes, and sizes and displaying them on the computer console. Most importantly, the framework would also contain an abstract class, say, by the name of `class GraphicsWindow`, that already contains methods like `setSize`, and `setVisible` but lacks a `paint` method. A programmer would use the framework and `class GraphicsWindow` in particular to write concrete classes that extend `GraphicsWindow` with `paint` methods. By using the framework, the programmer generates graphics windows with minimal effort.

We have been doing something similar, of course, when we used Java's *Abstract Window Toolkit* (`java.awt`) and *Swing* (`javax.swing`) packages to build graphics windows. These two packages form a framework for a range of graphics applications, and by extending `class JPanel`, we "complete" the framework and create graphics windows with little effort on our own part.

The next chapter surveys the AWT/Swing framework.

9.6 Subtypes versus Subclasses

It is time to review the crucial distinctions between Java interfaces and subclasses:

- An interface defines a behavioral specification or "connection point" between classes or subassemblies that are developed separately. When a class `implements` an interface, the class provides, with its methods, the behavior promised by the interface.

- A subclass provides codings—implementations—of some methods that are omitted from the superclass. When a class `extends` another, it inherits the existing coded methods and provides codings for additional ones.

In a nutshell,

Interfaces list behaviors, subclasses list codings.

Therefore,

- Use an abstract class or a superclass when you are building a "family" of related classes whose internal structures are similar; the superclass holds the coding common to all the subclasses.
- Use an interface when you are connecting classes together, and you do not know or care how the classes will be coded.

Because interfaces and abstract classes have different purposes, it is acceptable to use both when a subassembly, as we saw in the case study with the varieties of card players.

One reason why interfaces and subclasses are confused is because both of them define subtype relationships. As noted earlier in the Chapter, if `class C implements I`, then data type `C` is a subtype of `I` written, `C <= I`. Similarly, if `class B extends A`, then `B <= A`. But remember that "subtype" is different from "subclass" — if `C <= D`, it does not mean that `C` and `D` are classes and `C` is a subclass of `D`. Indeed, `D` and even `C` might be interfaces.

9.7 `class Object` and Wrappers

In the section, "Class Hierarchies," we saw how collections of classes are grouped into hierarchies. The Java compiler forces such a hierarchy upon a user whether she desires it or not: Within the package `java.lang`, there is a `class Object`, which defines basic coding that all Java objects must have for construction in computer storage. The Java compiler automatically attaches `extends Object` to every class that does not already extend another.

For example, if we wrote

```
public class A {...}

public class B extends A {...}
```

the Java compiler treats the first class as if it were written

```
public class A extends Object {...}
```

The practical upshot of this transformation is `C <= Object`, for every class, `C`. In this way, `class Object` defines a type `Object` which is the "data type of all objects."

Some programmers exploit the situation by writing methods whose arguments use type `Object`, e.g., here is a class that can hold any two objects whatsoever:

```
public class Pair
{ Object[] r = new Object[2];

  public Pair(Object ob1, Object ob2)
  { r[0] = ob1;
    r[1] = ob2;
  }

  public Object getFirst()
  { return r[0]; }

  public Object getSecond()
  { return r[1]; }
}
```

For example,

```
Pair p = new Pair(new JPanel(), new int[3]);
Object item = p.getFirst();
if ( item instanceof JPanel )
   { (JPanel)item.setVisible(true); }
```

Because `class Pair` is written to hold and deliver objects of type `Object`, a cast is needed to do any useful work with the objects returned from its methods.

Of course, primitive values like integers, booleans, and doubles are not objects, so they cannot be used with `class Pair`. But it is possible to use `class Integer`, `class Boolean`, and `class Double` as so-called *wrappers* and embed primitive values into objects. For example, if we write

```
Integer wrapped_int = new Integer(3);
```

this creates an object of type `Integer` that holds 3. Now, we might use it with `class Pair`:

```
Pair p = new Pair(wrapped_int, new int[3]);
Object item = p.getFirst();
if ( item instanceof Integer )
   { System.out.println( ((Integer)item).intValue() + 4 ); }
```

In a similar way, we can "wrap" a boolean within a `new Boolean` and later use the `booleanValue()` method and wrap a double within a `new Double` and use the `doubleValue()` method.

Finally, since every object is built from a class that `extends Object`, every object inherits from `class Object` a method, `toString`, that returns a string representation of an object's "identity." For example, if we write

```
Integer wrapped_int = new Integer(3);
String s = "abc";
Pair p = new Pair(wrapped_int, s);
  System.out.println(wrapped_int.toString());
  System.out.println(s.toString());
  System.out.println(p.toString());
}
```

We receive this output:

```
3
abc
Pair@1f14c60
```

The third line is the string representation of `p`; it displays the type of `p`'s object as saved in computer storage and a coding, called a *hash code*, of the object's storage address. The `toString` method can prove useful for printing tracing information when locating program errors.

9.8 Packages

When you write an application or a subassembly that consists of multiple classes, it is best to keep the classes together in their own folder (disk directory). (If you have been using an IDE to develop your applications, the IDE has done this for you, under the guise of a "project name" for each application you write.) If you keep each application's classes in its own folder, you will better manage your continuously growing collection of files, and other programmers will find it easier to use your files, because they need only to remember the folder names where the application live.

A Java *package* is a folder of classes where the classes in the folder are marked as belonging together. We have used several Java packages already, such as `java.util`, `java.awt`, and `javax.swing`. When you write a program that requires components from a package named `P`, you must insert an `import P.*` statement at the beginning of your program. The `import` statement alerts the Java compiler to search inside package `P` for the classes your program requires.

We can make our own packages. Say that we wish to group the components written for bank accounting, listed in Figures 1 and 3, into a package named `Bank`. We do the following:

- Create a folder (called a "project package," if you are using an IDE) with the name `Bank`, and move the classes into this folder.
- Insert, as the first line of each class, the statement `package Bank`. For example, Figure 1 is revised to read,

  ```
  package Bank;
  /** BankAccountSpecification specifies the expected behavior of a
    *  bank account.   */
  public interface BankAccountSpecification
  {
    ... // the body of the interface remains the same
  }
  ```

  and Figure 3 is revised as follows:

  ```
  package Bank;
  /** BankAccount manages a single bank account; as stated in its
    * header line, it _implements_ the BankAccountSpecification:  */
  public class BankAccount implements BankAccountSpecification
  {
    ... // the body of the class remains the same
  }
  ```

- Compile each of the classes. If you are using an IDE, this step is the usual one. If you are using the JDK, you must first close the folder, and then you compile each class by mentioning both its folder and file names, e.g., `javac Bank\BankAccount.java`
- Say that the package you assembled contained a class that has a `main` method, and you wish to execute that class. To execute an application that is contained in a package, you may proceed as usual when you use an IDE.

  If you are using the JDK, then you must state both the package name and the class name to execute. For example, pretend that the `Bank` package contained a file, `MortgagePaymentApplication.java`, that has a `main` method. We compile this file like the others, e.g., `javac Bank\MortgagePaymentApplication.java`. We execute the class by typing `java Bank.MortgagePaymentApplication`. *Note the dot rather than the slash.*

Once a collection of classes is grouped in a package, other applications can *import* the package, just like you have imported `javax.swing` to use its graphics classes. For example, perhaps we write a new application, `class MyCheckingAccount`, so that it uses classes in package `Bank`. to do this, insert `import Bank.*` at the beginning of the class. This makes the components in package `Bank` immediately available, e.g.,

```
import Bank.*;
/** MyCheckingAccount contains methods for managing my account */
public class MyCheckingAccount
{ private BankAccount my_account = new BankAccount();
  private MortgagePaymentCalculator calculator
                         = new MortgagePaymentCalculator(my_account);
  ...
}
```

If other applications will import the `Bank` package, then the package must be placed where the applications can find it. You do this by adding `Bank`'s directory path to the Java compiler's *classpath*.

(*Begin Footnote:* A *classpath* is a list of paths to folders that hold packages. Whenever you compile and execute a Java program, a classpath is automatically used to locate the needed classes. The usual classpath includes paths to the standard packages, `java.lang`, `java.util`, etc. If you use an IDE, you can read and update the classpath by selecting the appropriate menu item. (Usually, you "Edit" the "Project Preferences" to see and change the classpath.) If you use the JDK, you must edit the *environment variable*, `CLASSPATH`, to change the path. *End Footnote*)

For example, if you created the package, `Bank`, as a folder within the folder, `C:\JavaPgms`, then add `C:\JavaPgms` to the Java compiler's classpath. If you do not wish to fight classpaths but wish to experiment with packages nonetheless, you can move the `Bank` folder into the folder where you develop your programs. (For example, if you do your development work in the folder, `A:\MyWork\JavaExperiments`, move `Bank` into that folder.) This also works when one package contains classes that use classes from another package—keep both packages within the same folder.

Exercise

Create a package from an application you have written. (Or, create the package, `Bounce`, from the classes of the moving ball animation in Figures 8 through 10, Chapter 7.)

9.8.1 Generating Package APIs with `javadoc`

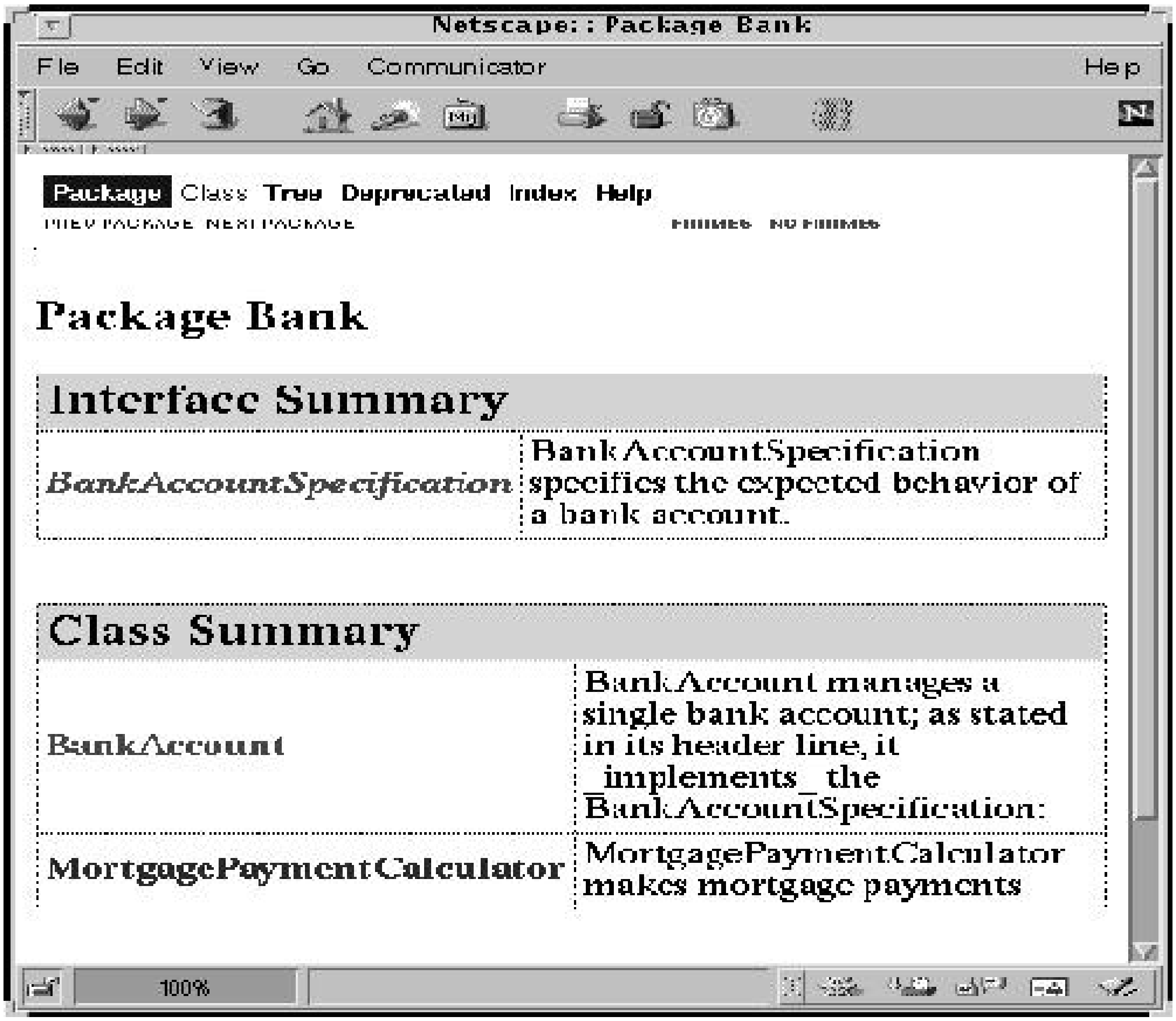
Recall that a class's *Application Programming Interface* (API) documents the class's interface in a readable format, say, as a web page. This can be done for a package, also. Given a package, `P`, located in a folder of the same name, we can type at the command line,

```
javadoc P
```

(If you use an IDE, consult the IDE's user guide for information about `javadoc`.) The `javadoc` program extracts the commentary from each class and creates a collection of

HTML pages, most notably, `package-summary.html`, which contains links to the API pages for each class in the package.

For example, here is the page created for the `Bank` package:

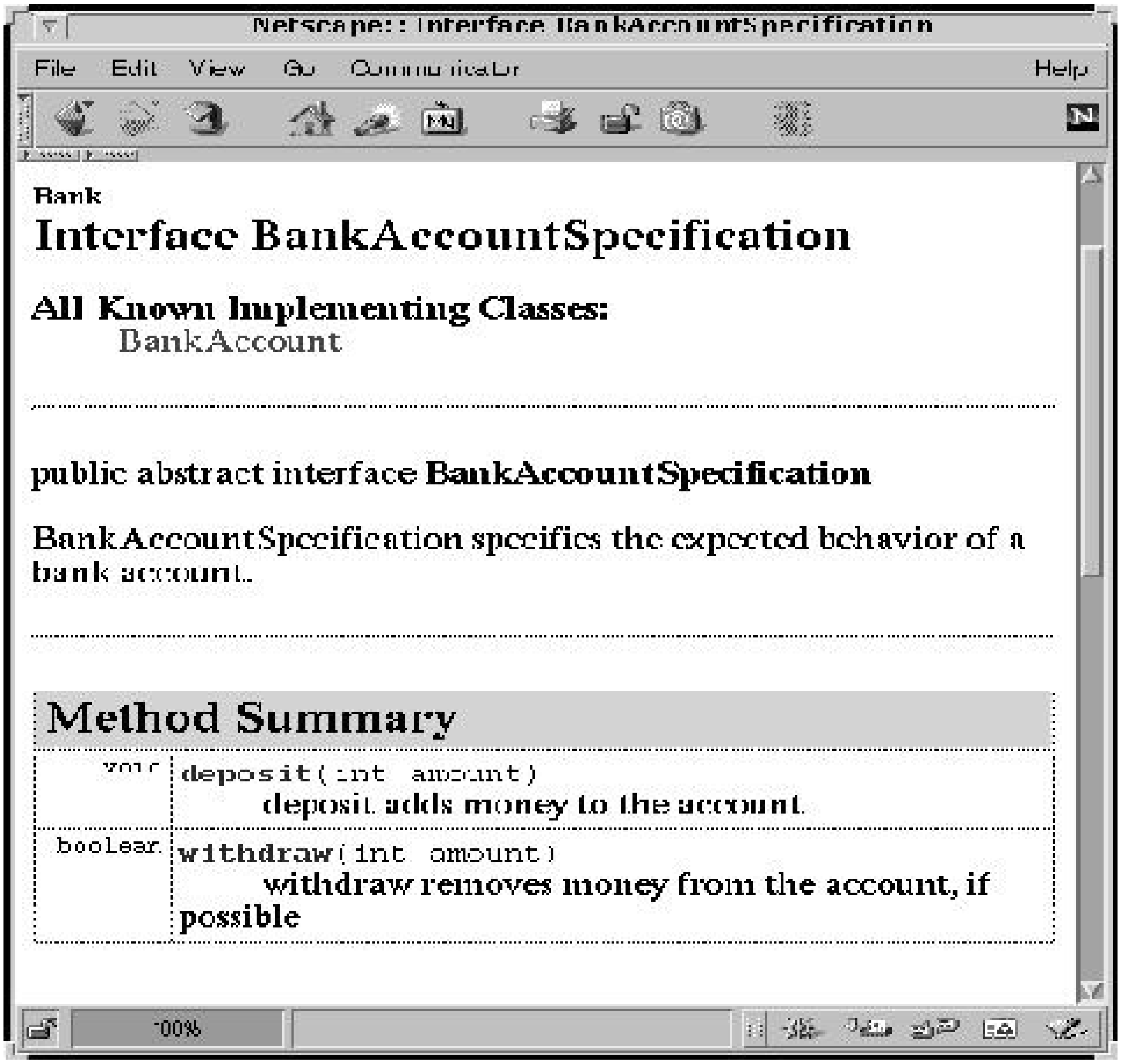

Netscape: : Package Bank
File Edit View Go Communicator Help
100%
Package Class Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package Bank

Interface Summary	
BankAccountSpecification	BankAccountSpecification specifies the expected behavior of a bank account.

Class Summary	
BankAccount	BankAccount manages a single bank account; as stated in its header line, it _implements_ the BankAccountSpecification:
MortgagePaymentCalculator	MortgagePaymentCalculator makes mortgage payments

By clicking on the link for, say, `BankAccountSpecification`, we see the documentation for the interface:



Bank

Interface BankAccountSpecification

All Known Implementing Classes:
BankAccount

public abstract interface BankAccountSpecification

BankAccountSpecification specifies the expected behavior of a bank account.

Method Summary

void	deposit(int amount) deposit adds money to the account.
boolean	withdraw(int amount) withdraw removes money from the account, if possible

Exercise

Use `javadoc` to generate the API documentation for a package you have created.

9.9 Case Study: An Adventure Game

(Note: This section can be skipped on first reading.)

Figure 9.14: Interfaces for an adventure game

```
/** RoomBehavior  defines the behavior of a room  */
public interface RoomBehavior
{ /** enter lets a player enter a room
    * @param p - the player who wishes to enter
    * @return whether the player sucessfully opened the door and entered. */
  public boolean enter(PlayerBehavior p);

  /** exit ejects a player from the room.
    * @param p - the player who wishes to leave the room  */
  public void exit(PlayerBehavior p);

  /** occupantOf  returns the identity of the room's occupant
    * @return the address of the occupant object;
    *   if room unoccupied, return null  */
  public PlayerBehavior occupantOf();
}

/** PlayerBehavior  defines the behavior of a player of an adventure game */
public interface PlayerBehavior
{ /** speak lets the player say one word
    * @return the word  */
  public String speak();

  /** explore attempts to enter a room and explore it
    * @param r - the room that will be explored
    * @return whether the room was successfully entered */
  public boolean explore(RoomBehavior r);
}
```

We finish the chapter by applying interfaces and inheritance to design a complex model whose components connect together in multiple, unpredictable ways: We are building an "adventure game," where players can enter and exit rooms. A player enters an unoccupied room by speaking the "secret word" that opens the room's door. A player exits a room whenever she chooses.

We want the adventure game to be as general as possible, so we retrict as little as possible the notions of "player" and "room." To start, we write specifications (Java `interface`s) that state the minimal expected behaviors of rooms and players. The interfaces might appear as in Figure 14. The two interfaces depend on each other for stating their respective behaviors, so the two interfaces must be compiled together. (To do this, place the files, `RoomBehavior.java` and `PlayerBehavior.java` in the same folder and compile one of them; the Java compiler will automatically compile both.)

Although we know nothing about the kinds of players in the adventure game, we can use the interfaces to write a basic class of room for the game; see Figure 15. A `BasicRoom` object remembers the address of the player that occupies it, and it is initialized with a `null` address for its occupant. When a `PlayerBehavior` object wishes to enter the room, it sends an `enter` message, enclosing its address as an argument. The `BasicRoom` is not so interesting, but it is a good "building block" for designing more elaborate rooms.

Next, we might write a class of player that remembers who it is and where it is. Figure 16 shows the class.

Method `explore` of `class Explorer` uses the `enter` method of `interface RoomBehavior` when it explores a room. The novelty within the method is the keyword, `this`. When we write,

```
boolean went_inside = r.enter(this);
```

the keyword, `this`, computes to the address of *this very object* that sends the message to object `r`. Whenever you see the keyword, `this`, read it as "this very object that is sending the message."

To understand this, consider this example:

```
RoomBehavior[] ground_floor = new RoomBehavior[4];
ground_floor[0] = new BasicRoom("kitchen", "pasta");
ground_floor[3] = new BasicRoom("lounge", "swordfish");

Explorer harpo = new Explorer("Harpo Marx", "swordfish");
Explorer chico = new Explorer("Chico Marx", "tomato");
boolean success = harpo.explore(ground_floor[3]);
```

Upon completion of the last statment, where the `harpo.explore` method is invoked,

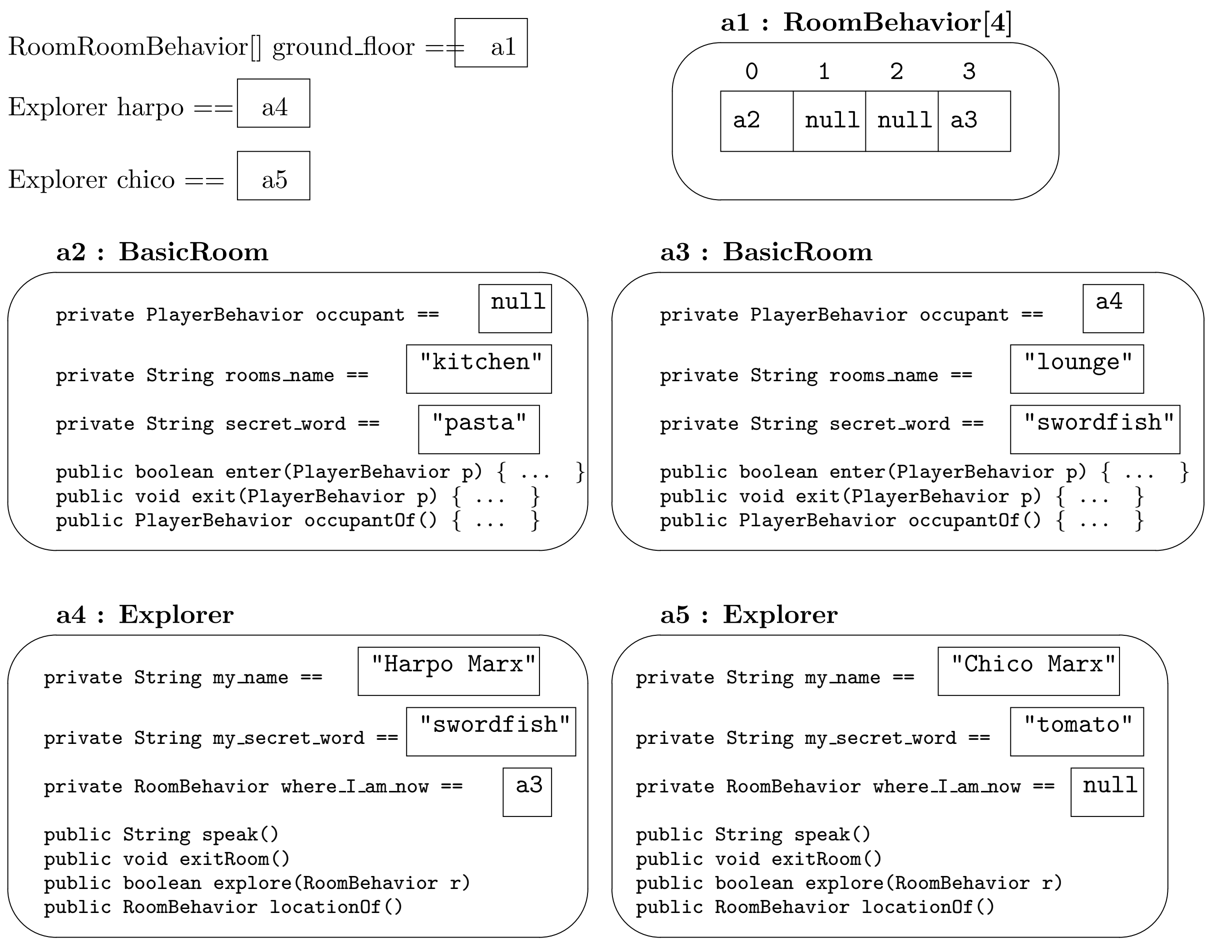
Figure 9.15: room for an adventure game

```
/** BasicRoom models a room that can have at most one resident at a time */
public class BasicRoom implements RoomBehavior
{ private PlayerBehavior occupant;  // who is inside the room at the moment
  private String rooms_name;
  private String secret_word;  // the password for room entry

  /** Constructor BasicRoom builds the room.
    * @param name - the room's name
    * @param password - the secret word for entry into the room */
  public BasicRoom(String name, String password)
  { occupant = null;  // no one is in the room initially
    rooms_name = name;
    secret_word = password;
  }

  public boolean enter(PlayerBehavior p)
  { boolean result = false;
    if ( occupant == null  &&  secret_word.equals(p.speak())  )
       { occupant = p;
         result = true;
       }
    return result;
  }

  public void exit(PlayerBehavior p)
  { if ( occupant == p )  // is  p  indeed in this room at the moment?
       { occupant = null; }
  }

  public PlayerBehavior occupantOf()
  { return occupant; }
}
```

Figure 9.16: a player that can explore rooms

```
/** Explorer models a player who explores rooms */
public class Explorer implements PlayerBehavior
{ private String my_name;
  private String my_secret_word;  // a password for entering rooms
  private RoomBehavior where_I_am_now;  // the room this object occupies

  /** Constructor Explorer builds the Player
    * @param name - the player's name
    * @param word - the password the player can speak */
  public Explorer(String name, String word)
  { my_name = name;
    my_secret_word = word;
    where_I_am_now = null;  // player does not start inside any room
  }

  public String speak()
  { return my_secret_word; }

  /** exitRoom causes the player to leave the room it occupies, if any */
  public void exitRoom()
  { if ( where_I_am_now != null )
       { where_I_am_now.exit(this);   // exit the room
         where_I_am_now = null;
       }
  }

  public boolean explore(RoomBehavior r)
  { if ( where_I_am_now != null )
       { exitRoom(); }  // exit current room to go to room  r:
    boolean went_inside = r.enter(this);  // ``this'' means ``this object''
    if ( went_inside )
       { where_I_am_now = r; }
    return went_inside;
  }

  /** locationOf returns the room that is occupied by this player */
  public RoomBehavior locationOf()
  { return where_I_am_now; }
}
```

we have this storage configuration, which shows that object `a4` has entered room `a3`:


RoomRoomBehavior[] ground_floor == a1
Explorer harpo == a4
Explorer chico == a5
a1 : RoomBehavior[4]
0 1 2 3
a2 null null a3
a2 : BasicRoom
private PlayerBehavior occupant == null
private String rooms_name == "kitchen"
private String secret_word == "pasta"
public boolean enter(PlayerBehavior p) { ... }
public void exit(PlayerBehavior p) { ... }
public PlayerBehavior occupantOf() { ... }
a3 : BasicRoom
private PlayerBehavior occupant == a4
private String rooms_name == "lounge"
private String secret_word == "swordfish"
public boolean enter(PlayerBehavior p) { ... }
public void exit(PlayerBehavior p) { ... }
public PlayerBehavior occupantOf() { ... }
a4 : Explorer
private String my_name == "Harpo Marx"
private String my_secret_word == "swordfish"
private RoomBehavior where_I_am_now == a3
public String speak()
public void exitRoom()
public boolean explore(RoomBehavior r)
public RoomBehavior locationOf()
a5 : Explorer
private String my_name == "Chico Marx"
private String my_secret_word == "tomato"
private RoomBehavior where_I_am_now == null
public String speak()
public void exitRoom()
public boolean explore(RoomBehavior r)
public RoomBehavior locationOf()

Because the object named `harpo` lives at address `a4`, the invocation, `harpo.explore(ground_floor[3])`, computes to `a4.explore(a3)`. Within `a4`'s `explore` method, there is the invocation, `r.enter(this)`, which computes to `a3.enter(a4)`, because `this` computes to `a4`, the address of this object in which the invocation appears.

Exercises

1. Write this class:

   ```
   class Dungeon implements RoomBehavior
   ```

   so that any object with `PlayerBehavior` can enter the room; no object that enters the room can ever exit it; and when asked by means of its `occupantOf` method, the `Dungeon` replies that no one is in the room.

Figure 9.17: interface for treasures

```
/** TreasureProperty  defines a treasure object */
public interface TreasureProperty
{ /** contentsOf  explains what the treasure is
    * @return the explanation  */
  public String contentsOf();
}
```

2. After working the previous exercise, construct an array of size 3, so that the first two rooms in the array are `BasicRoom` objects named `"kitchen"` and `"lounge"`, respectively, and the third object is a `Dungeon`. Then, construct

   ```
   Explorer harpo = new Explorer("Harpo Marx", "swordfish");
   ```

   and write a for-loop that makes `harpo` systematically try to enter and then exit each room in the array.

3. A close examination of Figures 15 and 16 shows us that it is possible for one `Explorer` object to occupy simultaneously multiple `BasicRooms`. Write a sequence of Java statements that makes this happen. (Hint: in the example at the end of the section, make both rooms use the same password.)

   How can we make a `BasicRoom` object "smart" enough so that it refuses to let a player be its occupant when the player currently occupies another room? To do this, add this method to `interface PlayerBehavior`:

   ```
   /** locationOf returns the room that is occupied by this player */
   public RoomBehavior locationOf();
   ```

   Now, rewrite the `enter` method of `BasicRoom`.

9.9.1 Interfaces and Inheritance Together

Classes `BasicRoom` and `Explorer` are too simplistic for an interesting game. We might design rooms that contain "treasures" so that players can take the treasures from the rooms they enter.

Since a treasure might be a variety of things, it is simplest to define an interface that gives the basic property of being a treasure. See Figure 17. Next, we can define an interface that defines what it means for an object to hold a treasure. We might write the interface in Figure 18.

A room that possesses an entry and exit as well as a treasure must implement both `interface RoomBehavior` as well as `interface Treasury`. It is indeed acceptable for one class to implement two interfaces, and it might look like this:

Figure 9.18: interface for a treasury

```
/** Treasury describes the method an object has for yielding
  *  a treasure  */
public interface Treasury
{ /** yieldTreasure surrenders the room's treasure
    * @param p - the player who requests the treasure
    * @return the treasure (or null, if the treasure is already taken) */
  public TreasureProperty yieldTreasure(PlayerBehavior p);
}
```

```
public class Vault implements RoomBehavior, Treasury
{ private PlayerBehavior occupant;  // who is inside the room at the moment
  private String rooms_name;
  private String secret_word;  // the password for room entry

  private TreasureProperty valuable;  // the treasure item held in this room

  /** Constructor Vault builds the room.
    * @param name - the room's name
    * @param password - the secret word for entry into the room
    * @param item - the treasure to be saved in the room  */
  public Vault(String name, String password, TreasureProperty item)
  { occupant = null;
    rooms_name = name;
    secret_word = password;
    valuable = item;
  }

  public boolean enter(PlayerBehavior p)
  { ... }  // insert coding from Figure 15

  public void exit(PlayerBehavor p)
  { ... }  // insert coding from Figure 15

  public PlayerBehavior occupantOf()
  { ... }  // insert coding from Figure 15

  public TreasureProperty yieldTreasure(PlayerBehavior p)
  { TreasureProperty answer = null;
    if ( p == occupant )
       { answer = valuable;
         valuable = null;
```

Figure 9.19: VaultRoom defined by inheritance

```
/** VaultRoom is a room that holds a treasure---it is constructed from
  *   class BasicRoom, by means of inheritance, plus the structure below.  */
public class VaultRoom extends BasicRoom implements Treasury
   // because BasicRoom implements RoomBehavior, so does VaultRoom
{ private TreasureProperty valuable;  // the treasure item held in this room

  /** Constructor VaultRoom builds the room.
    * @param name - the room's name
    * @param password - the secret word for entry into the room
    * @param item - the treasure to be saved in the room  */
  public VaultRoom(String name, String password, TreasureProperty item)
  { // first, invoke class  BasicRoom's  constructor method:
    super(name, password);  // ``super'' means ``superclass''
    valuable = item;
  }

 public TreasureProperty yieldTreasure(PlayerBehavior p)
  { TreasureProperty answer = null;
    if ( p == occupantOf() ) // invokes the  occupantOf  method in  BasicRoom
                             // You can also state,  super.occupantOf()
                             // Another format is,   this.occupantOf()
       { answer = valuable;
         valuable = null;
       }
    return answer;
  }
}
```

```
      }
    return answer;
  }
}
```

In the general case, a class can implement an arbitrary number of interfaces; the interfaces are listed in the `implements` clause, separated by commas, within the class's header line.

The coding of `class Vault` possesses a major flaw: It copies the codings of methods already present in `class BasicRoom`. For this reason, we should rewrite the class so that it uses inheritance to use `BasicRoom`'s methods. Figure 19 shows the corrected coding.

The header line of `class VaultRoom` indicates that `VaultRoom extends BasicRoom`; this makes `VaultRoom` a subclass of the superclass `BasicRoom`. The class also implements the interface, `Treasury`, because it has a `yieldTreasure` method that matches the one in that interface. (Indeed, the class also implements `interface RoomBehavior`, because it `extends BasicRoom` and `BasicRoom` implements `RoomBehavior`.)

Within `VaultRoom`'s constructor method, we see `super(name, password)`, which invokes the constructor method of the superclass, ensuring that the private fields within `BasicRoom` are initialized. (Recall that when the `super`-constructor is invoked, it must be invoked as the *first* statement in the subclass's constructor.)

Within the `yieldTreasure` method, we see an invocation of the `occupantOf` method of `BasicRoom`:

```
if ( p == occupantOf() )
```

The invocation asks this object to execute its own `occupantOf` method to learn the player that occupies it. The invocation is required because the field `occupant` is declared as private to `class BasicRoom`, so subclass `VaultRoom` cannot reference it directly.

As the comment next to the invocation states, we can also invoke this particular method by

```
if ( p == super.occupantOf() )
```

This format asserts that the `occupantOf` method must be located in a superclass part of this object. (If the method is not found in a superclass of `VaultRoom`, the Java compiler will announce an error.) Finally, it is also acceptable to state

```
if ( p == this.occupantOf() )
```

which is equivalent to the first invocation and documents that the invocation message is sent to this very object.

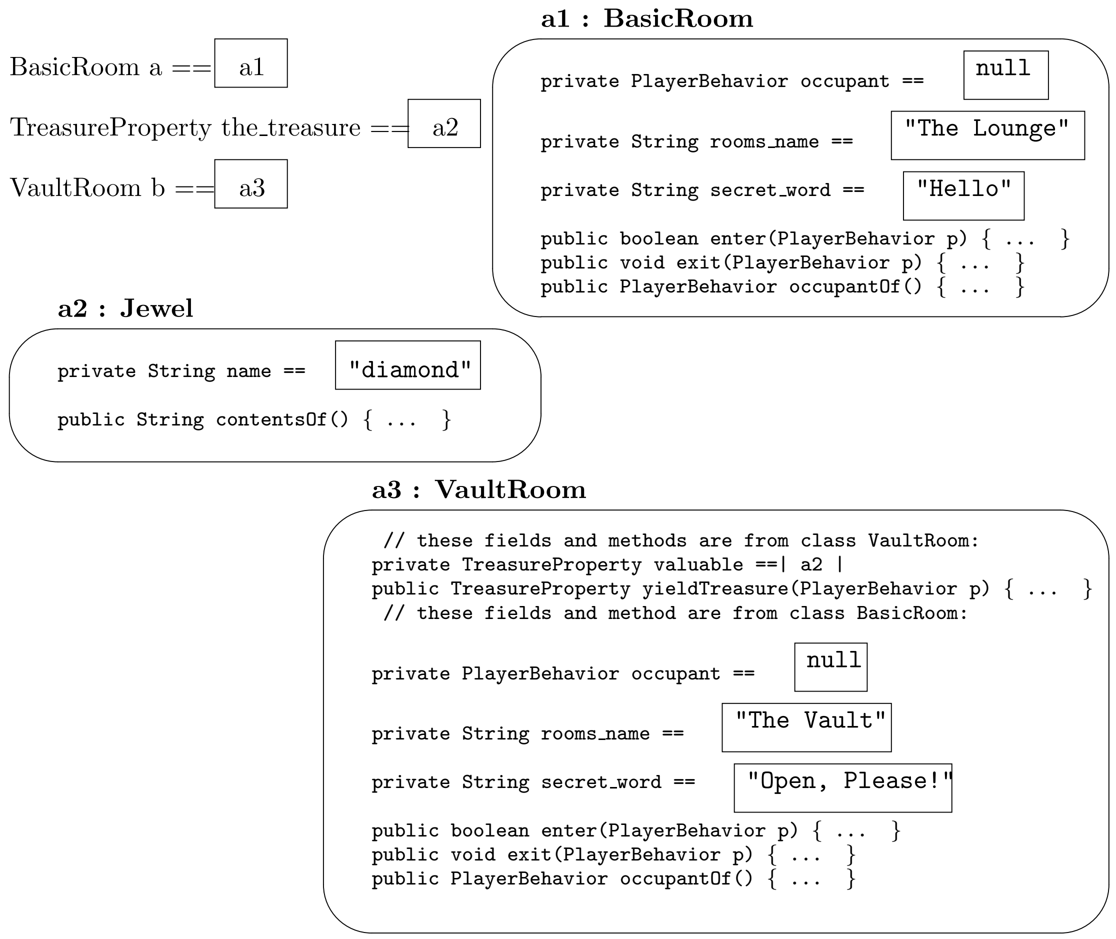
Using the above classes, say that we construct one `BasicRoom` object and one `VaultRoom` object:

```
BasicRoom a = new BasicRoom("The Lounge", "Hello");

TreasureProperty the_treasure = new Jewel("diamond");
VaultRoom b = new VaultRoom("The Vault", "Open, please!", the_treasure);
```

where we define `class Jewel` this simply:

```
public class Jewel implements TreasureProperty
{ private String name;  // the name of the jewel

  public Jewel(String id)
  { name = id; }

  public String contentsOf()
  { return name; }
}
```

Here is a picture of storage after the statements execute:



```
BasicRoom a == a1
TreasureProperty the_treasure == a2
VaultRoom b == a3

a1 : BasicRoom
  private PlayerBehavior occupant == null
  private String rooms_name == "The Lounge"
  private String secret_word == "Hello"
  public boolean enter(PlayerBehavior p) { ... }
  public void exit(PlayerBehavior p) { ... }
  public PlayerBehavior occupantOf() { ... }

a2 : Jewel
  private String name == "diamond"
  public String contentsOf() { ... }

a3 : VaultRoom
   // these fields and methods are from class VaultRoom:
  private TreasureProperty valuable ==| a2 |
  public TreasureProperty yieldTreasure(PlayerBehavior p) { ... }
   // these fields and method are from class BasicRoom:
  private PlayerBehavior occupant == null
  private String rooms_name == "The Vault"
  private String secret_word == "Open, Please!"
  public boolean enter(PlayerBehavior p) { ... }
  public void exit(PlayerBehavior p) { ... }
  public PlayerBehavior occupantOf() { ... }
```

The objects in storage show us that the structure of `class BasicRoom` is used to build the objects at addresses `a1` and `a3`. In particular, the `VaultRoom` object at `a3` inherited the `BasicRoom` structure plus its own—all the methods of a `BasicRoom` object can be used with a `VaultRoom` object.

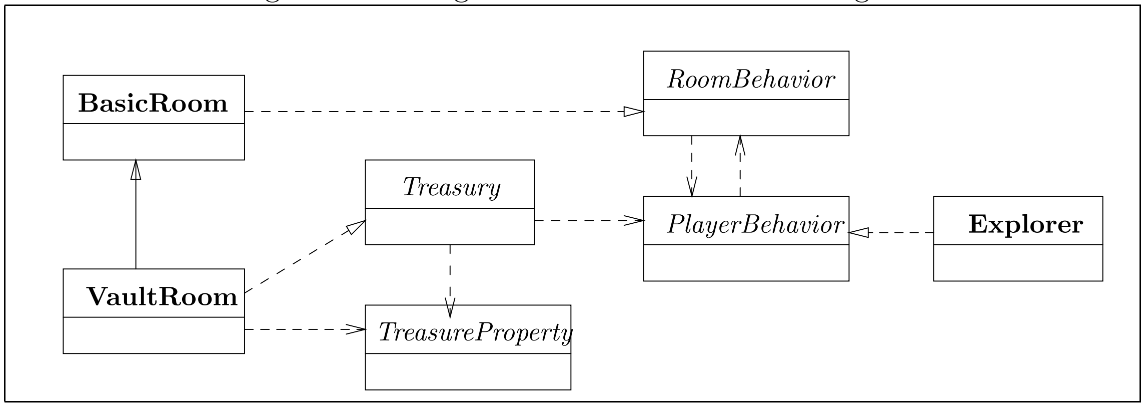
Because `VaultRoom extends BasicRoom`, the resulting data types are related by subtyping—data type `VaultRoom` is a subtype of data type `BasicRoom`. Further, since `BasicRoom` is a subtype of `RoomBehavior`, then by *transitivity*, `VaultRoom` is a subtype of `RoomBehavior` as well. Here is a drawing of the situation:

```
           <=  BasicRoom  <=  RoomBehavior
VaultRoom
           <=  Treasury
```

The Java compiler uses these subtyping relationships to verify that `VaultRoom` objects are sent appropriate messages.

Finally, the diagram for the classes defined in this and the previous sections appears in Figure 20. The diagram shows how interfaces act as the connection points

Figure 9.20: diagram for model of adventure game



for the concrete classes; it shows that the classes `BasicRoom` and `VaultRoom` form a subassembly that is kept separate from `Explorer`. We also see that a `VaultRoom` has `RoomBehavior` and depends on `PlayerBehavior`. Indeed, as we study the diagram further, we might conclude that an `Explorer` might be profitably extended by methods that fetch treasures from the rooms explored; this is left for the Exercises.

Exercises

1. Write a class that implements `TreasureProperty`:

   ```
   public class GoldCoin implements TreasureProperty
   ```

   and construct a `Vault` object with the name, `"lounge"` and the password, `"swordfish"`, to hold an object constructed from `class GoldCoin`.

2. Modify the method, `explore`, in Figure 10 so that once the player successfully enters the room, it tries to extract the treasure from it. Note that not all rooms will implement interface `Treasury`, however. (Hint: use `instanceof`.) Then, construct an explorer named `"Harpo Marx"` that explores the `"lounge"`.

3. Here is an interface:

   ```
   /** TreasureHunterBehavior describes the behavior of a player that tries
     *  to take treasures  */
   public interface TreasureHunterBehavior
   { /** takeTreasure  tries to extract the treasure from the current room
       *  that this player occupies.
       * @return true if the treasure was succesfully extracted from the
       *   room and saved by this player; return false otherwise */
     public boolean takeTreasure();
   ```

```
  /** getTreasures  returns an array of all the treasures located so far
    *  by this player
    * @return the array of treasures  */
  public TreasureProperty[] getTreasures();
}
```

Write this class:

```
/** TreasureHunter explores rooms and saves all the treasures it extracts
  *  from the rooms.  */
public class TreasureHunter extends Explorer implements TreasureHunterBehavior
```

4. The `TreasureHunter` defined in the previous exercise does not automatically try to take a treasure each time it successfully enter a room. (The `TreasureHunter` must be sent an `enter` message followed by a `takeTreasure` message.)

   We can create a treasure hunter that has more aggressive behavior: First write this method:

```
/** explore  attempts to enter a room and explore it.  If the room is
  * successfully entered, then an attempt is made to take the treasure
  * from the room and keep it (if the room indeed holds a treasure).
  * @param r - the room that will be explored
  * @return whether the room was successfully entered */
public boolean explore(RoomBehavior r)
```

   Now, insert your coding of the new `explore` into this class:

```
public class RuthlessHunter extends TreasureHunter
{ ...

  public boolean explore(RoomBehavior r)
  { ... }
}
```

   When we construct a `RuthlessHunter`, e.g.,

```
VaultRoom bank = new VaultRoom("Bank", "bah!", new Jewel("ruby"));
RuthlessHunter bart = new RuthlessHunter("Black Bart", "bah!");
bart.explore(bank);
```

   The new `explore` method of the `RuthlessHunter` executes instead of the old, same-named method in the superclass, `Explorer`. We say that the new `explore` method *overrides* the old one.

   Construct this object:

```
Explorer indiana = new Explorer("Indiana Jones", "hello");
indiana.explore(bank);
```

Which version of `explore` method executes? Method overriding is studied in the "Beyond the Basics" section at the end of this Chapter.

9.9.2 Inheritance of Interfaces

Interfaces can be connected by inheritance, just like classes are. For example, perhaps we want to ensure that any object that has `Treasury` behavior must also have `RoomBehavior`. This can be enforced by changing the interface for the `Treasury` to inherit `RoomBehavior`:

```
/** TreasuryBehavior describes a room that holds a treasure */
public interface TreasuryBehavior extends RoomBehavior
{ /** yieldTreasure surrenders the room's treasure
    * @param p - the player who requests the treasure
    * @return the treasure (or null, if the treasure is already taken) */
  public TreasureProperty yieldTreasure(PlayerBehavior p);
}
```

Because it `extends RoomBehavior`, interface `TreasuryBehavior` requires all the methods of `RoomBehavior` plus its own. It so happens that `class VaultRoom` in Figure 13, because it `extends BasicRoom`, also is capable of implementing `TreasuryBehavior`. Indeed, we can change its header line accordingly:

```
public class VaultRoom extends BasicRoom implements Treasury, TreasuryBehavior
```

of course, if we no longer use `interface Treasury` in the game's architecture, we can shorten the header line to read merely

```
public class VaultRoom extends BasicRoom implements TreasuryBehavior
```

Because `TreasuryBehavior extends RoomBehavior`, the expected subtyping relation is established: `TreasuryBehavior <= RoomBehavior`. The Java compiler uses the subtyping to check compatibility of classes to interfaces.

9.10 Summary

This chapter has presented four constructions that can improve the design and implementation of programs: the interface, subclass, abstract class, and package.

New Constructions

Here are examples of the constructions introduced in this chapter:

- *interface* (from Figure 1):

```
public interface BankAccountSpecification
{ public void deposit(int amount);
  public boolean withdraw(int amount);
}
```

- *inheritance* (from Figure 10):

```
public class HumanPlayer extends CardPlayer
{ public HumanPlayer(int max_cards)
  { super(max_cards); }

  public boolean wantsACard()
  { String response = JOptionPane.showInputDialog
                        ("Do you want another card (Y or N)?");
    return  response.equals("Y");
  }
}
```

- *abstract class* (from Figure 9):

```
public abstract class CardPlayer implements CardPlayerBehavior
{ private Card[] my_hand;
  private int card_count;

  public CardPlayer(int max_cards)
  { my_hand = new Card[max_cards];
    card_count = 0;
  }

  public abstract boolean wantsACard();  // method will be written later

  public void receiveCard(Card c)
  { my_hand[card_count] = c;
    card_count = card_count + 1;
  }
}
```

- *package*:

```
package Bank;
/** BankAccount manages a single bank account; as stated in its
  * header line, it _implements_ the BankAccountSpecification:  */
public class BankAccount implements BankAccountSpecification
{ ... }
```

- `instanceof` operation:

```
Record mystery_record = db.find(mystery_key);
if ( mystery_record instanceof BasicPerson )
     { System.out.println( ((BasicPerson)mystery_record).nameOf() ); }
else { System.out.println("unknown record type"); }
```

- `super`:

```
public class PersonAddress extends Person
{ private String address;

  public PersonAddress(String the_name, String the_addr)
  { super(the_name);  // this gives  the_name  to  Person's  constructor
    address = the_addr;
  }

  public String getAddress()
  { return address; }
}
```

New Terminology

- *interface*: a specification of the behaviors (methods) expected of a class. A Java `interface` is a named collection of header lines of public methods.
- *implementing an interface*: writing a class that contains methods whose header lines match the ones named in the interface.
- *inheritance*: writing a class that includes in itself the fields and methods of an existing class and adding new ones. The keyword, `extends`, precedes the name of the existing class that will be included in the new one.
- *abstract class*: a class that lacks some of its methods; the missing methods are noted by a header line that contains the keyword, `abstract`.
- *concrete class*: a “normal” class, all of whose methods have bodies.

- *package*: a collection of classes that are grouped together in a folder and labelled with a common name.
- *subtyping relationship*: a relationship between two data types; We write `C <= D` to state that `C` is a subtype of `D`, meaning that `C`-typed values can be used in any context where a `D`-typed values is expected.
- *run-time data type*: the data type that is saved inside an object when the object is constructed in computer storage. The saved data type is the name of the class from which the object was constructed.
- `instanceof`: a Java operation that examines the run-time data type of an object; `ob instanceof C` returns true exactly when the run-time data type of `ob` is a subtype of `C`.
- `super`: the statement that invokes the constructor method of a superclass within the constructor method of its subclass.
- *abstract method*: a method without its body found in an abstract class. The method's body is supplied in a subclass of the abstract class.
- *class hierarchy*: a collection of abstract and concrete classes, arranged by their super/subclass relationships, usually depicted as a tree structure.
- *framework*: a collection of classes designed to be augmented with only a few additional classes to construct complete applications in a problem domain.
- `class Object`: a built-in Java class that is automatically a superclass to all other classes.
- *wrapper class*: a class whose primary purpose is to hold a single primitive value, thereby allowing the primitive value to be used as if it were an object, e.g., `new Integer(2)` makes `2` into an object.

Points to Remember

- Interfaces are used to specify the behavior of a class that to be written; other classes can refer to the interface in their codings. Interfaces are also used to specify "connection points" to which subassemblies of an application may connect.
- A class *implements* an interface by having methods whose header lines match the ones named in the interface.
- A Java `interface` defines a data type name, and when a class implements the interface, the class's data type name is a subtype of the interface name.

- Just as you have written graphics-window classes that `extend JPanel`, you may write your own class `C` and extend it by `class D extends C`. This makes `D` a subclass of `C`, meaning that all of `C`'s structure is included within the structure of a `D`-constructed object. Further, data type `D` is a subtype of `C`.
- An abstract class is used when you wish to specify a partially written class, which has codings (bodies) for some of its methods. The classes that extend the abstract class use the methods coded in the abstract class.
- Use packages to group together collections of classes that are likely to be used together.

9.11 Programming Projects

1. Return to an application that you wrote in response to a Programming Project in Chapter 7 or 8. Redesign and reimplement the application with the assistance of interfaces and abstract classes. (If you did not work a substantial project from either of those two chapters, then build the library database application described in Project 6, Chapter 8.)
2. Build a computerized "adventure game," where one or more players explore rooms and collect treasures. Rooms are entered and exited through doors, and every door connects to a passageway, which itself leads to zero or more doors to other rooms. To orient herself, a player can ask a room its name. Invent additional rules for the game (e.g., a player can leave behind in a room a "message" for another player to find; treasures have point values; a player "loses energy" as it goes from room to room and must find "food" to eat in a room to regain energy, etc.)
3. Say that a company that sells chocolates, and the company requires a database that remembers its salespeople and how many chocolates each has sold. The database must also remember the managers of the salespeople and how well each managers' people have done at selling chocolates. Design and implement the database; use interfaces where appropriate.
4. Design and build a telephone directory database program. The program must manage a collection of patrons and their telephone numbers. Patrons are either individuals or businesses; telephone numbers are either listed or unlisted. Input to the program is a sequence of queries. Queries can take these forms:
    - an individual's name or a business's name or a name that might be either. (The program prints the patrons with that surname and their telephone numbers.)

- a telephone number (The program prints the patron that owns that number.)

Remember that a patron might own more than one telephone number and that unlisted numbers cannot be included in the response to a query that supplies a patron's name.

5. Say that a bank deals with two forms of customers: regular and preferred. (A preferred customer is charged no service charge on her accounts.) The bank offers three kinds of accounts: checking, savings, and checking-with-interest. (See the Exercises for the "Abstract Classes" section for details about the three forms of accounts.)

   Design and implement two databases: one for the bank's accounts and one for the bank's customers. Remember that a customer might own more than one form of bank account, and every bank account has an owner, which is a customer.

6. Construct a "framework" to help a programmer build card games where one human player interacts with a dealer and competes against zero or more computerized players.

7. One of the Projects in Chapter 8 suggested that you implement a database program for maintaining a sales inventory. Each sales item had this associated information: item's name, id number, wholesale price, retail price, and quantity in stock.

   Extend this problem as follows: Say that the inventory is for an automobile company (or computer company or bicycle company or ...) that services several, related models of car (computer, bicycle). The parts inventory must be organized so that each part is associated with the models of car that use it. Design a database that implements the parts inventory so that a user can obtain (among other output data), for a specific model of car, the portions of a car, the parts contained in each portion, and the prices of all the parts.

9.12 Beyond the Basics

9.12.5 Semantics of Overloading

The optional sections that follow examine the consequences that arise when the same name is given to more than one method. There are two variants of this phenomenon—overriding and overloading

9.12.1 Subclasses and Method Overriding

As noted earlier in this chapter, we used inheritance to construct graphics panels:

```
public class MyPanel extends JPanel
{ ...
  public void paintComponent(Graphics g) { ... }
  ...
}
```

The `paintComponent` method contains the instructions for painting on the panel.

There is an important detail which was skipper earlier: *There already is a method named* `paintComponent` *inside* `class JPanel`, but the method does nothing! Nonetheless, the Java language lets us write a new, second coding of `paintComponent`, which will be used with `new MyPanel` objects. The new coding of `paintComponent` *overrides* the useless version.

(*Begin Footnote:* Perhaps `class JPanel` should have been written as an abstract class, where its `paintComponent` method should have been written as `public abstract void paintComponent(Graphics g)`. But `class JPanel` is written so that its "default" `paintComponent` lets a beginner easily construct a blank panel. *End Footnote*)

When one writes a class, `C2`, that extends a class, `C1`, and `class C2` contains a method whose name and formal parameters are exactly the same one in `class C1`, then we say that the new method *overrides* the same-named method in `C1`. Here is a small example: Given `class C`:

```
public class C
{ public C() { }

  public int add(int i, int j)
  { return  i + j; }
}
```

we can extend `C` and override its method:

```
public class D extends C
{ public D() { }

  public int add(int i, int j)
  { return  i * j; }
  public int sub(in i, int j)
  { return  i / j; }
}
```

The `add` method in `D` overrides the one in `C`. The intent is: `new C()` objects use the `add` method coded in `class C`, and `new D()` objects use the `add` method in `class D`. If we write these statements,

```
C a = new C();
D b = new D();
System.out.println(b.add(3, 2));
System.out.println(a.add(3, 2));
```

`6` and then `5` appear on the display.

Method overriding is intended for "method improvement," and the situation with `JPanel`'s `paintComponent` method is typical: An existing class has an unexciting method that can be improved within a subclass. Method override is also used to make a subclass's method "more intelligent" by using fields that are declared only in the subclass. But method overriding can be complex, so we study more examples.

Figure 21 shows two classes that model bank accounts. A `BasicAccount` allows only deposits (an escrow account, which holds one's loan payment money, might be an example); a `CheckingAccount` ("current" account) is a `BasicAccount` extended with the ability to make withdrawals.

There is a problem: `CheckingAccount`'s `withdraw` method must alter the private `balance` field of `BasicAccount`. Java allows such behavior, if the field in the superclass is relabelled as `protected`. (Public access is still disallowed of a `protected` field, but subclasses—and alas, other classes in the same package as these two—can alter the field.)

This arrangement is not completely satisfactory, because it means that `class CheckingAccount` depends on the internals of `class BasicAccount`, so our freedom to alter and improve the components of `BasicAccount` is restricted. (Some people say this situation "breaks encapsulation," because the internals are "exposed" to another class.) It might be be better to redesign the classes in the Figure so that they both extend an abstract class, but we press on.

Next, Figure 22 introduces a `class AccountWithInterest` that models a checking account that pays interest, provided that the balance in the account never falls below a stated minimum. Here, the `withdraw` method must be "more intelligent" and monitor whether a withdrawal causes the account's balance to fall below the minimum amount for interest payment, and the new version of `withdraw` overrides the one in the superclass.

The body of the new, overriding method uses the invocation, `super.withdraw(amount)`, to make the physical removal of the `amount`: The keyword, `super`, forces the method

Figure 9.21: two classes for bank accounts

```
/** BasicAccount is a bank account that holds money */
public class BasicAccount
{ protected int balance;  // the money; note the keyword, ``protected''

  /** BasicAccount creates the account
    * @param initial_amount - the starting balance */
  public BasicAccount(int initial_amount)
  { balance = initial_amount; }

  /** deposit adds money to the account
    * @param amount - the amount to be deposited  */
  public void deposit(int amount)
  { if ( amount > 0 )
       { balance = balance + amount; }
  }

  /** balanceOf returns the current balance
    * @return the balance  */
  public int balanceOf()
  { return balance; }
}


/** CheckingAccount is a basic account from which withdrawals can be made */
public class CheckingAccount extends BasicAccount
{
  /** CheckingAccount creates the account
    * @param initial_amount - the starting balance */
  public CheckingAccount(int initial_amount)
  { super(initial_amount); }

  /** withdraw removes money from the account, if possible
    * @param amount - the amount to be removed
    * @return true, only if the balance is enough to make the withdrawal  */
  public boolean withdraw(int amount)
  { boolean outcome = false;
    if ( amount <= balance )
       { balance = balance - amount;
         outcome = true;
       }
    return outcome;
  }
}
```

Figure 9.22: checking account with interest

```
/** AccountWithInterest models an interest bearing checking account */
public class AccountWithInterest extends CheckingAccount
{ int minimum_balance;  // the amount required to generate an interest payment
  boolean eligible_for_interest;  // whether the account has maintained
                                  //  the minimum_balance for this time period

  /** AccountWithInterest  creates the account
    * @param required_minimum - the minimum balance that must be maintained
    *    to qualify for an interest payment
    * @param initial_amount - the starting balance */
  public AccountWithInterest(int required_minimum, int initial_balance)
  { super(initial_balance);
    minimum_balance = required_minimum;
    eligible_for_interest = (initial_balance > minimum_balance);
  }

  /** withdraw removes money from the account, if possible
    * @param amount - the amount to be removed
    * @return true, only if the balance is enough to make the withdrawal  */
  public boolean withdraw(int amount)
  { boolean ok = super.withdraw(amount);
    eligible_for_interest = eligible_for_interest    // is minimum maintained?
                            &&  balanceOf() > minimum_balance;
    return ok;
  }

  /** computeInterest deposits an interest payment, if the account qualifies
    * @param interest_rate - the rate of interest, e.g., 0.05 for 5%
    * @return whether the account qualified for an interest payment.  */
  public boolean computeInterest(double interest_rate)
  { boolean outcome = false;
    if ( eligible_for_interest )
       { int interest = (int)(balanceOf() * interest_rate);
         deposit(interest);
         outcome = true;
       }
    eligible_for_interest = (balanceOf() > minimum_balance);  // reset
    return outcome;
  }
}
```

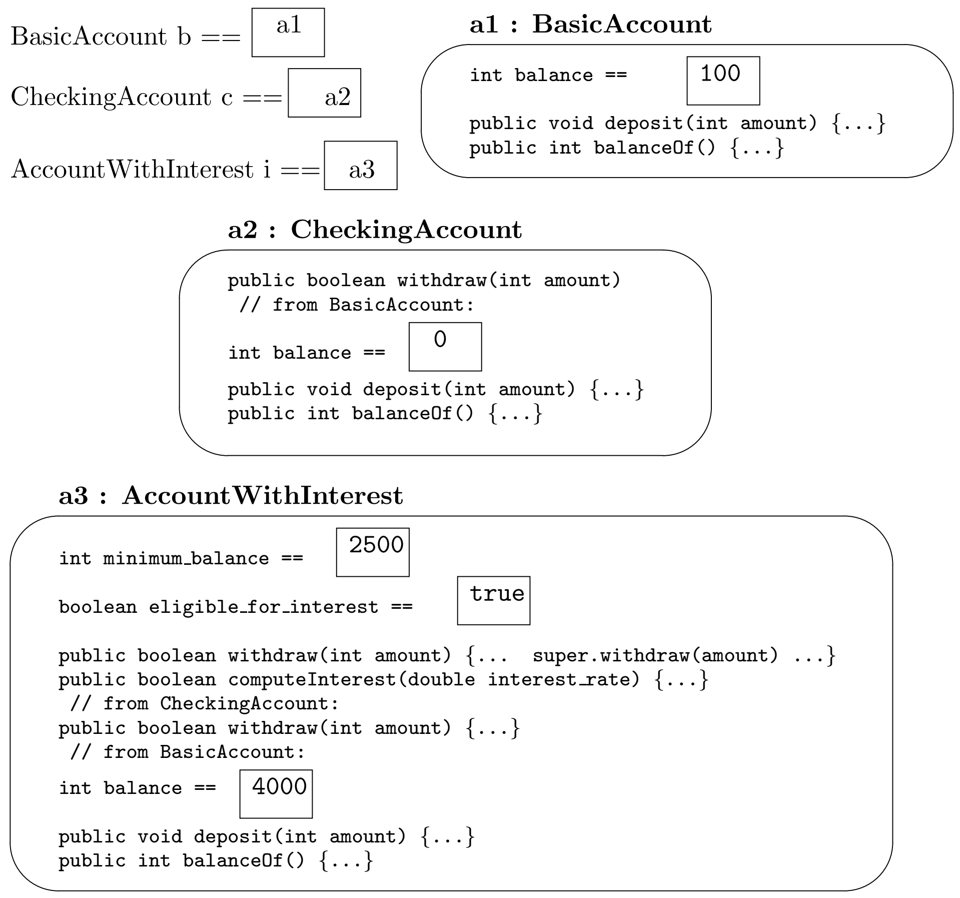
in the superclass—here, method `withdraw` in `CheckingAccount`—to be used. In this way, the new `withdraw` method exploits the behavior of the method it overrides.

The override of `withdraw` in Figure 22 is acceptable programming style, because it maintains the same responsibility of the `withdraw` method in Figure 21—to withdraw money—but does so in a more intelligent way.

The classes in Figures 21 and 22 let us create a variety of different bank accounts simultaneously, e.g.,

```
BasicAccount b = new BasicAccount(100);
CheckingAccount c = new BasicAccount(0);
AccountWithInterest i = new AccountWithInterest(2500, 4000);
```

creates these objects:


BasicAccount b == a1
CheckingAccount c == a2
AccountWithInterest i == a3
a1 : BasicAccount
int balance == 100
public void deposit(int amount) {...}
public int balanceOf() {...}
a2 : CheckingAccount
public boolean withdraw(int amount)
// from BasicAccount:
int balance == 0
public void deposit(int amount) {...}
public int balanceOf() {...}
a3 : AccountWithInterest
int minimum_balance == 2500
boolean eligible_for_interest == true
public boolean withdraw(int amount) {... super.withdraw(amount) ...}
public boolean computeInterest(double interest_rate) {...}
// from CheckingAccount:
public boolean withdraw(int amount) {...}
// from BasicAccount:
int balance == 4000
public void deposit(int amount) {...}
public int balanceOf() {...}

Each of its three objects has its own combination of fields and methods and behaves uniquely to specific messages. At this point, if we send the message, `i.withdraw(500)`, the object at `a3` uses its "newest" `withdraw` method, the one from `class AccountWithInterest`, to make the withdrawal. This method itself invokes the older `withdraw` method to complete the transaction. (The keyword, `super`, forces the older method to be used.)

9.12.2 Semantics of Overriding

The semantics of method override possesses a surprising feature: The method that the Java compiler selects as the receiver of an invocation might be different from the method that is selected when the invocation is executed!

You can avoid reading this section if you promise *never to override a superclass's method that is invoked by other methods in the superclass.* That is, *don't do this:*

```
public class C1
{ ...
  public ... f(...)
  { ... }

  public ... g(...)
  { ... f(...) ... }
}

public class C2 extends C1
{ ...
  public ... f(...)
  { ... }
}
```

This form of example causes surprising behavior when `g` is invoked.

To understand why, we start by reviewing the main points from the section, "Formal Description of Methods," at the end of Chapter 5. There, we learned that the Java compiler checks the data typing of every method invocation by selecting the method that is the target of the invocation. For simplicity, we consider normal (non-static) invocations. As described in Chapter 5, data-type checking is a four-step process:

Given an invocation,

```
[[ RECEIVER . ]]?  NAME0 ( EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)
```

the compiler

1. *determines the data type of* `RECEIVER`*, which will be a class name or an interface name,* `C0`*.* (Note: Since Chapter 5, we have learned that `RECEIVER` can be the keyword, `super`. In this case, the data type is the superclass of the class where the method invocation appears. There is one more keyword, `this`, which can appear as a `RECEIVER`. The data type of `this` is the class where the method invocation appears.)

2. *selects the best matching method for* `NAME0`*.* This will be a method of the form

   ```
   VISIBILITY TYPE0 NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
   ```

that is found within class or interface `C0` (or, if not in `C0`, then in `C0`'s superclass, `C1`, or then in `C1`'s superclass, `C2`, and so on—see Chapter 5 for the algorithm).

3. *attaches the header-line information to the invocation.* The method invocation now looks like this:

   ```
   [[ RECEIVER . ]]?  NAME0 (EXPRESSION1: TYPE1, EXPRESSION2: TYPE2,
                 ..., EXPRESSIONn: TYPEn) SUFFIX;
   ```

   where `SUFFIX` is one of

   - `public`: the selected method is a public method
   - `private Ck`: the selected method is a private method that resides in class `Ck`
   - `super Ck`: the `RECEIVER` is `super`, and the selected method is a public method that resides in class `Ck`

4. *returns* `TYPE0` *as the result type of the invocation.*

When a method invocation is executed, the Java Virtual Machine uses the typing information attached by the Java compiler attached to locate the invoked method. As stated in Chapter 5, given the annotated invocation,

```
[[  RECEIVER .  ]]?  NAME0 ( EXPRESSION1 : TYPE1, EXPRESSION2 : TYPE2,
               ..., EXPRESSIONn : TYPEn ) SUFFIX
```

the execution follows these five steps:

1. `RECEIVER` *computes to an address of an object,* `a`. (Note: If `RECEIVER` is omitted, or is `super` or `this`, use the address of the object in which this invocation is executing.)

2. *Within the object at address* `a`*, select the method* `NAME0`. This step depends on the value of the `SUFFIX`:

   - If it is `private Ck`, select the private method that came from `class Ck` whose header line has this form:

     ```
     private ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
     ```

   - If it is `super Ck`, select the public method that came from `class Ck` whose header line has this form:

     ```
     public ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
     ```

   - If it is `public`, look at the data type attached to object `a`; say that it is `C0`. Search the public methods starting with the ones that came from `class C0`, for a method whose header line has the form,

```
public ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
```

If the method is found in `class C0`, select it. Otherwise, search the public methods that came from `C0`'s superclass; repeat the search until the method is found.

3. *Evaluate the actual parameters.*

4. *Bind the actual parameters to the formal parameters.*

5. *Execute the body of the selected method.*

Step 2 is of interest to us when an invocation's `SUFFIX` is marked `public`. In this case, the Java interpreter makes a search for a method named `NAME0`, and the method that is selected might be different from the one that the Java compiler selected. We see this in the example that follows.

Figure 23 defines two classes, `C1` and `C2`, that hold integers and report their values. Since the second class, `C2`, augments the integer held in its superclass, `C1`, with one of its own, it overrides the `stringVal` function so that both integers are included in the answer returned by the function. This raises an interesting question: Which version of `stringVal` will be invoked within `print`? The answer depends on the object created. For this case,

```
C1 c1 = new C1(3);
c1.print();
```

it is easy to guess correctly that `value is C1:  3` is printed; `print` invokes the `stringVal` method in `C1`. Indeed, when the Java compiler type checked `class C1`, it came to the conclusion that `print` invokes `stringVal` in `C1`.

But if we write,

```
C2 c2 = new C2(4,5);
c2.print();
```

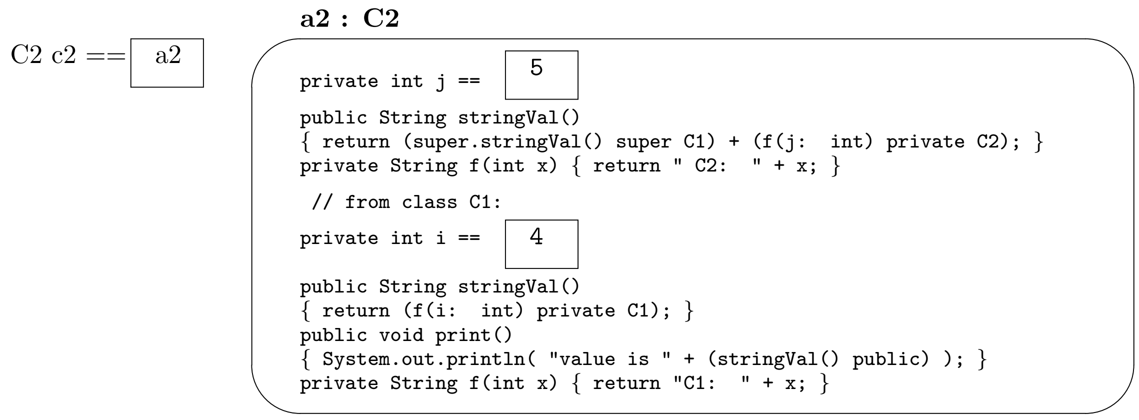
this code's execution produces `value is C1:  4 C2:  5`. This occurs because `print` is executed within a `C2` object, and therefore uses the object's "newest" version of `stringVal`, which is the `C2` version. This is not what was selected by the Java compiler, but the execution works properly because *the data typing of the newer version of* `stringVal` *is identical to the data typing of the older version.* This means the Java compiler's type checking efforts are not harmed.

We can see the rationale behind this execution result by drawing the object created by the above declaration. The drawing displays the annotations that the Java

Figure 9.23: example of method override

```
public class C1
{ private int i;

  public C1(int x)
  { i = x; }

  public String stringVal()
  { return  f(i); }

  public void print()
  { System.out.println( "value is " + stringVal() ); }

  private String f(int x)
  { return "C1: " + x; }
}

public class C2 extends C1
{ private int j;

  public C2(int x, int y)
  { super(x);
    j = y;
  }

  public String stringVal()
  { return super.stringVal() + f(j) ; }

  private String f(int x)
  { return " C2: " + x; }
}
```

compiler attached to each method invocation:



```
private int j == 5

public String stringVal()
{ return (super.stringVal() super C1) + (f(j:  int) private C2); }
private String f(int x) { return " C2:  " + x; }

 // from class C1:
private int i == 4

public String stringVal()
{ return (f(i:  int) private C1); }
public void print()
{ System.out.println( "value is " + (stringVal() public) ); }
private String f(int x) { return "C1:  " + x; }
```

The annotations, `private Ci` and `super Ci`, select specific methods for execution; there is no re-search. But a method invocation that is labelled `public` requires a search of all the public methods to find the newest version of the invoked method name.

For the example, the compiler labels the starting invocation as `c2.print() public`. First, `c2` computes to address `a2`, and the public methods of `a2` are searched for the newest version of `print`. The only version, the one from `class C1`, is located, so the statement,

```
System.out.println( "value is " + (stringVal() public) );
```

executes. This sends the message, `stringVal() public`, which is directed again to `a2`, because there is no `RECEIVER` on the method name.

Another search proceeds, this time for the newest version of `stringVal`. This time, *the version from* `class C2` is selected. (This is a different result than the Java compiler's search, and certainly the author of `class C1` might be a bit surprised as well!)

The selected method executes this statement:

```
return  (super.stringVal() super C1) + (f(j: int) private C2);
```

The first message, `super.stringVal() super C1`, is directed again to `a2`, which is forced to use the `stringVal` method from `class C1`, because of the `SUFFIX`, `super C1`. The body of `C1`'s `stringVal` states,

```
return  (f(i: int) private C1);
```

which forces the private method named `f` from `class C1` to execute. Therefore, the string, `"C1:  4"`, is returned as the answer.

Similarly, the invocation, `f(j:  int) private C2` forces `class C2`'s `f` method to execute, producing `"C2:  5"`. These strings are combined and produce the printed answer, `value is C1:  4 C2:  5`.

In summary, the Java compiler selects precisely and correctly the private and "super" methods used by invocations. But it is misled about invocations of public methods—the method the compiler selects might be overridden by a newer version (fortunately, with the same data typing!) when objects are created at execution. Whether this is a positive feature of the Java language is subject to debate, but in any case, you must be well aware of these execution behaviors when you write overriding methods.

9.12.3 `final` components

You can label a class or any of its components as *final*, if you wish to prohibit modifications to it.

A *final class* is labelled,

```
public final class C
{  ... }
```

Now, the Java compiler will not allow any other class to use the class as a superclass, that is, `public class D extends C` is disallowed. One consequence is that no method in `C` can be overridden.

When a method is labelled `final`, then it cannot be overridden. For example, if the override of `C1`'s `stringVal` method in Figure 23 is upsetting, we can modify `C1` to read

```
public class C1
{ private int i;

  public final String stringVal()
  { return  f(i); }

  public void print()
  { System.out.println( "value is " + stringVal() ); }

  private String f(int x)
  { return "C1: " + x; }
}
```

This makes it impossible to override `stringVal` in `class C2` (and likely forces `C2` to override the `print` method, which is a more benign override).

A third use of `final` is to restrict a variable so that once it is initialized, it can not be altered. A simple example is,

```
private final int MAX_SIZE = 50;
```

which permanently fixes the value of `MAX_SIZE` to 50.

A final variable can be initialized from within a constructor:

```
public class FrozenBankAccount
{ private final int balance;

  public FrozenBankAccount(int final_balance)
  { balance = final_balance; }

  ...  // here, assignments to balance are disallowed
}
```

9.12.4 Method Overloading

In Chapter 6, Figure 15, we saw two methods in `class BankWriter` with the same name:

```
/** BankWriter writes bank transactions */
public class BankWriter extends JPanel
{ ...

  /** showTransaction displays the result of a monetary bank transaction
    * @param message - the transaction
    * @param amount - the amount of the transaction */
  public void showTransaction(String message, int amount)
  { last_transaction = message + " " + unconvert(amount);
    repaint();
  }

  /** showTransaction displays the result of a bank transation
    * @param message - the transaction */
  public void showTransaction(String message)
  { last_transaction = message;
    repaint();
  }
}
```

The two methods differ in the number of parameters they require; the method name, `showTransaction`, is *overloaded.*

A method name is said to be overloaded when there are two methods that are labelled with the same name but have different quantities or data types of formal parameters; the two same-named methods appear in the same class (or one appears in a class and another in a superclass).

The overloading technique is promoted as a "memory aid" for a programmer: If there is a collection of methods that more-or-less act the "same way" but differ in the parameters they require to "act," then the methods can be overloaded. The above example is typical—both variants of `showTransaction` display a transaction result, but the first handles transactions that include text and numerical values, and the second handles text-only transactions.

Overloading often appears in a class's constructor methods:

```
public class BankAccount
{ private int balance;

  public BankAccount()
  { balance = 0; }

  public BankAccount(int initial_balance)
  { balance = initial_balance; }

  ...
}
```

Both constructor methods are initializing the bank account, but they differ in the parameters they use to do so. Now, a bank-account object can be created two ways—by `new BankAccount()` or by `new BankAccount(1000)`, say.

Another form of overloading is based on parameter data type. The following methods overload the name, `displayValue`, by distinguishing between the types of the methods' parameter:

```
public void displayValue(int i)
{ System.out.println(i); }

public void displayValue(String s)
{ System.out.println(s); }

public void displayValue(BankAccount b)
{ System.out.println(b.balanceOf()); }
```

It is easy to select the proper method based on the actual parameter supplied with the method invocation, e.g., `displayValue("Hello")` or `displayValue(new BankAccount(300))`. But some invocations do not match any of the methods, e.g., `displayValue(2.5)`—the Java compiler will report an error in this situation.

Perhaps you have deduced that `System.out`'s `println` is itself overloaded. Indeed, there are ten different methods! These include

```
public void println() {...}
public void println(boolean x) {...}
public void println(char x) {...}
public void println(int x) {...}
public void println(long x) {...}
public void println(double x) {...}
```

and so on. (Consult the Java API for details.)

In the case of `println`, we note that some of the data types used in the overloaded method are related by subtyping. For example, `int` is a subtype of `long` and of `double`. This suggests that any of the three `println` methods that can print integers, longs, or doubles, can be used to execute `System.out.println(3)`. But the method, `public void println(int x) {...}` is selected because it is the best matching method for the invocation.

But this approach gets quickly confusing. Say that the name, `f`, is overloaded by these two methods:

```
public boolean f(int x)
{ System.out.println("nonfractional");
  return true;
}

public boolean f(double x)
{ System.out.println("fractional");
  return false;
}
```

The behavior of `f(3)` and `f(3.5)` are not too difficult to guess correctly. What about

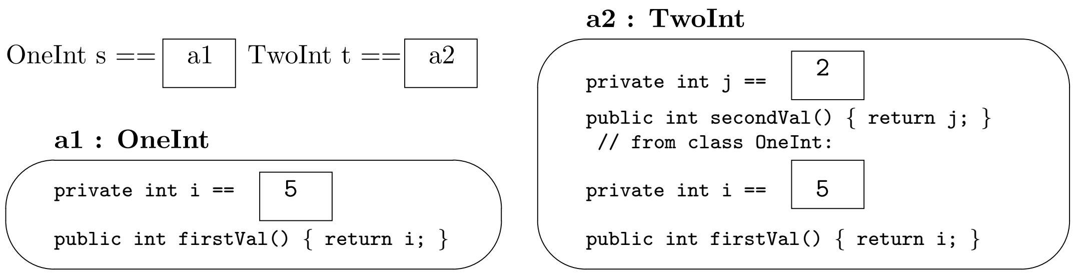
```
long x = 3000000000;
... f(x) ...
```

That is, when the actual parameter is a long integer? There is no variant of `f` for data type `long`, but since `long` is a subtype of `double` (and not of `int`), the parameter is cast into a double, and the second variant is chosen; `"fractional"` is printed and `false` is returned.

Figure 24 presents a more perplexing example. In the example, if we declare

```
OneInt s = new OneInt(5);
TwoInt t = new TwoInt(5, 2);
```

we get these two objects in storage:



Say that, in a third class, we write these two methods:

Figure 9.24: one class that extends another

```
/** OneInt is a ``wrapper'' for a single integer */
public class OneInt
{ private int i;

  public OneInt(int x) { i = x; }

  public int firstVal() { return i; }
}


/** TwoInt is a ``wrapper'' for a pair of integers. */
public class TwoInt extends OneInt
{ private int j;

  public TwoInt(int x, int y)
  { super(x);  // execute the constructor for  OneInt,  the superclass
    j = y;
  }

  public int secondVal() { return j; }
}
```

```
public boolean equals(OneInt x, OneInt y)
{ return (x.firstVal() == y.firstVal()); }

public boolean equals(TwoInt x, TwoInt y)
{ return (x.firstVal() == y.firstVal())  &&  (x.secondVal() == y.secondVal()) }
```

These seem like reasonable definitions of equality checks for the two forms of objects. But consider this statement, which uses objects `s` and `t`:

```
System.out.println(equals(s, t));
```

This statement prints `true`! There is no `equals` method when one actual parameter has type `OneInt` and the other has type `TwoInt`, but since `TwoInt` is a subtype of `OneInt`, actual parameter `t` is cast to `OneInt`, and the first method is selected.

If you are uncomfortable with the above answer, you can write these two additional methods:

```
public boolean equals(OneInt x, TwoInt y)
{ return false; }

public boolean equals(TwoInt x, OneInt y)
{ return false; }
```

Now, all combinations of data types are listed, and `equals(s, t)` returns `false`. But this example,

```
OneInt u = t;
System.out.println(equals(u, t));
```

returns `false` as well, even though variables `u` and `t` both hold `a2` as their value!

The precise reasons why these behaviors unfold as they do are explained in the next section. But the moral is, if at all possible, *avoid overloading a method name by methods that are distinguished by a formal parameter whose respective data types are related by subtyping.* That is,

- it is acceptable to overload a method name based on quantity of parameters (cf., the `showTransaction` example earlier)
- it is acceptable to overload a method name based on the data type of a parameter, if the data types are unrelated by subtyping (cf. the `displayValue` example)

but it is questionable to overload based on parameter data types related by subtyping (cf. the `equals` example).

In many cases, what appears to be an "essential" use of overloading based on parameter data type can be eliminated by augmenting the classes involved with an abstract class. For example, we can retain all four of the above variants of `equals` if we change, in a trivial way, the relationship between classes `OneInt` and `TwoInt` with an abstract class:

```
public abstract class AbsInt
{ private int i;

  public AbsInt(int x) { i = x; }

  public int firstVal() { return i; }
}

public class OneInt extends AbsInt
{ public OneInt(int x)
  { super(x); }
}

public class TwoInt extends AbsInt
{ private int j;

  public TwoInt(int x, int y)
  { super(x);
    j = y;
  }

  public int secondVal() { return j; }
}
```

Since it is illegal to create `AbsInt` objects, `class OneInt` is essential. And, no longer is there a subtyping relationship between `OneInt` and `TwoInt`, which was at the root of our earlier difficulties.

9.12.5 Semantics of Overloading

The Java compiler must type check overloaded method names, like the ones just seen. First, you should review the section, "Formal Description of Methods," in Chapter 5, and study closely the definition of "best matching method." This definition must be revised to handle the more difficult examples in the previous section.

You can avoid reading this section if if you promise *never to overload a method name based on parameter data types related by subtyping.*

Revised Definition of Best Matching Method

Say that the Java compiler must locate the best matching method for this invocation:

```
[[  RECEIVER NAME0 .  ]]? (EXPRESSION1, EXPRESSION2, ..., EXPRESSIONn)
```

Say that the compiler has calculated that `C0` is the data type of the `RECEIVER` and say that each `EXPRESSIONi` has data type `Ti`, for `i` in `1..n`.

What method will be invoked by this invocation? If the invocation appears within `class C0` also, then the *best matching method* is the method the compiler finds by `searching(all public and private methods of class C0)`; Otherwise, the best matching method is the method found by `searching(all public methods of class C0)`.

The algorithm for `searching(some of the methods of class C0)` is defined as follows:

Within `some of the methods of class C0`, find the method(s) whose header line has the form,

```
... ... NAME0(TYPE1 NAME1, TYPE2 NAME2, ..., TYPEn NAMEn)
```

such that each `Ti` is a subtype of `TYPEi`, for all `i` in the range of `1..n`.

- If exactly one such method definition of `NAME0` in `C0` is found, then this method is selected.

- If there are two or more variants of `NAME0` in `C0` that fit the criterion stated above (that is, `NAME0` is overloaded), then the variant whose formal parameter types *most closely match* the types of the actual parameters is selected. If none of the variants most closely match, then the invocation is said to be *ambiguous* and is not well typed; the search fails. (The definition of "most closely match" is given below.)
- If there is no method definition for `NAME0` that is found, and if `class C0 extends C1`, then the best matching method comes from `searching(all public methods of class C1)`. But if `class C0` extends no other class, there is no best matching method, and the search fails.

Here is the definition of "most closely match": Consider a single actual parameter, `E`, with data type, `T` and the data types, `T1`, `T2`..., `Tn`, such that `T` is a subtype of each of `T1`, `T1`, ... `Tn`. We say that one of the data types, `Tk`, *most closely matches* `T` if `Tk` is itself a subtype of all of `T1`, `T2`, ..., `Tn`. (`Tk` is considered to be a subtype of itself by default.)

Next, take this definition and apply it to all `n` of the actual parameters of a method invocation: For a variant of `NAME0` to mostly closely match a method invocation, the data type of each of its formal parameters must most closely match the type of the corresponding actual parameter of the method invocation.

If we return to the example in Figure 18 and reconsider this situation:

```
public boolean equals(OneInt x, OneInt y)
{ return (x.firstVal() == y.firstVal()); }

public boolean equals(TwoInt x, TwoInt y)
{ return (x.firstVal() == y.firstVal())  &&  (x.secondVal() == y.secondVal()) }

public boolean equals(OneInt x, TwoInt y)
{ return false; }

OneInt s = new OneInt(5);
TwoInt t = new TwoInt(5, 2);
System.out.println(equals(s, t));
```

we see that first and third methods named `equal` match the invocation, `equals(s, t)`, because the data types of the actual parameters are `OneInt` and `TwoInt`, respectively. When we compare the data types of the formal parameters of the two methods, we find that the third variant most closely matches.

Finally, we must note that the Java compiler in fact implements a more restrictive version of best matching method, prohibiting some examples of overloading that extend across super- and subclasses. For example, the Java compiler will refuse to calculate a best matching method for this example:

```
public class A
{ public A() { }

  public void f(int i) { System.out.println("A"); }
}

public class B extends A
{ public B() { }

  public void f(double i) { System.out.println("B"); }
}
   ...
B ob = new B();
ob.f(3);
```

because the above definition of "best matching method" would select the version of `f` in `B` as the best match to `ob.f(3)`, even though there is a more appropriate version in the superclass, `A`.