

Politehnica University of Bucharest
Faculty of Engineering in Foreign Languages
English Stream

Object Oriented Programming

Luca Dan Serbanati

2005

Introduction to OOP

Traditional structured programming consists of designing a set of functions (or *algorithms*) to solve a problem. After the functions were determined, the traditional next step was to find appropriate ways to store the data.

OOP reverses the order and puts data first, then looks at the algorithms that operate on the data. The key to being most productive in OOP is to make each object responsible for carrying out a set of related tasks. If an object relies on a task that isn't its responsibility, it needs to have access to another object whose responsibilities include that task. The first object then asks the second object to carry out the task. This is done with a more generalized version of the function call that you are familiar with in procedural programming. (Recall that in Java these function calls are usually called *method calls*.)

In particular, an object should never directly manipulate the internal data of another object, nor should it expose data for other objects to access directly. All communication should be via method calls. This approach is called *data hiding*. It facilitates the overall control of a complex program by turning aside attention from the implementation details.

By *encapsulating* object data, you maximize reusability, reduce data dependency and minimize debugging time. *Encapsulation* is a key concept in working with objects. Formally, encapsulation is nothing more than combining data and behaviour in one package and hiding the implementation of the data from the user of the object.

The class provides the main mechanism of encapsulation. A class is the template or blueprint from which objects are actually made. When you *construct* an object from a class, you are said to have created an *instance* of the class.

The data in an object are called its *instance fields*, and the functions and procedures that operate on the data are called its *methods*. A specific object that is an instance of a class will have specific values for its instance fields. The set of those values is the current *state* of the object. Whenever you apply a method to an object, its state may change.

It cannot be stressed enough that the key to making encapsulation work is to have methods *never* directly access instance fields in a class other than their own. Programs should interact with object data *only* through the object's methods. Encapsulation is the way to give the object its "black box" behavior, which is the key to reuse and reliability.

Objects

To work with OOP, you should be able to identify three key characteristics of objects. The three key characteristics are:

- ❑ The object's *behaviour* - what can you do with this object, or what methods can you apply to it?
- ❑ The object's *state* - how does the object react when you apply those methods?
- ❑ The object's *identity* - how is the object distinguished from others that may have the same behaviour and state?

All objects that are instances of the same class share a family resemblance by supporting the same *behaviour*. The behaviour of an object is defined by the methods that you can call.

Next, each object stores information about what it currently looks like. This is the object's *state*. An object's state may change over time, but not spontaneously. A change in the state of an object must be a consequence of method calls. (If the object state changed without a method call on that object, someone broke encapsulation.)

However, the state of an object does not completely describe it, since each object has a distinct *identity*. For example, in an order-processing system, two orders are distinct even if they request

identical items. Notice that the individual objects that are instances of a class *always* differ in their identity and *usually* differ in their state.

These key characteristics can influence each other. For example, the state of an object can influence its behaviour. (If an order is "shipped" or "paid," it may reject a method call that asks it to add or remove items. Conversely, if an order is "empty," that is, no items have yet been ordered, it should not allow itself to be shipped.)

In a traditional procedure-oriented program, you start the process at the top, with the **main** function. When designing an object-oriented system, there is no "top," and newcomers to OOP often wonder where to begin. The answer is: You first find classes and then you add methods to each class.

But how identify classes?

A simple rule of thumb in identifying classes is to look for nouns in the problem analysis. Methods, on the other hand, correspond to verbs.

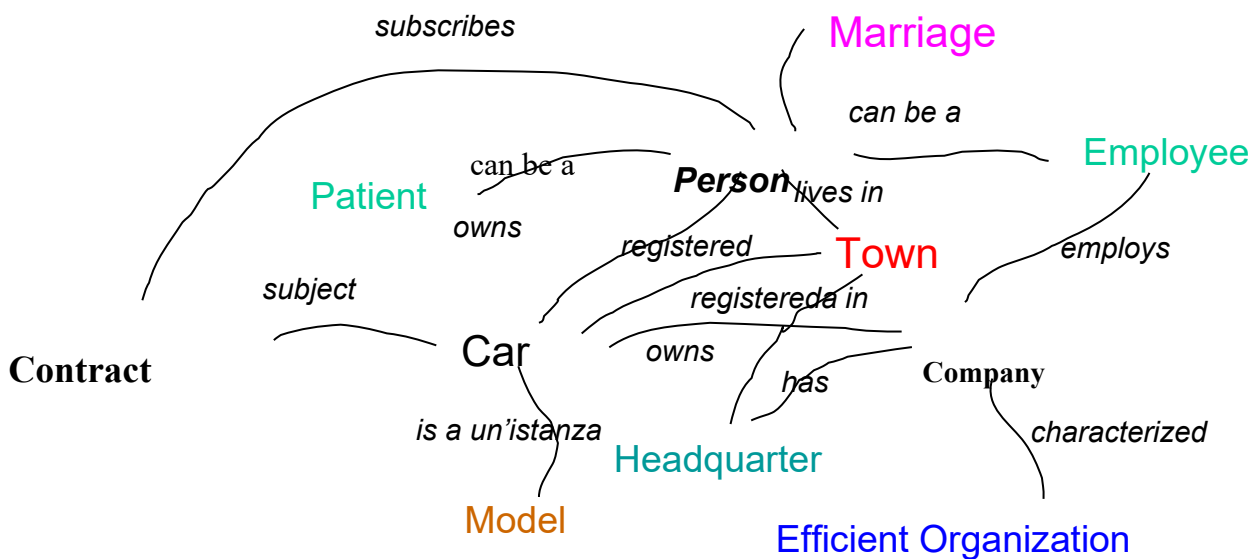
For example, in an order-processing system, some of these nouns are:

- ❑ Item
- ❑ Order
- ❑ Shipping address
- ❑ Payment
- ❑ Account

Concepts

Concepts and Reality

Humans have the ability to form concepts, can label their concepts with symbols and can



manipulate these symbols to define and communicate meaning.

A **concept** is an idea or notion that we apply to the things, or objects, in our awareness.

1. For us, concepts are a recognition device.

The concepts may be:

- Tangible (person, car, table)
- Intangible (time, quality, company)
- Roles (doctor, patient, owner)
- Judgments (high pay, good example, productive job)
- Relational (marriage, partnership, ownership)
- Events (sale, purchase, market crash)

- Other types of concepts (string, number, icon, image, signal, process)
2. For us concepts are also a test for reality: we can apply concepts to objects in our reality and discard concepts that no longer apply. Each concept is based on tests that determine when it applies.

Documenting Concepts

We use concepts as units of knowledge in which each has an intension and an extension. There are two sides of the same coin.

Intension (definition, idea, understanding): the complete definition of the concept and the test that determines whether or not the concept applies to an object.

Extension: the set of all objects to which the concept applies.

There are concepts with no name (a person might define a concept without symbolizing it with a name).

There are concepts with no instances. A new company might not yet have any instance of Order or Customer.

There are concepts with no definition. (A code as P-CD-X99 has several instances: "X17", "Z42", ..., but non definition).

Some concepts have synonyms (Customer, Client)

Domains

A **domain** is a collection of objects in a selected area of interest.

A **domain specification** is the collection of concepts that apply to a domain.

The domain of a system is the real world that the analyst is analyzing: the domain specification is a product of systems analysis.

Objects

An **object** is anything to which a concept applies. It is an instance of a concept.

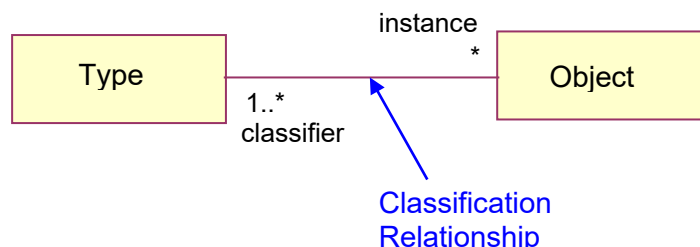
Any thing that we can think of, refer to, describe, discuss, or experience is an object as long as we have the concepts to do so.

The concepts can be concepts. Compaq Laptop is a concept that applies to many thousands of serial-numbered instances produced by Compaq. Compaq Laptop is also an object between other instances of the concept Product Type. The concept Product Type is an object itself, an instance of the concept called Concept.

Classification

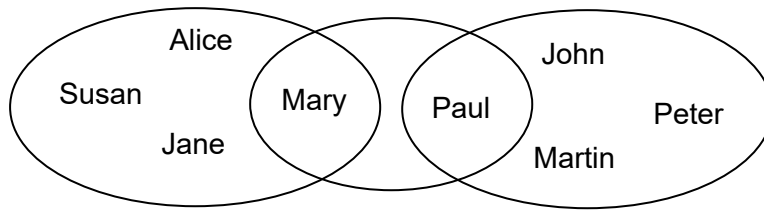
Classification is the act or result of applying a concept (i.e. type) to an object.

Classification is one mechanism we use to manage the complexity of the objects in the world: it allows us to treat many individual phenomena (objects) with a single notion (type).



Object Lifecycles

Most objects have periods of existence. For these objects the beginning and ending points can be reasonably defined. Objects appear in our awareness when we first apply a concept to them, and objects disappear when concepts no longer apply.



Woman Employee Man

Real objects undergoes changes. Sometimes for such a change an other concept may apply: car, squashed cube in a junkyard, sliced cube etc. In this way, objects can seem to have their own *lifecycles*. To describe these changes, we can employ sets.

Set = A particular collection of objects.

Set membership is determined solely by its concept's complete definition or intension. The definition acts as a filter that screens all objects, permitting only those objects into the set that pass the membership test. So, *set* is another name for the extension of a concept.

A set can include many objects and each object can be a member of many sets.

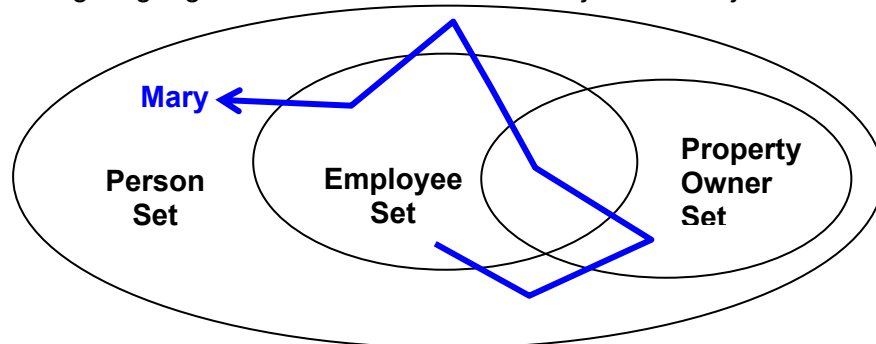
Classification. When we determine that a concept applies to a specific object, the object is classified as a member of a specific set. An object can be declassified when it is removed from a particular set

Specifying the method of these classification changes is a technique at the very heart of OO process specification.

In its lifetime an object may be a member of several sets and, on many occasions, change its set membership. In other words:

- **Multiple classification:** an object can have multiple concepts that apply to it at any one moment.
- **Dynamic classification:** the collection of concepts that applies to an object can change over time.

Most OO programming languages are restrictive with their objects: an object can be instance of

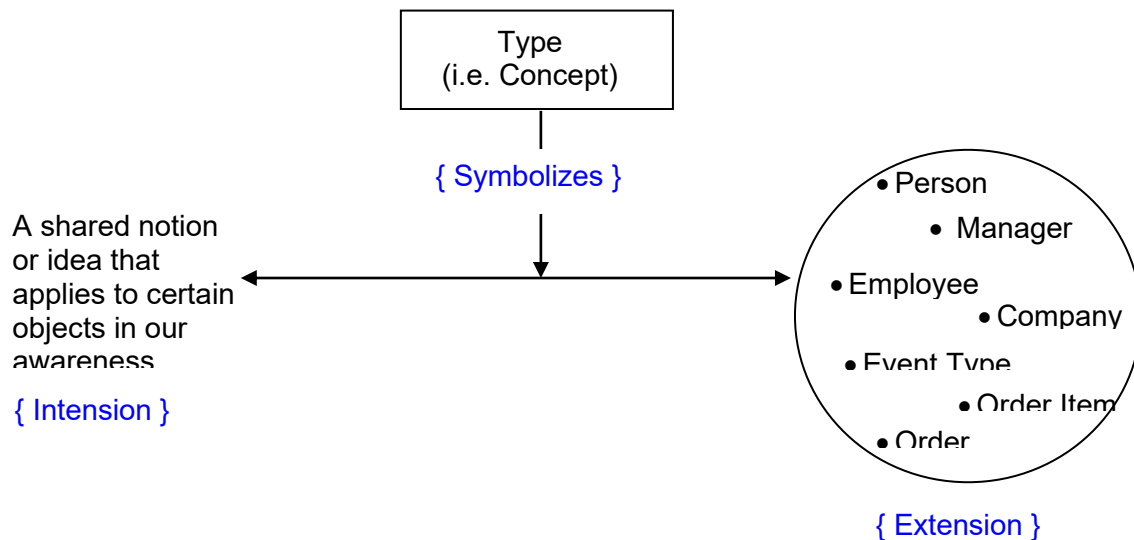


only one OOPL class for life. To change its class, a new object must be created to replace the old one. However, in OO analysis, we are not modeling how computer languages and databases work. We are analyzing the enterprise world as people understand it.

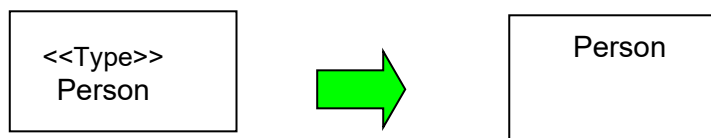
Concept Versus Type

Type = Concept or kind of object. It is a recommended term for concept in the OO analysis standard community. It seems to be more intuitive as the concept term.

Because Type is synonymous with Concept, it too has an intension and an extension. Its intension is the definition of concept. Its extension is the set of all concepts.



Types are classes used in analysis and Implementation classes are classes used for implementation purposes.

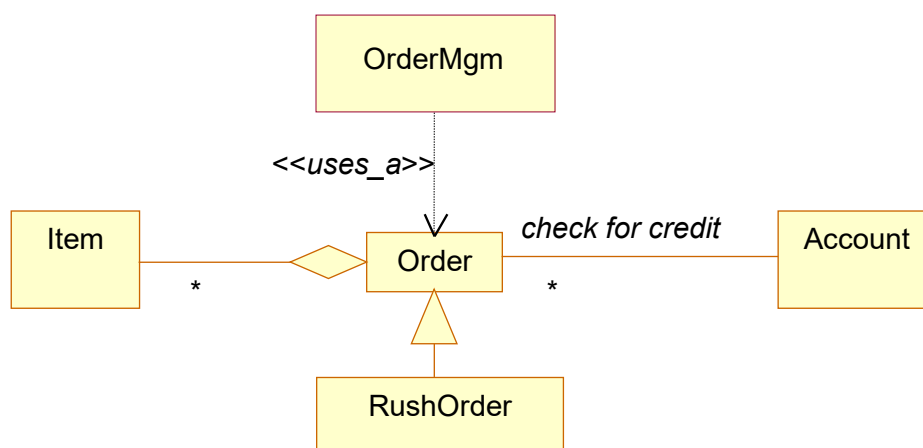


In OOP concepts and types are expressed as classes.

Relationships Between Classes

The most common relationships between classes are:

- ❑ *Dependence* ("uses-a")
- ❑ *Association* ("knows-a")
- ❑ *Aggregation* ("has-a")
- ❑ *Generalisation* ("is-a")



Many programmers use the UML (Unified Modeling Language) notation to draw *class diagrams* that describe the relationships between classes. You draw classes as rectangles, and relationships as arrows with various adornments.

Dependence

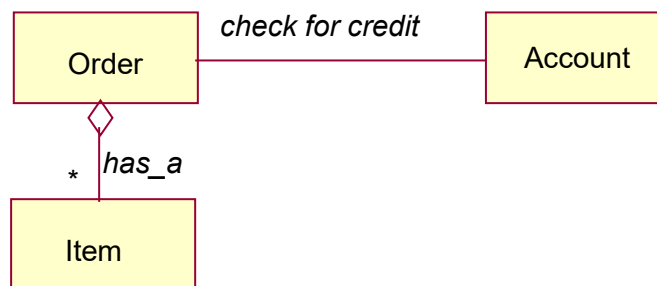
The *dependence* or "uses-a" relationship is the most obvious and also the most general. For example, the Order class uses the Account class, since Order objects need to access Account objects to check for credit status. But the Item class does not depend on the Account class, since Item objects never need to worry about customer accounts. Thus, a class depends on another class if its methods manipulate objects of that class.

Association

The *association* between two classes is when objects of a class "knows" objects of the other class, that is can call them when is needed.

Association = A user-defined relationship.

For example, the Order class knows the Account class, since Order objects need to access Account objects to check for credit status. But the Item class does not depend on the Account class, since Item objects never need to worry about customer accounts. Thus, a class is in association with another class if its objects contain information for identification of objects of that class.



Aggregation

The *aggregation* or "has-a" relationship is easy to understand because it is concrete; for example, an Order object contains Item objects. Containment means that objects of class A (physically or logically) contain objects of class B.

Aggregation is a mechanism for forming a whole from component parts. For example, each Hammer consists of its Head and Handle.

Other aggregations can be more conceptual and subjective in nature, such as a Jury is a collection of Juror members or a Marriage is composed of a Husband and a Wife.

Aggregation is the act or result of forming an object *whole* using other objects as its *parts*.

The component parts might, in addition, have their own component parts. In this way, aggregation can define hierarchies of part-whole configurations.

Notice that the aggregation is expressed in terms of types. This does not mean that these types are composed of other "component" types. It means that the *instances* of these types are composed of *instances* of other types. Granted, aggregation is based on objects consisting of other objects. Yet, defining aggregation solely in terms of objects is conceptually meaningless. To understand objects, we need to know the applicable concepts.

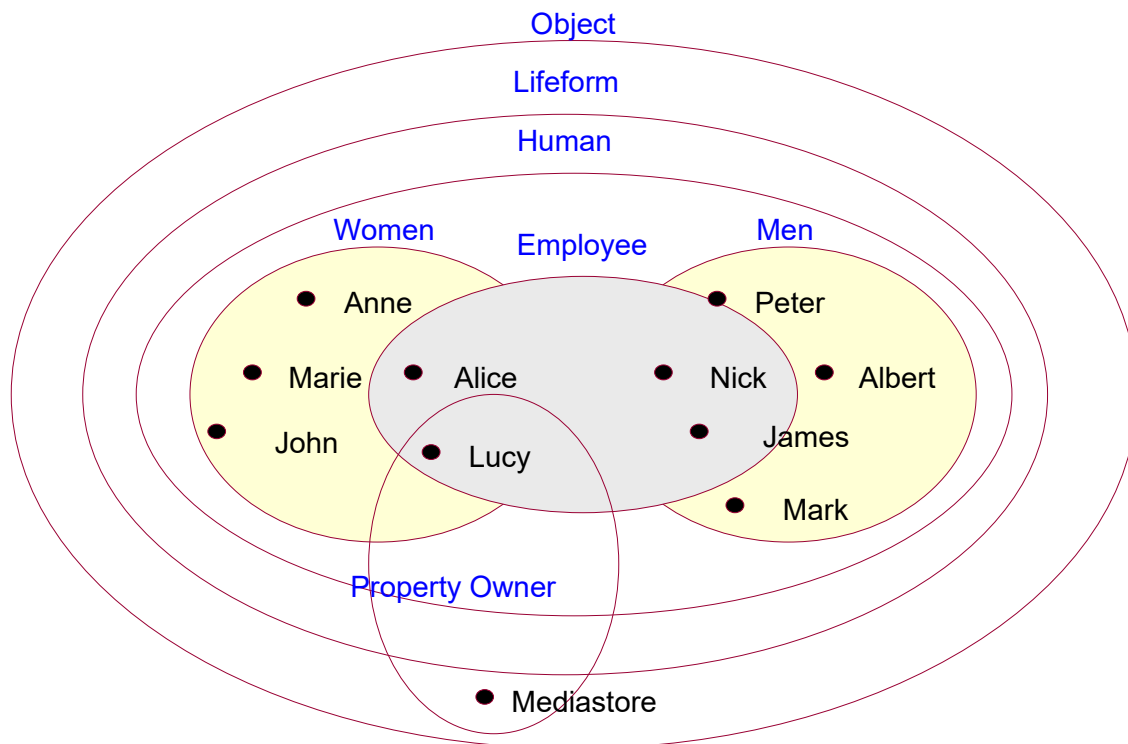
In other words, if an object has no applicable concepts, we have no way of understanding how to perceive and manipulate it.

Aggregation reduces complexity by treating a configuration of many things as one.

Generalisation

The *generalisation* or "is-a" relationship expresses a relationship between a more special and a more general class. For example, a RushOrder class inherits from an Order class. The specialized RushOrder class has special methods for priority handling and a different method for computing shipping charges, but its other methods, such as adding items and billing, are inherited from the Order class.

Generalisation is the act or result of distinguishing a concept (i.e. type) that completely includes or encompasses another.



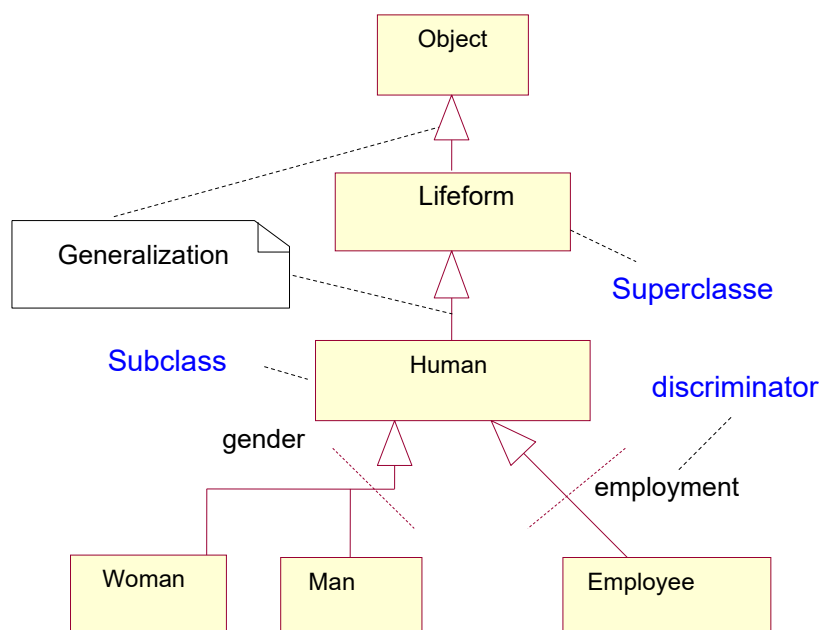
Generalisation enables us to say that all instances of a specific type are also instances of more general type – but not necessarily the other way around.

With generalisation we can define hierarchies of types, forming more and more general types. The more general type is called *supertype*.

The opposite of generalization is *specialization*. The specialized types are *subtypes*.

Generalisation in Java is expressed as *class extension*.

In general, if class A extends class B, class A inherits methods from class B but has more capabilities. (We will describe inheritance more fully in the next chapter, in which we discuss this important notion at some length.)



Classification and generalization/specialization are two different phenomena and, therefore, have different hierarchies.

Confusion between classification and generalization/specialization is common. One reason for this is that we manage complexity in both by abstracting, that is by removing distinctions and identifying commonalities. In one case, we remove distinctions between objects. In the other, we remove *types* of distinctions between *types* of objects. Therefore, classification is a relationship between type and object and generalization/specialization is a relationship between types only. A classification can remove the individual distinctions between objects Jane and Paul by classifying them as Human. In the figure above gender- and employment-related *types* of distinctions are removed between the types of objects called Man, Woman, and Employee by generalizing them as Human.

Generalization, then, enables us to define our types in even more general terms by using supertypes. The reverse, specialization, allows us to be more specific about our types by using subtypes.

Contrasting OOP with Traditional Procedural Programming Techniques

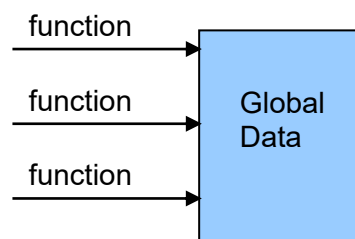
In procedure-oriented programming, you identify the tasks to be performed and then:

- ❑ By a stepwise refinement process, break the task to be performed into subtasks, and these into smaller subtasks, until the subtasks are simple enough to be implemented directly (this is the top-down approach).
- ❑ Write procedures to solve simple tasks and combine them into more sophisticated procedures, until you have the functionality you want (this is the bottom-up approach).

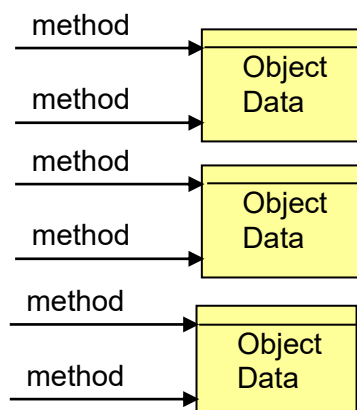
The important difference is that in OOP, you *first* isolate the classes in the project. Only then do you look for the methods of the class. And there is another important difference between traditional procedures and OOP methods: each method is associated with the class that is responsible for carrying out the operation.

For small problems, the breakdown into procedures works very well. But for larger problems, classes and methods have two advantages. Classes provide a convenient clustering mechanism for methods. A simple Web browser may require 2,000 functions for its implementation, or it may require 100 classes with an average of 20 methods per class. The latter structure is much easier to grasp by a programmer.

Procedure Orientation



Object Orientation



Objects and Object Variables

To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.

In the Java programming language, you use *constructors* to construct new instances. A constructor is a special method whose purpose is to construct and initialize objects. Let us look at an example. The standard Java library contains a `Date` class. Its objects describe points in time, such as "December 31, 1999, 23:59:59 GMT."

Constructors always have the same name as the class name. Thus, the constructor for the `Date` class is called `Date`. To construct a `Date` object, you combine the constructor with the `new` operator, as follows:

```
new Date()
```

This expression constructs a new object. The object is initialized to the current date and time. If you like, you can pass the object to a method:

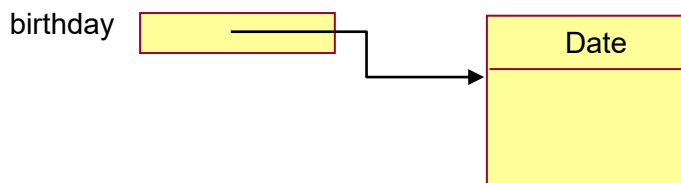
```
System.out.println(new Date());
```

Alternatively, you can apply a method to the object that you just constructed. One of the methods of the `Date` class is the `toString` method. That method yields a string representation of the date. Here is how you would apply the `toString` method to a newly constructed `Date` object.

```
String s = new Date().toString();
```

In these two examples, the constructed object is used only once. Usually, you will want to hang on to the objects that you construct so you can keep using them. Simply store the object in a variable:

```
Date birthday = new Date();
```



There is an important difference between objects and object variables. For example, the statement `Date deadline; // deadline doesn't refer to any object` defines an object variable, `deadline`, that can refer to objects of type `Date`. It is important to realize that the variable `deadline` *is not an object* and, in fact, does not yet even refer to an object. You cannot use any `Date` methods on this variable at this time. The statement `s = deadline.toString(); // not yet` would cause a compile-time error.

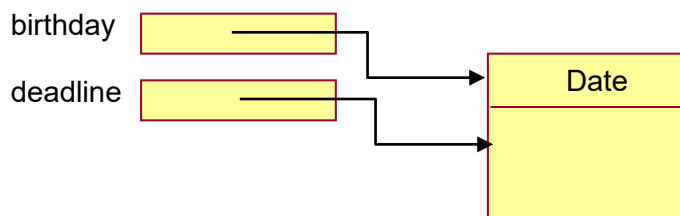
You must first initialize the `deadline` variable. You have two choices. Of course, you can initialize the variable with a newly constructed object:

```
deadline = new Date();
```

or you can set the variable to refer to an existing object:

```
deadline = birthday;
```

Now both variables refer to the *same* object.



It is important to realize that an object variable doesn't actually contain an object. It only *refers* to an object. In Java, the value of any object variable is a reference to an object that is stored elsewhere. The return value of the `new` operator is also a reference. A statement such as `Date deadline = new Date();`

has two parts. The expression `new Date()` makes an object of type `Date`, and its value is a reference to that newly created object. That reference is then stored in the `deadline` variable.

You can explicitly set an object variable to null to indicate that it currently refers to no object.

```
deadline = null;
```

```
. . .
```

```
if (deadline != null)
```

```
System.out.println(deadline);
```

If you apply a method to a variable that holds null, then a run-time error occurs.

```
birthday = null;
```

```
String s = birthday.toString(); // runtime error!
```

Local object variables are not automatically initialized to null. You must initialize them, either by calling new or by setting them to null.

The Date class has only a small number of methods that allow you to compare two points in time. For

example, the before and after methods tell you if one point in time comes before or after another.

```
if (today.before(birthday))
```

```
System.out.println("Still time to shop for a gift.");
```

The GregorianCalendar Class

An instance of the Date class has a state, namely *a particular point in time*.

Although you don't need to know this when you use the Date class, the time is represented by the number of milliseconds (positive or negative) from a fixed point, the so-called *epoch*, which is 00:00:00 UTC, January 1, 1970. UTC is the Coordinated Universal Time, the scientific time standard that is, for practical purposes, the same as the more familiar GMT or Greenwich Mean Time.

But a date description such as "December 31, 1999, 23:59:59" is an arbitrary convention, governed by a *calendar*. The Java library designers decided to separate the concerns of keeping time and attaching names to points in time. Therefore, the standard Java library contains two separate classes: the Date class which represents a point in time, and the GregorianCalendar class which expresses dates in the familiar calendar notation.

The expression

```
new GregorianCalendar()
```

constructs a new object that represents the date and time at which the object was constructed. You can construct a calendar object for midnight on a specific date by supplying year, month, and day:

```
new GregorianCalendar(1999, 11, 31)
```

Somewhat curiously, the months are counted from 0. Therefore, 11 is December. For greater clarity, there are constants like Calendar.DECEMBER.

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

You can also set the time:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Of course, you will usually want to store the constructed object in an object variable:

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

The GregorianCalendar has encapsulated instance fields to maintain the date to which it is set.

Mutator and accessor methods

To query one of these settings you use the get method of the GregorianCalendar class. To select the item that you want to get, you pass a constant defined in the Calendar class, such as Calendar.MONTH or Calendar.DAY_OF_WEEK:

```
GregorianCalendar now = new GregorianCalendar();
```

```
int month = now.get(Calendar.MONTH);
```

```
int day = now.get(Calendar.DAY_OF_WEEK);
```

You change the state with a call to the set method:

```
deadline.set(Calendar.YEAR, 2001);
```

```
deadline.set(Calendar.MONTH, Calendar.APRIL);
```

```
deadline.set(Calendar.DAY, 15);
```

There is also a convenience method to set the year, month, and day with a single call:

```
deadline.set(2001, Calendar.APRIL, 15);
```

Finally, you can add a number of days, weeks, months, etc., to a given date.

```
deadline.add(Calendar.MONTH, 3); // move deadline by 3 months
```

If you add a negative number, then the calendar is moved backwards.

There is a conceptual difference between the get method on the one hand and the set and add methods on the other hand. The get method only looks up the state of the object and reports on it. The set and add methods modify the state of the object. Methods that change instance fields are called *mutator methods* and those that only access instance fields without modifying them are called *accessor methods*.

A common convention is to prefix accessor methods with the prefix get and mutator methods with the prefix set. For example, the `GregorianCalendar` class has methods `getTime` and `setTime` that get and set the point in time that a calendar object represents.

```
Date time = calendar.getTime();
```

```
calendar.setTime(time);
```

Building Your Own Classes

The simplest form for a class definition in Java is:

```
class NameOfClass {  
    field1  
    field2  
  
    . . .  
    constructor1  
    constructor2  
  
    . . .  
    method1  
    method2  
    . . .  
}
```

A very simplified version of an `Employee` class that might be used by a business in writing a payroll system.

```
class Employee {  
    private String name;  
    private double salary;  
    private Date hireDay;  
    // constructor  
    public Employee(String n, double s, int year, int month, int day) {  
        name = n;  
        salary = s;  
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);  
        hireDay = calendar.getTime();  
    }  
    // method 1  
    public String getName() {  
        return name;  
    }  
    // more methods  
    . . .  
    // instance fields  
}
```

A program code that shows the `Employee` class in action. In the program, we construct an `Employee` array and fill it with three employee objects:

```
public class EmployeeTest {  
    public static void main(String[] args) {  
        // fill the staff array with three Employee objects  
        Employee[] staff = new Employee[3];
```

```

staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
for (i = 0; i < staff.length; i++)
    staff[i].raiseSalary(5);
for (int i = 0; i < staff.length; i++) {
    Employee e = staff[i];
    System.out.println("name=" + e.getName()
+ ", salary=" + e.getSalary()
+ ", hireDay=" + e.getHireDay());
}

```

Note that the example program consists of *two* classes: the `Employee` class and a class `EmployeeTest` with the public access specifier. The `main` method with the instructions that we just described is contained in the `EmployeeTest` class.

Notice that all of these methods can access the private instance fields by name. This is a key point: instance fields are always accessible by the methods of their own class.

For example,

```

public void raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
    salary += raise;
}

```

sets a new value for the salary instance field in the object that executes this method. (This particular method does not return a value.) For example, the call

```
number007.raiseSalary(5);
```

The `raiseSalary` method has two parameters. The first parameter, called the *implicit* parameter, is the object of type `Employee` that appears before the method name. The second parameter, the number inside the parentheses after the method name, is an *explicit* parameter.

As you can see, the explicit parameters are explicitly listed in the method declaration, for example, `double byPercent`. The implicit parameter does not appear in the method declaration.

In every method, the keyword `this` refers to the implicit parameter. If you like, you can write the `raiseSalary` method as follows:

```

public void raiseSalary(double byPercent) {
    double raise = this.salary * byPercent / 100;
    this.salary += raise;
}

```

Some programmers prefer that style because it clearly distinguishes between instance fields and local variables.

Sometimes, it happens that you want to get and set the value of an instance field. Then you need to supply

three items:

- ❑ A private data field;
- ❑ A public field accessor method;
- ❑ A public field mutator method.

This is a lot more tedious than supplying a single public data field, but there are considerable benefits:

1. The internal implementation can be changed without affecting any code other than the methods of the class.

For example, if the storage of the name is changed to

```
String firstName;
String lastName;
```

then the `getName` method can be changed to return

```
firstName + " " + lastName
```

This change is completely invisible to the remainder of the program.

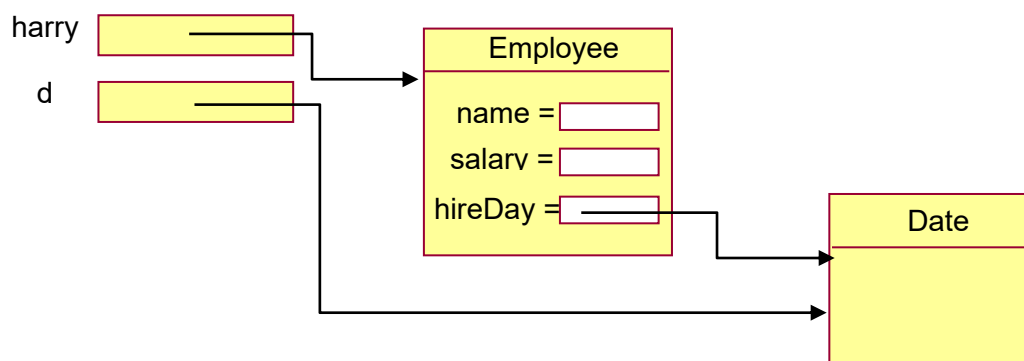
Of course, the accessor and mutator methods may need to do a lot of work and convert between the old and the new data representation. But that leads us to our second benefit.

2. Mutator methods can perform error-checking, whereas code that simply assigns to a field may not go through the trouble.

For example, a `setSalary` method might check that the salary is never less than 0.

```
Employee harry = . . . ;
Date d = harry.getHireDay();
double tenYearsInMilliseconds
= 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long)tenYearsInMilliseconds);
// let's give Harry ten years added seniority
```

The reason is subtle. Both `d` and `harry.hireDay` refer to the same object. Applying mutator methods to `d` automatically changes the private state of the employee object!



Method Access to Private Data

You know that a method can access the private data of the object on which it is invoked. What many people find surprising is that a method can access the private data of *all objects of its class*. For example, consider a method `equals` that compares two employees.

```
class Employee {
    . . .
    boolean equals(Employee other) {
        return name.equals(other.name);
    }
}
```

A typical call is

```
if (harry.equals(boss)) . . .
```

This method accesses the private fields of `harry`, which is not surprising. It also accesses the private fields of `boss`. This is legal because `boss` is an object of type `Employee`, and a method of the `Employee` class is permitted to access the private fields of *any* object of type `Employee`.

Private Methods

When implementing a class, we make all data fields private, since public data are dangerous, and most methods are public. However private methods occur quite frequently. These methods can be called only from other methods of the same class. The reason is simple: to implement certain methods, you may wish to break up the code into many separate methods. Some of these internal methods may not be particularly useful to the public.

To implement a private method in Java, simply change the `public` keyword to `private`.

By making a method private, you are under no obligation to keep it available if you change to another implementation.

In sum, choose private methods:

- ❑ For those methods that are of no concern to the class user ;
- ❑ For those methods that could not easily be supported if the class implementation were to change.

Final Instance Fields

You can define an instance field as final. Such a field must be initialized when the object is constructed. That is, it must be guaranteed that the field value is set after the end of every constructor. Afterwards, the field may not be modified again. For example, the name field of the Employee class may be declared as final since it never changes after the object is constructed there is no setName method.

```
class Employee {
    . . .
    private final String name;
}
```

Static Fields and Methods

Static Fields

If you define a field as static, then there is only one such field per class. In contrast, each object has its own copy of all instance fields. For example, let's suppose we want to assign a unique identification number to each employee. We add an instance field id and a static field nextId to the Employee class:

```
class Employee {
    . . .
    private int id;
    private static int nextId = 1;
}
```

Now, every employee object has its own id field, but there is only one nextId field that is shared among all instances of the class. Let's put it another way. If there are one thousand objects of the Employee class, then there are one thousand instance fields id, one for each object. But there is a single static field nextId. Even if there are no employee objects, the static field nextId is present. It belongs to the class, not to any individual object.

Let's implement a simple method:

```
public void setId() {
    id = nextId;
    nextId++;
}
```

Suppose you set the employee identification number for harry:

```
harry.setId();
```

Then the id field of harry is set, and the value of the static field nextId is incremented:

```
harry.id = . . .;
Employee.nextId++;
```

Constants

Static variables are quite rare. However, static constants are more common.

```
public class Math {
    . . .
    public static final double PI = 3.14159265358979323846;
    . . .
}
```

You can access this constant in your programs as Math.PI.

Static Methods

Static methods are methods that do not operate on objects. For example, the pow method of the Math class is a static method. The expression:

```
Math.pow(x, y)
```

computes the power x^y . It does not use any Math object to carry out its task. In other words, it has no implicit parameter. In other words, you can think of static methods as methods that don't have a this parameter.

Because static methods don't operate on objects, you cannot access instance fields from a static method. But static methods can access the static fields in their class. Here is an example of such a static method:

```
public static int getNextId() {  
    return nextId; // returns static field  
}
```

To call this method, you supply the name of the class:

```
int n = Employee.getNextId();
```

Could you have omitted the keyword `static` for this method? Yes, but then you would need to have an object reference of type `Employee` to invoke the method. You use static methods in two situations:

1. When a method doesn't need to access the object state because all needed parameters are supplied as explicit parameters (example: `Math.pow`);
2. When a method only needs to access static fields of the class (example: `Employee.getNextId`).

Note that you can call static methods without having any objects. For example, you never construct any objects of the `Math` class to call `Math.pow`.

Factory Methods

Here is another common use for static methods. Consider the methods

```
NumberFormat.getNumberInstance()  
NumberFormat.getCurrencyInstance()
```

Each of these methods returns an object of type `NumberFormat`. For example,

```
NumberFormat formatter = NumberFormat.getCurrencyInstance();  
System.out.println(formatter.format(salary));  
// prints salary with currency symbol
```

These are static methods—you call them on a class, not an object. However, their purpose is to generate an object of the same class. Such a method is called a *factory method*.

Why don't we use a constructor instead? There are two reasons. You can't give names to constructors.

The constructor name is always the same as the class name. In the `NumberFormat` example, it makes sense to have two separate names for getting number and currency formatter objects. Furthermore, the factory method can return an object of the type `NumberFormat`, or an object of a subclass that inherits from `NumberFormat`. A constructor does not have that flexibility.

The main method

The main method is a static method.

```
public class Application {  
    public static void main(String[] args) {  
        // construct objects here  
        . . .  
    }  
}
```

The main method does not operate on any objects. In fact, when a program starts, there aren't any objects yet. The static main method executes, and constructs the objects that the program needs.

```
public class StaticTest {  
    public static void main(String[] args){  
        // fill the staff array with three Employee objects  
        Employee[] staff = new Employee[3];  
        staff[0] = new Employee("Tom", 40000);  
        staff[1] = new Employee("Dick", 60000);  
        staff[2] = new Employee("Harry", 65000);  
        // print out information about all Employee objects  
        for (int i = 0; i < staff.length; i++) {  
            Employee e = staff[i];
```

```

        e.setId();
        System.out.println("name=" + e.getName() + ",id=" + e.getId() +
            ",salary=" + e.getSalary());
    }
    int n = Employee.getNextId(); // calls static method
    System.out.println("Next available id=" + n);
}
}
class Employee {
    private String name;
    private double salary;
    private int id;
    private static int nextId = 1;

    public Employee(String n, double s) {
        name = n;
        salary = s;
        id = 0;
    }
    public String getName(){
        return name;
    }
    public double getSalary() {
        return salary;
    }
    public int getId() {
        return id;
    }
    public void setId() {
        id = nextId; // set id to next available id
        nextId++;
    }
    public static int getNextId() {
        return nextId; // returns static field
    }
    public static void main(String[] args) // unit test {
        Employee e = new Employee("Harry", 50000);
        System.out.println(e.getName() + " " + e.getSalary());
    }
}

```

Note that the `Employee` class also has a static `main` method for unit testing. Try running both `java Employee` and `java StaticTest` to execute both main methods.

Method Parameters

Let us review the computer science terms that describe how parameters can be passed to a method (or a function) in a programming language.

The term *call by value* means that the method gets just the value that the caller provides.

In contrast, *call by reference* means that the method gets the *location* of the variable that the caller provides.

Thus, a method can *modify* the value stored in a variable that is passed by reference but not in one that is passed by value.

These "call by ..." terms are standard computer science terminology that describe the behavior of method parameters in various programming languages, not just Java.

The Java programming language *always* uses call by value. That means, the method gets a copy of all parameter values. In particular, the method cannot modify the contents of any parameter variables that are passed to it.

For example, consider the following call:

```
double percent = 10;
harry.raiseSalary(percent);
```

No matter how the method is implemented, we know that after the method call, the value of `percent` is still 10.

Let us look a little more closely at this situation. Suppose a method tried to triple the value of a method parameter:

```
public static void tripleValue(double x) { // doesn't work
    x = 3 * x;
}
```

Let's call this method:

```
double percent = 10;
tripleValue(percent);
```

However, this does not work. After the method call, the value of `percent` is still 10. Here is what happens:

1. `x` is initialized with a copy of the value of `percent` (that is, 10).
2. `x` is tripled—it is now 30. But `percent` is still 10 (see Figure 4-6).
3. The method ends, and the parameter variable `x` is no longer in use.

There are, however, two kinds of method parameters:

- Primitive types (numbers, Boolean values);
- Object references.

You have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object parameters. You can easily implement a method that triples the salary of an employee:

```
public static void tripleSalary(Employee x) { // works
    x.raiseSalary(200);
}
```

When you call

```
harry = new Employee(. . .);
tripleSalary(harry);
```

then the following happens:

1. `x` is initialized with a copy of the value of `harry`, that is, an object reference.
2. The `raiseSalary` method is applied to that object reference. The `Employee` object to which both `x` and `harry` refer gets its salary raised by 200%.
3. The method ends, and the parameter variable `x` is no longer in use. Of course, the object variable `harry` continues to refer to the object whose salary was tripled.

Let's try to write a method that swaps two employee objects:

```
public static void swap(Employee x, Employee y) { // doesn't work
    Employee temp = x;
    x = y;
    y = temp;
}
```

If the Java programming language used call by reference for objects, this method would work:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?
```

However, the method does not actually change the object references that are stored in the variables `a` and `b`. The `x` and `y` parameters of the `swap` method are initialized with *copies* of these references. The method then proceeds to swap these copies.

```
// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice
```

But ultimately, this is a wasted effort. When the method ends, the parameter variables *x* and *y* are abandoned. The original variables *a* and *b* still refer to the same objects as they did before the method call.

This discussion demonstrates that the Java programming language does not use call by reference for objects. Instead, *object references are passed by value*.

Here is a summary of what you can and cannot do with method parameters in the Java programming language:

- ❑ A method cannot modify a parameter of primitive type (that is, numbers or Boolean values).
- ❑ A method can change the *state* of an object parameter.
- ❑ A method cannot make an object parameter refer to a new object.

Object Construction

You have seen how to write simple constructors that define the initial state of your objects.

However,

because object construction is so important, Java offers quite a variety of mechanisms for writing constructors. We will go over these mechanisms in the sections that follow.

Overloading

Recall that the `GregorianCalendar` class had more than one constructor. We could use:

```
GregorianCalendar today = new GregorianCalendar();
```

or:

```
GregorianCalendar deadline = new GregorianCalendar(2099,  
Calendar.DECEMBER, 31);
```

This capability is called *overloading*. Overloading occurs if several methods have the same name (in this case, the `GregorianCalendar` constructor method) but different parameters. The compiler must sort out which method to call. It picks the correct method by matching the parameter types in the headers of the various methods with the types of the values used in the specific method call. A compile-time error occurs if the compiler cannot match the parameters or if more than one match is possible. (This process is called *overloading resolution*.)

Default Field Initialization

If you don't set a field explicitly in a constructor, it is automatically set to a default value: numbers to zero, Booleans to false, and object references to null. But it is considered poor programming practice to rely on this. Certainly, it makes it harder for someone to understand your code if fields are being initialized invisibly.

Default Constructors

A *default constructor* is a constructor with no parameters. (This constructor is sometimes called a *no-arg* constructor.) For example, here is a default constructor for the `Employee` class:

```
public Employee() {  
    name = "";  
    salary = 0;  
    hireDay = new Date();  
}
```

If you write a class with no constructors whatsoever, then a default constructor is provided for you. This default constructor sets *all* the instance fields to their default values. So, all numeric data contained in the instance fields would be zero, all Booleans would be false, and all object variables would be set to null.

If a class supplies at least one constructor but does not supply a default constructor, it is illegal to construct objects without construction parameters. For example, our original `Employee` class provided a single constructor:

```
Employee(String name, double salary, int y, int m, int d)
```

With that class, it was not legal to construct default employees. That is, the call

```
e = new Employee();
```

would have been an error.

Explicit Field Initialization

Since you can overload the constructor methods in a class, you can obviously build in many ways to set the initial state of the instance fields of your classes. It is always a good idea to make sure that, regardless of the constructor call, every instance field is set to something meaningful.

```
class Employee{
    . . .
    private String name = "";
}
```

The initialization value doesn't have to be a constant value:

```
class Employee {
    static int nextId=1;
    private int id = assignId();
    . . .
    static int assignId(){
        int r = nextId;
        nextId++;
        return r;
    }
    . . .
}
```

Calling Another Constructor

The keyword `this` refers to the implicit parameter of a method. However, there is a second meaning for the keyword.

If *the first statement of a constructor* has the form `this(. . .)`, then the constructor calls another constructor of the same class. Here is a typical example:

```
public Employee(double s) {
    // calls Employee(String, double)
    this("Employee #" + nextId, s);
    nextId++;
}
```

Initialization Blocks

You have already seen two ways to initialize a data field:

- ❑ By setting a value in a constructor;
- ❑ By assigning a value in the declaration.

There is actually a *third* mechanism in Java; it's called an *initialization block*. Class declarations can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed.

```
class Employee {
    private String name;
    private double salary;
    public Employee(String n, double s) {
        name = n;
        salary = s;
    }
    public Employee() {
        name = "";
        salary = 0;
    }
    . . .
    // must define before use in initialization block
    private int id;
    private static int nextId;
    // object initialization block
```

```

    {
        id = nextId;
        nextId++;
    }
    . . .
}

```

In this example, the `id` field is initialized in the object initialization block, no matter which constructor is used to construct an object. The initialization block runs first, and then the body of the constructor is executed. This mechanism is never necessary and is not common.

With so many ways of initializing data fields, it can be quite confusing to give all possible pathways for the construction process. Here is what happens in detail when a constructor is called.

1. All data fields are initialized to their default value (0, `false`, or `null`).
2. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. If the first line of the constructor calls another constructor, then the body of that constructor is executed.
4. The body of the constructor is executed.

If the static fields of your class require complex initialization code, use a static initialization block.

```

// static initialization block
static {
    Random generator = new Random();
    nextId = generator.nextInt(10000);
}

```

Static initialization occurs when the class is first loaded. Like instance fields, static fields are 0, `false` or `null` unless you explicitly set them to another value. All static field initializers and static initialization blocks are executed in the order in which they occur in the class declaration.

Object Destruction and the finalize Method

Some object-oriented programming languages, notably C++, have explicit destructor methods for any cleanup code that may be needed when an object is no longer used. The most common activity in a destructor is reclaiming the memory set aside for objects. Since Java does automatic garbage collection, manual memory reclamation is not needed, and Java does not support destructors.

Of course, some objects utilize a resource other than memory, such as a file or a handle to another object that uses system resources. In this case, it is important that the resource be reclaimed and recycled when it is no longer needed.

You can add a `finalize` method to any class. The `finalize` method will be called before the garbage collector sweeps away the object. In practice, *do not rely on the `finalize` method* for recycling any resources that are in short supply—you simply cannot know when this method will be called.

If a resource needs to be closed as soon as you have finished using it, you need to manage it manually. Add a `dispose` method that *you* call to clean up what needs cleaning. Just as importantly, if a class you use has a `dispose` method, you will want to call it when you are done with the object. In particular, if your class has an instance field that has a `dispose` method, provide a `dispose` method that disposes of the instance fields.

Multidimensional Arrays

Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. You can safely skip this section until you have a need for this storage mechanism.

Suppose you want to make a table of numbers that shows how much an investment of \$10,000 will grow under different interest rate scenarios in which interest is paid annually and reinvested. Table 1 illustrates this scenario.

The obvious way to store this information is in a two-dimensional array (or matrix), which we will call `balance`.

Declaring a matrix in Java is simple enough. For example:

```
double[][] balance;
```

As always, you cannot use the array until you initialize it with a call to `new`. In this case, you can do the initialization as follows:

```
balance = new double[NYEARS][NRATES];
```

In other cases, if you know the array elements, you can use a shorthand notion for initializing multidimensional arrays without needing a call to `new`. For example;

```
int[][] magicSquare =
{
    {16, 3, 2, 13},
    {5, 10, 11, 8},
    {9, 6, 7, 12},
    {4, 15, 14, 1}
};
```

Table 1. Growth of an investment at different interest rates

10%	11%	12%	13%	14%	15%
\$10,000.00	\$10,000.00	\$10,000.00	\$10,000.00	\$10,000.00	\$10,000.00
\$11,000.00	\$11,100.00	\$11,200.00	\$11,300.00	\$11,400.00	\$11,500.00
\$12,100.00	\$12,321.00	\$12,544.00	\$12,769.00	\$12,996.00	\$13,225.00

Once the array is initialized, you can access individual elements, by supplying two brackets, for example `balance[i][j]`.

The example program stores a one-dimensional array `interest` of interest rates and a two-dimensional array `balance` of account balances, one for each year and interest rate.

```
1. import java.text.*;
2. import javax.swing.*;
4. public class CompoundInterest {
6.     public static void main(String[] args) {
8.         final int STARTRATE = 10;
9.         final int NRATES = 6;
10.        final int NYEARS = 10;
12.        // set interest rates to 10 . . . 15%
13.        double[] interestRate = new double[NRATES];
14.        for (int j = 0; j < interestRate.length; j++)
15.            interestRate[j] = (STARTRATE + j) / 100.0;
17.        double[][] balance = new double[NYEARS][NRATES];
19.        // set initial balances to 10000
20.        for (int j = 0; j < balance[0].length; j++)
21.            balance[0][j] = 10000;
23.        // compute interest for future years
25.        for (int i = 1; i < balance.length; i++) {
27.            for (int j = 0; j < balance[i].length; j++) {
29.                // get last year's balance from previous row
30.                double oldBalance = balance[i - 1][j];
32.                // compute interest
33.                double interest = oldBalance * interestRate[j];
34.
35.                // compute this year's balance
36.                balance[i][j] = oldBalance + interest;
37.            }
38.        }
40.        // print one row of interest rates
42.        NumberFormat formatter = NumberFormat.getPercentInstance();
44.        for (int j = 0; j < interestRate.length; j++) {
46.            System.out.print(" ");
47.            System.out.print(formatter.format(interestRate[j]));
48.        }
49.        System.out.println();
51.        // print balance table
53.        formatter = NumberFormat.getCurrencyInstance();
55.        for (int i = 0; i < balance.length; i++) {
57.            // print table row
58.            for (int j = 0; j < balance[i].length; j++) {
60.                System.out.print(" ");
```

```

61.                 System.out.print(formatter.format(balance[i][j]));
62.             }
63.             System.out.println();
64.         }
65.     }
66. }

```

Ragged Arrays

So far, what you have seen is not too different from other programming languages. But there is actually something subtle going on behind the scenes that you can sometimes turn to your advantage: Java has *no* multidimensional arrays at all, only one-dimensional arrays.

Multidimensional arrays are faked as "arrays of arrays."

For example, the `balance` array in the preceding example is actually an array that contains ten elements, each of which is an array of six floating-point numbers

It is also easy to make "ragged" arrays, that is, arrays in which different rows have different lengths. Here is the standard example. Let us make an array in which the entry at row *i* and column *j* equals the number of possible outcomes of a "choose *j* numbers from *i* numbers" lottery.

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```

Because *j* can never be larger than *i*, the matrix is triangular. The *i*-th row has *i* + 1 elements. (We allow choosing 0 elements; there is one way to make such a choice.) To build this ragged array, first allocate the array holding the rows.

```

int[][] odds = new int[NMAX + 1][];
1. public class LotteryArray
2. {
3.     public static void main(String[] args)
4.     {
5.         final int NMAX = 10;
6.
7.         // allocate triangular array
8.         int[][] odds = new int[NMAX + 1][];
9.         for (int n = 0; n <= NMAX; n++)
10.            odds[n] = new int[n + 1];
11.
12.        // fill triangular array
13.        for (int n = 0; n < odds.length; n++)
14.            for (int k = 0; k < odds[n].length; k++)
15.            {
16.                /*
17.                compute binomial coefficient
18.                n * (n - 1) * (n - 2) * . . . * (n - k + 1)
19.                -----
20.                1 * 2 * 3 * . . . * k
21.                */
22.                int lotteryOdds = 1;
23.                for (int i = 1; i <= k; i++)
24.                    lotteryOdds = lotteryOdds * (n - i + 1) / i;
25.
26.                odds[n][k] = lotteryOdds;
27.            }
28.
29.        // print triangular array
30.        for (int n = 0; n < odds.length; n++)
31.        {
32.            for (int k = 0; k < odds[n].length; k++)
33.            {
34.                // pad output with spaces

```

```

35. String output = " " + odds[n][k];
36. // make output field 4 characters wide
37. output = output.substring(output.length() - 4);
38. System.out.print(output);
39. }
40. System.out.println();
41. }
42. }
43. }

```

Packages

Java allows you to group classes in a collection called a *package*. Packages are convenient for organizing your work and for separating your work from code libraries provided by others. The standard Java library is distributed over a number of packages, including `java.lang`, `java.util`, `java.net`, and so on. The standard Java packages are examples of hierarchical packages. Just as you have nested subdirectories on your hard disk, you can organize packages by using levels of nesting. All standard Java packages are inside the `java` and `javax` package hierarchies. The main reason for using packages is to guarantee the uniqueness of class names.

A package can then be further subdivided into subpackages such as `java.util.jar`. The sole purpose of package nesting is to manage unique names. From the point of view of the compiler, there is absolutely no relationship between nested packages. For example, the packages `java.util` and `java.util.jar` have nothing to do with each other. Each is its own independent collection of classes. A class can use all classes from its own package and all *public* classes from other packages. You can access the public classes in another package in two ways. The first is simply to add the full package name in front of *every* classname. For example:

```
java.util.Date today = new java.util.Date();
```

That is obviously tedious. The simpler, and more common, approach is to use the `import` keyword. The point of the `import` statement is simply to give you a shorthand to refer to the classes in the package. Once you use `import`, you no longer have to give the classes their full names.

You can import a specific class or the whole package. You place `import` statements at the top of your source files (but below any package statements). For example, you can import all classes in the `java.util` package with the statement:

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

Then you can use

```
Date today = new Date();
```

without a package prefix. You can also import a specific class inside a package.

```
import java.util.Date;
```

Importing all classes in a package is simpler. It has no negative effect on code size, so there is generally no reason not to do it.

However, note that you can only use the `*` notation to import a single package. You cannot use `import java.*` or `import java.*.*` to import all packages with the `java` prefix.

The only time that you need to pay attention to packages is when you have a name conflict.

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

```
import java.sql.*;
```

If you now use the `Date` class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

You can solve this problem by adding a specific `import` statement:

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

```
import java.sql.*;
```

```
import java.util.Date;
```

or:

```
java.util.Date deadline = new java.util.Date();
```

```
java.sql.Date today = new java.sql.Date();
```

Adding a class into a package

To place classes inside a package, you must put the name of the package at the top of your source file, *before* the code that defines the classes in the package.

If you don't put a package statement in the source file, then the classes in that source file belong to the *default package*. The default package has no package name.

```
. (base directory)
PackageTest.java
PackageTest.class
com/
    resources/
        staff/
            Employee.java
            Employee.class
    appl/
        PayrollApp.java
        PayrollApp.class
```

The code should declare the package: `com.resources.staff`:

```
package com.resources.staff;
class Employee {
```

```
....
```

In this situation, you still must compile and run classes from the *base* directory, that is, the directory containing the `com` directory:

```
javac com/appl/PayrollApp.java
```

```
java com.appl.PayrollApp
```

Then:

```
import com.resources.staff.*;
public class PackageTest {
    public static void main(String[] args) {
        Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
    }
    ....
```

The path to the class must match the package name. You can also use the JAR utility to add class files to an *archive*. An archive contains multiple class files and subdirectories inside a single file, saving space and reducing access time. For example, the thousands of classes of the runtime library are all contained in the runtime library file `rt.jar`. You can find that file in the `jre/lib` subdirectory of the Java SDK.

Generally, multiple programs need to access package files. To share your packages among programs, you need to do the following:

1. Place your classes inside one or more special directories, say `/home/user/classdir`. Note that this directory is the *base* directory for the package tree. If you add the class `com.resources.staff.Employee`, then the class file must be located in the subdirectory `/home/user/classdir/com/resources/staff`.
2. Set the *class path*. The class path is the collection of all base directories whose subdirectories can contain class files.

This class path contains:

- ❑ The base directory `/home/user/classdir` or `c:\classes`
- ❑ The current directory `.`
- ❑ The JAR file `/home/user/archives/archive.jar` or `c:\archives\archive.jar`

The runtime library files (`rt.jar` and the other JAR files in the `jre/lib` and `jre/lib/ext` directories) are always searched for classes; you don't include them explicitly in the class path.

Suppose the interpreter searches for the class file of the `com.horstmann.corejava.Employee` class. It first looks in the system class files which are stored in archives in the `jre/lib` and `jre/lib/ext` directories. It won't find the class file there, so it turns to the class path. It then looks for the following files:

- ❑ `/home/user/classdir/com/resources/staff/Employee.class`
- ❑ `com/resources/staff/Employee.class` starting from the current directory

- ❑ `com/resources/staff/Employee.class` inside `/home/user/archives/archive.jar`.

Class Design Hints

1. *Always keep data private.*

This is first and foremost: doing anything else violates encapsulation. You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private. Bitter experience has shown that how the data are represented may change, but how they are used will change much less frequently. When data are kept private, changes in their representation do not affect the user of the class, and bugs are easier to detect.

2. *Always initialize data.*

Java won't initialize local variables for you, but it will initialize instance fields of objects. Don't rely on the defaults, but initialize the variables explicitly, either by supplying a default or by setting defaults in all constructors.

3. *Don't use too many basic types in a class.*

The idea is to replace multiple *related* uses of basic types with other classes. This keeps your classes easier to understand and to change. For example, replace the following instance fields in a `Customer` class

```
private String street;  
private String city;  
private String state;  
private int zip;
```

with a new class called `Address`. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

4. *Not all fields need individual field accessors and mutators.*

You may need to get and set an employee's salary. You certainly won't need to change the hiring date once the object is constructed. And, quite often, objects have instance fields that you don't want others to get or set, for example, an array of state abbreviations in an `Address` class.

5. *Use a standard form for class definitions.*

We always list the contents of classes in the following order:

```
public features  
package scope features  
private features
```

Within each section, we list:

```
instance methods  
static methods  
instance fields  
static fields
```

After all, the users of your class are more interested in the public interface than in the details of the private implementation. And they are more interested in methods than in data. However, there is no universal agreement on what is the best style. The Sun coding style guide for the Java programming language recommends listing fields first and then methods. Whatever style you use, the most important thing is to be consistent.

6. *Break up classes with too many responsibilities.*

This hint is, of course, vague: "too many" is obviously in the eye of the beholder. However, if there is an obvious way to make one complicated class into two classes that are conceptually simpler, seize the opportunity. (On the other hand, don't go overboard; 10 classes, each with only one method, is usually overkill.)

Here is an example of a bad design.

```
class CardDeck // bad design {  
public CardDeck() { . . . }  
public void shuffle() { . . . }  
public int getTopValue() { . . . }  
public int getTopSuit() { . . . }  
public void draw() { . . . }  
private int[] value;
```

```
private int[] suit;
}
```

This class really implements two separate concepts: a *deck of cards*, with its `shuffle` and `draw` methods, and a *card*, with the methods to inspect the value and suit of a card. It makes sense to introduce a `Card` class that represents an individual card. Now you have two classes, each with its own responsibilities:

```
class CardDeck {
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public Card getTop() { . . . }
    public void draw() { . . . }
    private Card[] cards;
}
class Card {
    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }
    private int value;
    private int suit;
}
```

7. Make the names of your classes and methods reflect their responsibilities.

Just as variables should have meaningful names that reflect what they represent, so should classes. (The standard library certainly contains some dubious examples, such as the `Date` class that describes time.)

A good convention is that a class name should be a noun (`Order`) or a noun preceded by an adjective (`RushOrder`) or a gerund (an "-ing" word, like `BillingAddress`). As for methods, follow the standard convention that accessor methods begin with a lowercase `get` (`getSalary`), and mutator methods use a lowercase `set` (`setSalary`).

Inheritance

The idea behind inheritance is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you reuse (or inherit) methods and fields, and you add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

Suppose you work for a company at which managers are treated differently than other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for receiving their salary, managers get *bonuses* if they actually achieve what they are supposed to do. This is the kind of situation that cries out for inheritance.

There is an obvious "is-a" relationship between `Manager` and `Employee`. Every manager *is an* employee. You can retain some of what you have already programmed in the `Employee` class, and *all* the fields of the original class could be preserved in `Manager`. You need only to define a new class, `Manager`, based on `Employee` and add functionality.

You can define a `Manager` class that inherits from the `Employee` class by using the keyword `extends` to denote inheritance.

```
class Manager extends Employee {
    added methods and fields
}
```

The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the *superclass*, *base class*, or *parent class*. The new class is called the *subclass*, *derived class*, or *child class*. The terms superclass and subclass are those most commonly used by Java programmers, although some programmers prefer the parent/child analogy, which also ties in nicely with the "inheritance" theme.

The `Employee` class is a superclass, but not because it is superior to its subclass or contains more functionality. *In fact, the opposite is true*: subclasses have *more* functionality than their superclasses. For example, as you will see when we go over the rest of the `Manager` class code,

the Manager class encapsulates more data and has more functionality than its superclass Employee.

Our Manager class has a new field to store the bonus, and a new method to set it:

```
class Manager extends Employee {  
    . . .  
    public void setBonus(double b) {  
        bonus = b;  
    }  
    private double bonus;  
}
```

There is nothing special about these methods and fields. If you have a manager object, you can simply apply the setBonus method.

```
Manager boss = . . .;  
boss.setBonus(5000);
```

Of course, if you have an Employee object, you cannot apply the setBonus method—it is not among the methods that are defined in the Employee class.

However, you *can* use methods such as getName and getHireDay with Manager objects. Even though these methods are not explicitly defined in the Manager class, they are automatically inherited from the Employee superclass. Similarly, the fields name, salary, and hireDay are inherited from the superclass. Every Manager object has four fields: name, salary, hireDay and bonus.

When designing classes, you place the most general methods into the superclass, and more specialized methods in the subclass. Factoring out common functionality by moving it to a superclass is very common in object-oriented programming.

However, some of the superclass methods are not appropriate for the Manager subclass. In particular, the getSalary method should return the sum of the base salary and the bonus. You need to supply a new

method to *override* the superclass method:

```
class Manager extends Employee {  
    . . .  
    public double getSalary() {  
        . . .  
    }  
    . . .  
}
```

How can you implement this method? At first glance, it appears to be simple: Just return the sum of the salary and bonus fields:

```
public double getSalary() {  
    return salary + bonus; // won't work  
}
```

However, that won't work. The getSalary method of the Manager class *has no direct access to the private fields of the superclass*. This means that the getSalary method of the Manager class cannot directly access the salary field, even though every Manager object has a field called salary. Only the methods of the Employee class have access to the private fields. If the Manager methods want to access those private fields, they have to do what every other method does—use the public interface, in this case, the public getSalary method of the Employee class.

So, let's try this again. You need to call getSalary instead of simply accessing the salary field.

```
public double getSalary() {  
    double baseSalary = getSalary(); // still won't work  
    return baseSalary + bonus;  
}
```

The problem is that the call to getSalary simply calls *itself*, because the Manager class has a getSalary method (namely the method we are trying to implement). The consequence is an infinite set of calls to the same method, which leads to a program crash.

We need to indicate that we want to call the `getSalary` method of the `Employee` superclass, not the current class. You use the special keyword `super` for this purpose: The call

```
super.getSalary()
```

calls the `getSalary` method of the `Employee` class. Here is the correct version of the `getSalary` method for the `Manager` class:

```
public double getSalary() {  
    double baseSalary = super.getSalary();  
    return baseSalary + bonus;  
}
```

As you saw, a subclass can *add* fields, and it can *add* or *override* methods of the superclass.

However, inheritance can never take away any fields or methods.

Finally, let us supply a constructor.

```
public Manager(String n, double s, int year, int month, int day) {  
    super(n, s, year, month, day);  
    bonus = 0;  
}
```

Here, the keyword `super` has a different meaning. The instruction

```
super(n, s, year, month, day);
```

is shorthand for "call the constructor of the `Employee` superclass with `n`, `s`, `year`, `month`, and `day` as parameters."

Since the `Manager` constructor cannot access the private fields of the `Employee` class, it must initialize them through a constructor. The constructor is invoked with the special `super` syntax. The call using `super` must be the first statement in the constructor for the subclass.

If the subclass constructor does not call a superclass constructor explicitly, then the superclass uses its default (no-parameter) constructor. If the superclass has no default constructor and the subclass constructor does not call another superclass constructor explicitly, then the Java compiler reports an error.

Having redefined the `getSalary` method for `Manager` objects, managers will *automatically* have the bonus added to their salaries.

Here's an example of this at work: we make a new manager and set the manager's bonus:

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);  
boss.setBonus(5000);
```

We make an array of three employees:

```
Employee[] staff = new Employee[3];
```

We populate the array with a mix of managers and employees:

```
staff[0] = boss;  
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);  
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

We print out everyone's salary:

```
for (int i = 0; i < staff.length; i++) {  
    Employee e = staff[i];  
    System.out.println(e.getName() + " " + e.getSalary());  
}
```

This loop prints the following data:

```
Carl Cracker 85000.0  
Harry Hacker 50000.0  
Tommy Tester 40000.0
```

Now `staff[1]` and `staff[2]` each print their base salary because they are `Employee` objects. However, `staff[0]` is a `Manager` object and its `getSalary` method adds the bonus to the base salary.

What is remarkable is that the call

```
e.getSalary()
```

picks out the *correct* `getSalary` method. Note that the *declared* type of `e` is `Employee`, but the *actual*

type of the object to which `e` refers can be either `Employee` (that is, when `i` is 1 or 2) or `Manager` (when `i` is 0).

When `e` refers to an `Employee` object, then the call `e.getSalary()` calls the `getSalary` method of the `Employee` class. However, when `e` refers to a `Manager` object, then the `getSalary` method of the `Manager` class is called instead. The virtual machine knows about the actual type of the object to which `e` refers, and therefore can invoke the correct method.

The fact that an object variable (such as the variable `e`) can refer to multiple actual types is called *polymorphism*. Automatically selecting the appropriate method at runtime is called *dynamic binding*.

Inheritance Hierarchies

Inheritance need not stop at deriving one layer of classes. We could have an `Executive` class that derives from `Manager`, for example. The collection of all classes extending from a common superclass is called an *inheritance hierarchy*. The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain*.

There is usually more than one chain of descent from a distant ancestor class. You could derive a `Programmer` class from `Employee` class or a `Secretary` class from `Employee`, and they would have nothing to do with the `Manager` class (or with each other). This process can continue as long as is necessary.

Polymorphism

There is a simple rule to know whether or not inheritance is the right design for your data. The "is-a" rule states that every object of the subclass is an object of the superclass. For example, every manager is an employee. Thus, it makes sense for the `Manager` class to be a subclass of the `Employee` class.

Naturally, the opposite is not true—not every employee is a manager.

Another way of formulating the "is-a" rule is the *substitution principle*. That principle states that you can use a subclass object whenever the program expects a superclass object.

For example, you can assign a subclass object to a superclass variable.

```
Employee e;  
e = new Employee(. . .); // Employee object expected  
e = new Manager(. . .); // OK, Manager can be used as well
```

In the Java programming language, object variables are *polymorphic*. A variable of type `Employee` can refer to an object of type `Employee` or an object of any subclass of the `Employee` class (such as `Manager`, `Executive`, `Secretary`, and so on).

We took advantage of this principle in:

```
Employee[] staff = new Employee[3];  
Manager boss = new Manager(. . .);  
staff[0] = boss;
```

In this case, the variables `staff[0]` and `boss` refer to the same object. However, `staff[0]` is considered to be only an `Employee` object by the compiler. That means, you can call

```
boss.setBonus(5000); // OK
```

but you can't call

```
staff[0].setBonus(5000); // ERROR
```

The declared type of `staff[0]` is `Employee`, and the `setBonus` method is not a method of the `Employee` class.

However, you cannot assign a superclass reference to a subclass variable. For example, it is not legal to make the assignment:

```
Manager m = staff[i]; // ERROR
```

The reason is clear: Not all employees are managers. If this assignment were to succeed and `m` were to refer to an `Employee` object that is not a manager, then it would later be possible to call `m.setBonus(...)`, and a runtime error would occur.

Dynamic Binding

It is important to understand what happens when a method call is applied to an object. Here are the details:

1. The compiler looks at the declared type of the object and the method name. Let's say we call `x.f(args)`, and the implicit parameter `x` is declared to be an object of class `C`. Note that there may be multiple methods, all with the same name `f`, but with different parameter types. For example, there may be a method `f(int)` and a method `f(String)`. The compiler enumerates all methods called `f` in the class `C` and all public methods called `f` in the superclasses of `C`.
Now the compiler knows all possible candidates for the method to be called.
2. Next, the compiler determines the types of the parameters that are supplied in the method call. If among all the methods called `f` there is a unique method whose parameter types are a best match for the supplied parameters, then that method is chosen to be called. This process is called *overloading resolution*. For example, in a call `x.f("Hello")`, the compiler picks `f(String)` and not `f(int)`. The situation can get complex because of type conversions (int to double, Manager to Employee, and so on). If the compiler cannot find any method with matching parameter types, or if there are multiple methods that all match after applying conversions, then the compiler reports an error.
Now the compiler knows the name and parameter types of the method that needs to be called.
3. If the method is private, static, final, or a constructor, then the compiler knows exactly which method to call. (The final modifier is explained in the next section.) This is called *static binding*. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at run time. In our example, the compiler would generate an instruction to call `f(String)` with dynamic binding.
4. When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the *actual* type of the object to which `x` refers. Let's say the actual type is `D`, a subclass of `C`. If the class `D` defines a method `f(String)`, that method is called. If not, `D`'s superclass is searched for a method `f(String)`, and so on.

It would be time-consuming to carry out this search every time a method is called.

Therefore, the virtual machine precomputes a *method table* for each class that lists all method signatures and the actual methods to be called. When a method is actually called, the virtual machine simply makes a table lookup. In our example, the virtual machine consults the method table for the class `D` and looks up the method to call for `f(String)`. That method may be `D.f(String)` or `X.f(String)`, where `X` is some superclass of `D`.

There is one twist to this scenario. If the call is `super.f(args)`, then the compiler consults the method table of the superclass of the implicit parameter.

Casting

Recall that the process of forcing a conversion from one type to another is called *casting*.

The Java programming language has a special notation for casts. For example:

```
double x = 3.405;
int nx = (int)x;
```

converts the value of the expression `x` into an integer, discarding the fractional part.

Just as you occasionally need to convert a floating-point number to an integer, you also need to convert an object reference from one class to another. To actually make a cast of an object reference, you use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast. For example:

```
Manager boss = (Manager)staff[0];
```

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the `Manager` class, the `staff` array had to be an array of `Employee` objects since *some* of its entries were regular employees. We would need to cast the managerial elements of the array back to `Manager` to access any of its new variables. (Note that in the sample code for the first section, we made a special effort to avoid the cast. We initialized the `boss` variable with a `Manager` object before storing it in the array. We needed the correct type to set the bonus of the manager.)

As you know, in Java, every object variable has a type. The type describes the kind of object the variable refers to and what it can do. For example, `staff[i]` refers to an `Employee` object (so it can also refer to a `Manager` object).

You rely on these descriptions in your code, and the compiler checks that you do not promise too much when you describe a variable. If you assign a subclass object to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass object to a subclass variable, you are promising more, and you must confirm that you mean what you say to the compiler with the (Subclass) cast notation.

What happens if you try to cast down an inheritance chain and you are "lying" about what an object contains?

```
Manager boss = (Manager)staff[1]; // ERROR
```

When the program runs, the Java runtime system notices the broken promise, and generates an exception. If you do not catch the exception, your program terminates. Thus, it is good programming practice to find out whether a cast will succeed before attempting it. Simply use the `instanceof` operator.

For example:

```
if (staff[1] instanceof Manager) {
    boss = (Manager)staff[1];
    . . .
}
```

Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed. For example, the cast

```
Date c = (Date)staff[1];
```

is a compile-time error because `Date` is not a subclass of `Employee`.

To sum up:

- ❑ You can cast only within an inheritance hierarchy.
- ❑ Use `instanceof` to check before casting from a superclass to a subclass.

Actually, converting the type of an object by performing a cast is not usually a good idea. In our example, you do not need to cast an `Employee` object to a `Manager` object for most purposes. The `getSalary` method will work correctly on both types because the dynamic binding that makes polymorphism work locates the correct method automatically.

The only reason to make the cast is to use a method that is unique to managers, such as `setBonus`. If you for some reason find yourself wanting to call `setBonus` on `Employee` objects, ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a `setBonus` method. Remember, it takes only one bad cast to terminate your program. In general, it is best to minimize the use of casts and the `instanceof` operator.

Abstract Classes

As you move up the inheritance hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes so general that you think of it more as a basis for other classes than as a class with specific instances you want to use. Consider, for example, an extension of our `Employee` class hierarchy. An employee is a person, and so is a student. Let us extend our class hierarchy to include classes `Person` and `Student`.

Both students and employees have names, and introducing a common superclass lets us factor out the `getName` method to a higher level in the inheritance hierarchy.

Now let's add another method, `getDescription`, whose purpose is to return a brief description of the person, such as:

an employee with a salary of \$50,000.00

a student majoring in computer science

It is easy to implement this method for the `Employee` and `Student` classes. But what information can you provide in the `Person` class? The `Person` class knows nothing about the person except for the name. Of course, you could implement `Person.getDescription()` to return an empty string. But there is a better way. If you use the `abstract` keyword, you do not need to implement the method at all.

```
public abstract String getDescription(); // no implementation required
```

For added clarity, a class with one or more abstract methods must itself be declared abstract.

```
abstract class Person {  
    . . .  
    public abstract String getDescription();  
}
```

In addition to abstract methods, abstract classes can have concrete data and methods. For example, the Person class stores the name of the person and has a concrete method that returns it.

```
abstract class Person {  
    private String name;  
    public Person(String n) {  
        name = n;  
    }  
    public abstract String getDescription();  
    public String getName() {  
        return name;  
    }  
}
```

Abstract methods act as placeholder methods that are implemented in the subclasses. When you extend an abstract class, you have two choices. You can leave some or all of the abstract methods undefined.

Then you must tag the subclass as abstract as well. Or you can define all methods. Then the subclass is no longer abstract.

For example, we will define a Student class that extends the abstract Person class and implements the getDescription method. Because none of the methods of the Student class are abstract, it does not need to be declared as an abstract class.

Abstract classes cannot be instantiated. That is, if a class is declared as abstract, no objects of that class can be created.

However, you can create objects of concrete subclasses.

Note that you can still create *object variables* of an abstract class, but such a variable must refer to an object of a nonabstract subclass. For example,

```
Person p = new Student("Vince Vu", "Economics");
```

Here p is a variable of the abstract type Person that refers to an instance of the non-abstract subclass Student.

Let us define a concrete subclass Student that extends the abstract Person class:

```
class Student extends Person {  
    private String major;  
    public Student(String n, String m) {  
        super(n);  
        major = m;  
    }  
    public String getDescription() {  
        return "a student majoring in " + major;  
    }  
}
```

The Student class defines the getDescription method. Therefore, all methods in the Student class are concrete, and the class is no longer an abstract class.

The following program defines the abstract superclass Person and two concrete subclasses Employee and Student. We fill an array of Person references with employee and student objects.

```
Person[] people = new Person[2];  
people[0] = new Employee(. . .);  
people[1] = new Student(. . .);
```

We then print the names and descriptions of these objects:

```
for (int i = 0; i < people.length; i++) {  
    Person p = people[i];  
    System.out.println(p.getName() + ", " + p.getDescription());  
}
```

```
}
```

Some people are baffled by the call:

```
p.getDescription()
```

Isn't this a call an undefined method? Keep in mind that the variable `p` never refers to a `Person` object since it is impossible to construct a `Person` object. The variable `p` always refers to an object of a concrete subclass such as `Employee` or `Student`. For these objects, the `getDescription` method is defined.

Could you have omitted the abstract method altogether from the `Person` superclass and simply defined the `getDescription` methods in the `Employee` and `Student` subclasses? Then you wouldn't have been able to invoke the `getDescription` method on the variable `p`. The compiler ensures that you only invoke methods that are declared in the class.

Abstract methods are an important concept in the Java programming language. You will encounter them most commonly inside *interfaces*.

```
import java.text.*;
```

```
public class PersonTest {
```

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

```
        Person[] people = new Person[2];
```

```
        // fill the people array with Student and Employee objects
```

```
        people[0] = new Employee("Harry Hacker", 50000);
```

```
        people[1] = new Student("Maria Morris", "computer science");
```

```
        // print out names and descriptions of all Person objects
```

```
        for (int i = 0; i < people.length; i++) {
```

```
            Person p = people[i];
```

```
            System.out.println(p.getName() + ", " + p.getDescription());
```

```
        }
```

```
    }
```

```
}
```

```
abstract class Person {
```

```
    private String name;
```

```
    public Person(String n) {
```

```
        name = n;
```

```
    }
```

```
    public abstract String getDescription();
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
}
```

```
class Employee extends Person {
```

```
    private double salary;
```

```
    public Employee(String n, double s) {
```

```
        // pass name to superclass constructor
```

```
        super(n);
```

```
        salary = s;
```

```
    }
```

```
    public double getSalary() {
```

```
        return salary;
```

```
    }
```

```
    public String getDescription() {
```

```
        NumberFormat formatter = NumberFormat.getCurrencyInstance();
```

```
        return "an employee with a salary of " + formatter.format(salary);
```

```
    }
```

```
    public void raiseSalary(double byPercent) {
```

```
        double raise = salary * byPercent / 100;
```

```
        salary += raise;
```

```
    }
```

```
}
```

```
class Student extends Person {
```

```

private String major;
public Student(String n, String m) {
    // pass n to superclass constructor
    super(n);
    major = m;
}
public String getDescription() {
    return "a student majoring in " + major;
}
}

```

Protected Access

As you know, fields in a class are best tagged as private, and methods are usually tagged as public. Any features declared private won't be visible to other classes. As we said at the beginning of this chapter, this is also true for subclasses: a subclass cannot access the private fields of its superclass.

There are times, however, when you want to restrict a method to subclasses only, or, less commonly, to allow subclass methods to access a superclass field. In that case, you declare a class feature as protected. For example, if the superclass `Employee` declares the `hireDay` field as protected instead of private, then the `Manager` methods can access it directly.

However, the `Manager` class methods can only peek inside the `hireDay` field of `Manager` objects, not of other `Employee` objects. This restriction is made so that you can't abuse the protected mechanism and form subclasses just to gain access to the protected fields.

In practice, use the `protected` attribute with caution. Suppose your class is used by other programmers and you designed it with protected fields. Unbeknownst to you, other programmers may inherit classes from your class and then start accessing your protected fields. In this case, you can no longer change the implementation of your class without upsetting the other programmers. That is against the spirit of OOP, which encourages data encapsulation.

Protected methods make more sense. A class may declare a method as `protected` if it is tricky to use. This indicates that the subclasses (which, presumably, know their ancestors well) can be trusted to use the method correctly, but other classes cannot.

Here is a summary of the four access modifiers in Java that control visibility:

- ❑ Visible to the class only (private).
- ❑ Visible to the world (public).
- ❑ Visible to the package and all subclasses (protected).
- ❑ Visible to the package-the (unfortunate) default. No modifiers are needed.

The Superclass Object

The `Object` class is the ultimate ancestor—every class in Java extends `Object`. However, you never have to write:

```
class Employee extends Object
```

The ultimate superclass `Object` is taken for granted if no superclass is explicitly mentioned.

Because every class in Java extends `Object`, it is important to be familiar with the services provided by the `Object` class. We will go over the basic ones in this chapter and refer you to later chapters or to the on-line documentation for what is not covered here.

You can use a variable of type `Object` to refer to objects of any type:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Of course, a variable of type `Object` is only useful as a generic holder for arbitrary values. To do anything specific with the value, you need to have some knowledge about the original type and then apply a cast:

```
Employee e = (Employee) obj;
```

The equals and toString methods

The `equals` method in the `Object` class tests whether or not one object is equal to another. The `equals` method, as implemented in the `Object` class, determines whether or not two objects point to

the same area of memory. This is not a useful test. If you want to test objects for equality, you will need to override equals for a more meaningful comparison. For example,

```
class Employee { // . . .
    public boolean equals(Object otherObject) {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;
        // must return false if the explicit parameter is null
        if (otherObject == null) return false;
        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass())
            return false;
        // now we know otherObject is a non-null Employee
        Employee other = (Employee)otherObject;
        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```

The getClass method returns the class of an object. For two objects to be equal, they must first be objects of the same class.

Class getClass()

returns a class object that contains information about the object. Java has a runtime representation for classes that is encapsulated in the Class class that you can often use to your advantage.

If you define the equals method for a subclass of a class that follows these rules, first call equals on the superclass. If that test doesn't pass, then the objects can't be equal. If the superclass fields are equal, then you are ready to compare the instance fields of the subclass.

```
class Manager extends Employee {
    . . .
    public boolean equals(Object otherObject) {
        if (!super.equals(otherObject)) return false;
        Manager other = (Manager)otherObject;
        // super.equals checked that this and otherObject
        // belong to the same class
        return bonus == other.bonus;
    }
}
```

Another important method in Object is the toString method that returns a string that represents the value of this object. Almost any class will override this method to give you a printed representation of the object's current state. Here is a typical example. The toString method of the Point class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Most (but not all) toString methods follow this format: the name of the class, followed by the field values enclosed in square brackets. Here is an implementation of the toString method for the Employee class:

```
public String toString() {
    return "Employee[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

Actually, you can do a little better. Rather than hardwiring the class name into the toString method, call getClass().getName() to obtain a string with the class name.

```
public String toString() {
    return getClass().getName()
        + "[name=" + name
```

```

    + ",salary=" + salary
    + ",hireDay=" + hireDay
    + "];"
}

```

Then the `toString` method also works for subclasses. Of course, the subclass programmer should define its own `toString` method and add the subclass fields. If the superclass uses `getClass().getName()`, then the subclass can simply call `super.toString()`. For example, here is a `toString` method for the `Manager` class:

```

class Manager extends Employee {
    . . .
    public String toString() {
        return super.toString() + "[bonus=" + bonus + "];"
    }
}

```

Now a `Manager` object is printed as:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

The `toString` method is ubiquitous for an important reason: whenever an object is concatenated with a string, using the `+` operator, the Java compiler automatically invokes the `toString` method to obtain a string representation of the object. For example,

```

Point p = new Point(10, 20);
String message = "The current position is " + p;
// automatically invokes p.toString()

```

If `x` is any object and you call

```
System.out.println(x);
```

then the `println` method simply calls `x.toString()` and prints the resulting string.

Another method of the `Object` class is `clone`.

```
Object clone()
```

creates a clone of the object. The Java runtime system allocates memory for the new instance and copies the memory allocated for the current object.

Design Hints for Inheritance

1. Place common operations and fields in the superclass.

This is why we put the `name` field into the `Person` class, rather than replicating it in `Employee` and `Student`.

2. Don't use protected fields.

Some programmers think it is a good idea to define most instance fields as `protected`, "just in case," so that subclasses can access these fields if they need to. However, the `protected` mechanism doesn't give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses `protected` instance fields, thereby breaking encapsulation. And in the Java programming language, all classes in the same package have access to `protected` fields, whether or not they are subclasses.

However, `protected` methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses. The `clone` method is a good example.

3. Use inheritance to model the "is-a" relationship.

Inheritance is a handy code-saver, and sometimes people overuse it. For example, suppose we need a `Contractor` class. Contractors have names and hire dates, but they do not have salaries. Instead, they are paid by the hour, and they do not stay around long enough to get a raise. There is the temptation to derive `Contractor` from `Employee` and add an `hourlyWage` field.

```

class Contractor extends Employee {
    . . .
    private double hourlyWage;
}

```

This is *not* a good idea, however, because now each contractor object has both a salary and hourly wage field. It will cause you no end of grief when you implement methods for printing paychecks or

tax forms. You will end up writing more code than you would have by not inheriting in the first place.

The contractor/employee relationship fails the "is-a" test. A contractor is not a special case of an employee.

4. *Don't use inheritance unless all inherited methods make sense.*

Suppose we want to write a Holiday class. Surely every holiday is a day, and days can be expressed as instances of the GregorianCalendar class, so we can use inheritance.

```
class Holiday extends GregorianCalendar { . . . }
```

Unfortunately, the set of holidays is not *closed* under the inherited operations. One of the public methods of GregorianCalendar is add. And add can turn holidays into nonholidays:

```
Holiday christmas;  
christmas.add(GregorianCalendar.DAY, 12);
```

Therefore, inheritance is not appropriate in this example.

4. *Use polymorphism, not type information.*

Whenever you find code of the form

```
if (x is of type 1)  
    action1(x);  
else if (x is of type 2)  
    action2(x);
```

think polymorphism.

Do *action1* and *action2* represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call

```
x.action();
```

and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.

Code using polymorphic methods or interface implementations is much easier to maintain and extend than code that uses multiple type tests.

6. *Don't overuse reflection.*

The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at run time. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications. Reflection is fragile—the compiler cannot help you find programming errors. Any errors are found at run time and result in exceptions.

Interfaces

The first, called an *interface*, is a way of describing *what* classes should do, without specifying *how* they should do it. A class can *implement* one or more interfaces. You can then use objects of these implementing classes anytime that conformance to the interface is required.

An interface is not a class but a set of *requirements* for classes that want to conform to the interface. Typically, the supplier of some service states: "If your class conforms to a particular interface, then I'll perform the service." Let's look at a concrete example. The sort method of the Arrays class promises to sort an array of objects, but under one condition: The objects must belong to classes that implement the Comparable interface.

Here is what the Comparable interface looks like:

```
public interface Comparable {  
    int compareTo(Object other);  
}
```

This means that any class that implements the Comparable interface is required to have a compareTo method, and the method must take an Object parameter and return an integer.

All methods of an interface are automatically public. For that reason, it is not necessary to supply the keyword public when declaring a method in an interface.

Of course, there is an additional requirement that the interface cannot spell out: When calling `x.compareTo(y)`, the `compareTo` method must actually be able to compare two objects and return an indication whether `x` or `y` is larger. The method is supposed to return a negative number if `x` is smaller than `y`, zero if they are equal, and a positive number otherwise.

This particular interface has a single method. Some interfaces have more than one method. As you will see later, interfaces can also define constants. What is more important, however, is what

interfaces *cannot* supply. Interfaces never have instance fields, and the methods are never implemented in the interface. Supplying instance fields and method implementations is the job of the classes that implement the interface. You can think of an interface as being similar to an abstract class with no instance fields. However, there are some differences between these two concepts—we will look at them later in some detail.

Now suppose we want to use the sort method of the Arrays class to sort an array of `Employee` objects. Then the `Employee` class must *implement* the `Comparable` interface. To make a class implement an interface, you have to carry out two steps:

1. You declare that your class intends to implement the given interface.
2. You supply definitions for all methods in the interface.

To declare that a class implements an interface, use the `implements` keyword:

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is a `compareTo` method that returns `-1` if the first employee's salary is less than the second employee's salary, `0` if they are equal, and `1` otherwise.

```
public int compareTo(Object otherObject) {
    Employee other = (Employee)otherObject;
    if (salary < other.salary) return -1;
    if (salary > other.salary) return 1;
    return 0;
}
```

But why can't the `Employee` class simply provide a `compareTo` method without implementing the `Comparable` interface?

The reason for interfaces is that the Java language is *strongly typed*. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the sort method, there will be statements like this:

```
if (a[i].compareTo(a[j]) > 0) {
    // rearrange a[i] and a[j]
    . . .
}
```

The compiler must know that `a[i]` actually has a `compareTo` method. If `a` is an array of `Comparable` objects, then the existence of the method is assured, because every class that implements the `Comparable` interface must supply the method.

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

```
public class EmployeeSortTest {
    public static void main(String[] args) {
        Employee[] staff = new Employee[3];
        staff[0] = new Employee("Harry Hacker", 35000);
        staff[1] = new Employee("Carl Cracker", 75000);
        staff[2] = new Employee("Tony Tester", 38000);
        Arrays.sort(staff);
        for (int i = 0; i < staff.length; i++){
            Employee e = staff[i];
            System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
        }
    }
}
```

```
class Employee implements Comparable{
    private String name;
    private double salary;
    public Employee(String n, double s){
        name = n;
        salary = s;
    }
    public String getName(){
        return name;
    }
}
```

```

    }
    public double getSalary(){
        return salary;
    }
    public void raiseSalary(double byPercent){
        double raise = salary * byPercent / 100;
        salary += raise;
    }
    public int compareTo(Object otherObject){
        Employee other = (Employee)otherObject;
        if (salary < other.salary) return -1;
        if (salary > other.salary) return 1;
        return 0;
    }
}

```

Properties of Interfaces

Interfaces are not classes. In particular, you can never use the new operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can't construct interface objects, you can still declare sinterface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .);
```

// OK provided Employee implements Comparable

Next, just as you use instanceof to check if an object is of a specific class, you can use instanceof to check if an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called Moveable.

```
public interface Moveable {
    void move(double x, double y);
}
```

Then, you could imagine an interface called Powered that extends it:

```
public interface Powered extends Moveable {
    double milesPerGallon();
}
```

Although you cannot put instance fields or static methods in an interface, you can supply constants in them. For example:

```
public interface Powered extends Moveable {
    double milesPerGallon();
    double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically public, fields are always public static final.

While each class can only have one superclass, classes can implement *multiple* interfaces. This gives you the maximum amount of flexibility in defining a class's behavior.

```
class Employee implements Cloneable, Comparable
```

Interfaces and Abstract Classes

If you read the section about abstract classes in Chapter 5, you may wonder why the designers of the Java programming language bothered with introducing the concept of interfaces. Why can't Comparable simply be an abstract class:

```
abstract class Comparable // why not? {
    public abstract int compareTo(Object other);
}
```

```
}
```

Then the Employee class would simply extend this abstract class and supply the compareTo method:

```
class Employee extends Comparable // why not? {  
    public int compareTo(Object other) { . . . }  
}
```

There is, unfortunately, a major problem with using an abstract base class to express a generic property. A class can only extend a single class. Suppose that the Employee class already extends a different class, say Person. Then it can't extend a second class.

```
class Employee extends Person, Comparable // ERROR
```

But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

Other programming languages, in particular C++, allow a class to have more than one superclass. This feature is called *multiple inheritance*. The designers of Java chose not to support multiple inheritance because it makes the language either very complex or less efficient. Instead, interfaces give most of the benefits of multiple inheritance while avoiding the complexities and inefficiencies.

Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you want to specify the action that should occur whenever a particular event happens. For example, you may want a particular action to occur when a button is clicked or a menu item is selected. However, since you have not yet seen how to implement user interfaces, we will consider a similar but simpler situation.

The javax.swing class contains a Timer class that is useful if you want to be notified whenever a time interval has elapsed. For example, if a part of your program contains a clock, then you can ask to be notified every second so that you can update the clock face.

When you construct a timer, you set the time interval, and you tell it what it should do whenever the time interval has elapsed.

How do you tell the timer what it should do? In many programming languages, you supply the name of a function that the timer should call periodically. However, the classes in the Java standard library take an object-oriented approach. You pass an object of some class. The timer then calls one of the methods on that object. Passing an object is more flexible than passing a function because the object can carry additional information.

Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the ActionListener interface of the java.awt.event package. Here is that interface:

```
public interface ActionListener {  
    void actionPerformed(ActionEvent event);  
}
```

The timer calls the actionPerformed method when the time interval has expired.

Suppose you want to print a message "At the tone, the time is . . .," followed by a beep, once every ten seconds. You need to define a class that implements the ActionListener interface. Then place whatever statements you want to have executed inside the actionPerformed method.

```
class TimePrinter implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        Date now = new Date();  
        System.out.println("At the tone, the time is " + now);  
        Toolkit.getDefaultToolkit().beep();  
    }  
}
```

Note the ActionEvent parameter of the actionPerformed method. This parameter gives information about the event, such as the source object that generated it. However, detail information about the event is not important in this program, and you can safely ignore the parameter. Next, you construct an object of this class and pass it to the Timer constructor.

```
ActionListener listener = new TimePrinter();  
Timer t = new Timer(10000, listener);
```

The first parameter of the Timer constructor is the time interval that must elapse between notifications, measured in milliseconds. We want to be notified every ten seconds. The second parameter is the listener object. Finally, you start the timer.

```
t.start();
```

Every ten seconds, a message like

At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
is displayed, followed by a beep.

Inner Classes

An *inner class* is a class that is defined inside another class. Why would you want to do that?

There are four reasons:

1. An object of an inner class can access the implementation of the object that created it—including data that would otherwise be private.
2. Inner classes can be hidden from other classes in the same package.
3. *Anonymous* inner classes are handy when you want to define callbacks on the fly.
4. Inner classes are very convenient when you are writing event-driven programs.

You will soon see examples that demonstrate the first three benefits.

Using an Inner Class to Access Object State

The syntax for inner classes is somewhat complex. For that reason, we will use a simple but somewhat artificial example to demonstrate the use of inner classes. We will write a program in which a timer controls a bank account. The timer's action listener object adds interest to the account once per second. However, we don't want to use public methods (such as deposit or withdraw) to manipulate the bank balance because anyone could call those public methods to modify the balance for other purposes. Instead, we will use an *inner class* whose methods can manipulate the bank balance directly.

Here is the outline of the BankAccount class:

```
class BankAccount {
    public BankAccount(double initialBalance) { . . . }
    public void start(double rate) { . . . }
    private double balance;
    private class InterestAdder implements ActionListener{
        // an inner class
        . . .
    }
}
```

Note the InterestAdder class that is located inside the BankAccount class. This does *not* mean that every BankAccount has an InterestAdder instance field. Of course, we will construct objects of the inner class, but those objects aren't instance fields of the outer class. Instead, they will be local to the methods of the outer class.

The InterestAdder class is a *private inner class* inside BankAccount. This is a safety mechanism—since only BankAccount methods can generate InterestAdder objects, we don't have to worry about breaking encapsulation. Only inner classes can be private. Regular classes always have either package or public visibility.

The InterestAdder class has a constructor which sets the interest rate that should be applied at each step. Since this inner class implements the ActionListener interface, it also has an actionPerformed method. That method actually increments the account balance. Here is the inner class in more detail:

```
class BankAccount {
    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }
    . . .
    private double balance;
    private class InterestAdder implements ActionListener {
        private double rate;
```

```

    public InterestAdder(double aRate) {
        rate = aRate;
    }
    public void actionPerformed(ActionEvent event) { . . . }
}

```

The `start` method of the `BankAccount` class constructs an `InterestAdder` object for the given interest rate, makes it the action listener for a timer, and starts the timer.

```

public void start(double rate) {
    ActionListener adder = new InterestAdder(rate);
    Timer t = new Timer(1000, adder);
    t.start();
}

```

As a result, the `actionPerformed` method of the `InterestAdder` class will be called once per second. Now let's look inside this method more closely:

```

public void actionPerformed(ActionEvent event) {
    double interest = balance * rate / 100;
    balance += interest;
    NumberFormat formatter = NumberFormat.getCurrencyInstance();
    System.out.println("balance=" + formatter.format(balance));
}

```

The name `rate` refers to the instance field of the `InterestAdder` class, which is not surprising. However, there is no `balance` field in the `InterestAdder` class. Instead, `balance` refers to the field of the `BankAccount` object that created this `InterestAdder`. This is quite innovative. Traditionally, a method could refer to the data fields of the object invoking the method. An inner class method gets to access both its own data fields *and* those of the outer object creating it. For this to work, of course, an object of an inner class always gets an implicit reference to the object that created it. This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object *outer*. Then, the `actionPerformed` method is equivalent to the following:

```

public void actionPerformed(ActionEvent event) {
    double interest = outer.balance * this.rate / 100;
    // "outer" isn't the actual name
    outer.balance += interest;
    NumberFormat formatter = NumberFormat.getCurrencyInstance();
    System.out.println("balance=" + formatter.format(outer.balance));
}

```

The outer class reference is set in the constructor. That is, the compiler adds a parameter to the constructor, generating code like this:

```

public InterestAdder(BankAccount account, double aRate) {
    outer = account;
    // automatically generated code
    rate = aRate;
}

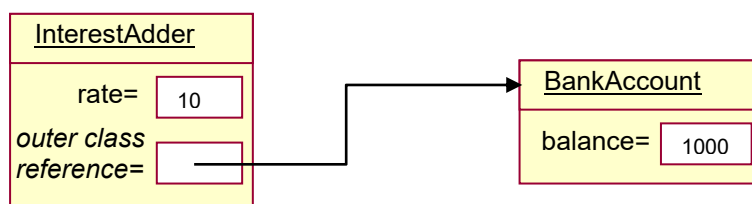
```

Again, please note, *outer* is not a Java keyword. We just use it to illustrate the mechanism involved in an inner class. When an `InterestAdder` object is constructed in the `start` method, the compiler passes the `this` reference to the current bank account into the constructor:

```

ActionListener adder = new InterestAdder(this, rate);
// automatically generated code

```



```

import java.awt.event.*;
import java.text.*;
import javax.swing.*;
public class InnerClassTest {
    public static void main(String[] args) {
        // construct a bank account with initial balance of $10,000
        BankAccount account = new BankAccount(10000);
        // start accumulating interest at 10%
        account.start(10);
        // keep program running until user selects "Ok"
        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}
class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }
    public void start(double rate) {
        ActionListener adder = new InterestAdder(rate);
        Timer t = new Timer(1000, adder);
        t.start();
    }
    private class InterestAdder implements ActionListener {
        private double rate;
        public InterestAdder(double aRate) {
            rate = aRate;
        }
        public void actionPerformed(ActionEvent event) {
            // update interest
            double interest = balance * rate / 100;
            balance += interest;
            // print out current balance
            NumberFormat formatter = NumberFormat.getCurrencyInstance();
            System.out.println("balance=" + formatter.format(balance));
        }
    }
}

```

Special Syntax Rules for Inner Classes

In the preceding section, we explained the outer class reference of an inner class by calling it *outer*. Actually, the proper syntax for the outer reference is a bit more complex. The expression *OuterClass.this* denotes the outer class reference. For example, you can write the `actionPerformed` method of the `InterestAdder` inner class as

```

public void actionPerformed(ActionEvent event) {
    double interest = BankAccount.this.balance * this.rate / 100;
    BankAccount.this.balance += interest;
    . . .
}

```

Conversely, you can write the inner object constructor more explicitly, using the syntax:
outerObject.new InnerClass(construction parameters)

For example,

```

ActionListener adder = this.new InterestAdder(rate);

```

Here, the outer class reference of the newly constructed InterestAdder object is set to the this reference of the method that creates the inner class object. This is the most common case. As always, the this. qualifier is redundant. However, it is also possible to set the outer class reference to another object by explicitly naming it. For example, if InterestAdder was a public inner class, you could construct an InterestAdder for any bank account:

```
BankAccount mySavings = new BankAccount(10000);
```

```
BankAccount.InterestAdder adder = mySavings.new InterestAdder(10);
```

Note that you refer to an inner class as

OuterClass.InnerClass

when it occurs outside the scope of the outer class. For example, if InterestAdder had been a public class, you could have referred to it as BankAccount.InterestAdder elsewhere in your program.

Local Inner Classes

If you look carefully at the code of the BankAccount example, you will find that you need the name of the type InterestAdder only once: when you create an object of that type in the start method. When you have a situation like this, you can define the class *locally in a single method*.

```
public void start(double rate) {
    class InterestAdder implements ActionListener {
        private double rate;
        public InterestAdder(double aRate) {
            rate = aRate;
        }
        public void actionPerformed(ActionEvent event) {
            double interest = balance * rate / 100;
            balance += interest;
            NumberFormat formatter = NumberFormat.getCurrencyInstance();
            System.out.println("balance=" + formatter.format(balance));
        }
    }
    ActionListener adder = new InterestAdder(rate);
    Timer t = new Timer(1000, adder);
    t.start();
}
```

Local classes are never declared with an access specifier (that is, public or private). Their scope is always restricted to the block in which they are declared.

Local classes have a great advantage—they are completely hidden from the outside world—not even other code in the BankAccount class can access them. No method except start has any knowledge of the InterestAdder class.

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes, they can even access local variables! However, those local variables must be declared final.

Here is a typical example.

```
public void start(final double rate) {
    class InterestAdder implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            double interest = balance * rate / 100;
            balance += interest;
            NumberFormat formatter = NumberFormat.getCurrencyInstance();
            System.out.println("balance=" + formatter.format(balance));
        }
    }
    ActionListener adder = new InterestAdder();
    Timer t = new Timer(1000, adder);
    t.start();
}
```

Note that the `InterestAdder` class no longer needs to store a rate instance variable. It simply refers to the parameter variable of the method that contains the class definition.

Maybe this should not be so surprising. The line

```
double interest = balance * rate / 100;
```

is, after all, ultimately inside the `start` method, so why shouldn't it have access to the value of the rate variable?

To see why there is a subtle issue here, let's consider the flow of control more closely.

1. The `start` method is called.
2. The object variable `adder` is initialized via a call to the constructor of the inner class `InterestAdder`. The `adder` reference is passed to the `Timer` constructor, the timer is started, and the `start` method
3. exits. At this point, the rate parameter variable of the `start` method no longer exists.
4. A second later, the `actionPerformed` method calls `double interest = balance * rate / 100;`.

For the code in the `actionPerformed` method to work, the `InterestAdder` class must have made a copy of the rate field before it went away as a local variable of the `start` method. That is indeed exactly what happens. In our example, the compiler synthesizes the name `BankAccount$1$InterestAdder` for the local inner class. If you use the `ReflectionTest` program again to spy on the `BankAccount$1$InterestAdder` class, you get the following output:

```
class BankAccount$1$InterestAdder {  
    BankAccount$1$InterestAdder(BankAccount, double);  
    public void actionPerformed(java.awt.event.ActionEvent);  
    private final double val$rate;  
    private final BankAccount this$0;  
}
```

Note the extra `double` parameter to the constructor and the `val$rate` instance variable. When an object is created, the value `rate` is passed into the constructor and stored in the `val$rate` field. This sounds like an extraordinary amount of trouble for the implementors of the compiler. The compiler must detect access of local variables, make matching data fields for each one of them, and copy the local variables into the constructor so that the data fields can be initialized as copies of them. From the programmer's point of view, however, local variable access is quite pleasant. It makes your inner classes simpler by reducing the instance fields that you need to program explicitly. As we already mentioned, the methods of a local class can refer only to local variables that are declared `final`.

For that reason, the rate parameter was declared `final` in our example. A local variable that is declared `final` cannot be modified. Thus, it is guaranteed that the local variable and the copy that is made inside the local class always have the same value.

Anonymous inner classes

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called *anonymous inner class*.

```
public void start(final double rate) {  
    ActionListener adder = new  
    ActionListener() {  
        public void actionPerformed(ActionEvent event) {  
            double interest = balance * rate / 100;  
            balance += interest;  
            NumberFormat formatter  
            = NumberFormat.getCurrencyInstance();  
            System.out.println("balance=" +  
            + formatter.format(balance));  
        }  
    };  
    Timer t = new Timer(1000, adder);  
    t.start();  
}
```

This is a very cryptic syntax indeed. What it means is:

Create a new object of a class that implements the ActionListener interface, where the required method actionPerformed is the one defined inside the braces { }.

Any parameters used to construct the object are given inside the parentheses () following the supertype name. In general, the syntax is

```
new SuperType(construction parameters) {  
    inner class methods and data  
}
```

Here, *SuperType* can be an interface, such as ActionListener; then, the inner class *implements* that interface. Or, *SuperType* can be a class; then, the inner class *extends* that class.

An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name. Instead, the construction parameters are given to the *superclass* constructor. In particular, whenever an inner class implements an interface, it cannot have any construction parameters. Nevertheless, you must supply a set of parentheses as in:

```
new InterfaceType() { methods and data }
```

You have to look very carefully to see the difference between the construction of a new object of a class and the construction of an object of an anonymous inner class extending that class. If the closing parenthesis of the construction parameter list is followed by an opening brace, then an anonymous inner class is being defined.

```
Person queen = new Person("Mary");  
// a Person object  
Person count = new Person("Dracula") { ... };  
// an object of an inner class extending Person
```

Example of anonymous inner class:

```
import java.awt.event.*;  
import java.text.*;  
import javax.swing.*;  
public class AnonymousInnerClassTest 6. {  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount(10000);  
        // start accumulating interest at 10%  
        account.start(10);  
        // keep program running until user selects "Ok"  
        JOptionPane.showMessageDialog(null, "Quit program?");  
        System.exit(0);  
    }  
}  
class BankAccount {  
    private double balance;  
    public BankAccount(double initialBalance) {  
        balance = initialBalance;  
    }  
    public void start(final double rate) {  
        ActionListener adder = new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                // update interest  
                double interest = balance * rate / 100;  
                balance += interest;  
                // print out current balance  
                NumberFormat formatter = NumberFormat.getCurrencyInstance();  
                System.out.println("balance=" + formatter.format(balance));  
            }  
        };  
        Timer t = new Timer(1000, adder);  
        t.start();  
    }  
}
```


Graphics Programming

Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a *frame* in Java. The AWT library has a class, called `Frame`, for this top level. The Swing version of this class is called `JFrame` that extends the `Frame` class. The `JFrame` is one of the few Swing components that is not painted on a canvas. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user's windowing system, not by Swing.

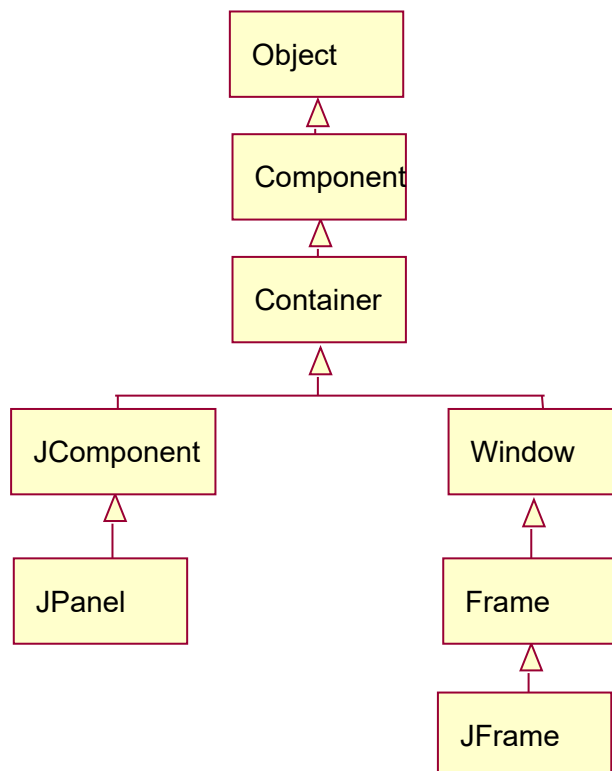
Frames are examples of *containers*. This means that a frame can contain other user interface components such as buttons and text fields. In this section, we want to go over the most common methods for working with a Swing `JFrame`.

```
import javax.swing.*;

public class SimpleFrameTest {
    public static void main(String[] args) {
        SimpleFrame frame = new SimpleFrame();
        // we define what should happen when the user closes this frame.
        // For this particular program, we want the program to exit.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

class SimpleFrame extends JFrame {
    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
    public SimpleFrame() {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```

The Swing classes are placed in the `javax.swing` package.



By default, a frame has a rather useless size of 0 x 0 pixels. We define a subclass SimpleFrame whose constructor sets the size to 300 x 200 pixels. In the main method of the SimpleFrameTest class, we start out by constructing a SimpleFrame object.

Simply constructing a frame does not automatically display it. Frames start their life invisible. That gives the programmer the chance to add components into the frame before showing it for the first time. To show the frame, the main method calls the show method of the frame.

Afterwards, the main method exits. Note that exiting main does not terminate the program, just the main thread. Showing the frame activates a user interface thread that keeps the program alive.

Frame Positioning

The JFrame class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of JFrame. Among the most important methods are the following ones:

- ❑ The dispose method that closes the window down and reclaims any system resources used in creating it;
- ❑ The setIconImage method, which takes an Image object to use as the icon when the window is minimized (often called *iconized* in Java terminology);
- ❑ The setTitle method for changing the text in the title bar;
- ❑ The setResizable method, which takes a boolean to determine if a frame will be resizable by the user.

the Component class (which is the ancestor of all GUI objects) and the Window class (which is the Frame's class parent) are where you need to look to find the methods to resize and reshape frames. For example, the show method that you use to display the frame lives in the Window class. As another example, the setLocation method in the Component class is one way to reposition a component. If you make the call setLocation(x, y)

The top-left corner is located x pixels across and y pixels down, where (0, 0) is the top-left corner of the screen. Similarly, the setBounds method in Component lets you resize and relocate a component (in particular, a JFrame) in one step, as

```
setBounds(x, y, width, height)
```

To give you an idea of what you can do with a window, we end this section by showing you a sample program that positions one of our closable frames so that:

- ❑ Its area is one-fourth that of the whole screen;
- ❑ It is centered in the middle of the screen.

We also supply an icon. Because the representation of images is also system dependent, we again use the toolkit to load an image. Then, we set the image as the icon for the frame.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CenteredFrameTest {
    public static void main(String[] args) {
        CenteredFrame frame = new CenteredFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}
class CenteredFrame extends JFrame {
    public CenteredFrame() {
        // get screen dimensions
        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth = screenSize.width;
        // center frame in screen
        setSize(screenWidth / 2, screenHeight / 2);
        setLocation(screenWidth / 4, screenHeight / 4);
        // set frame icon and title
```

```

    Image img = kit.getImage("icon.gif");
    setIconImage(img);
    setTitle("CenteredFrame");
}
}

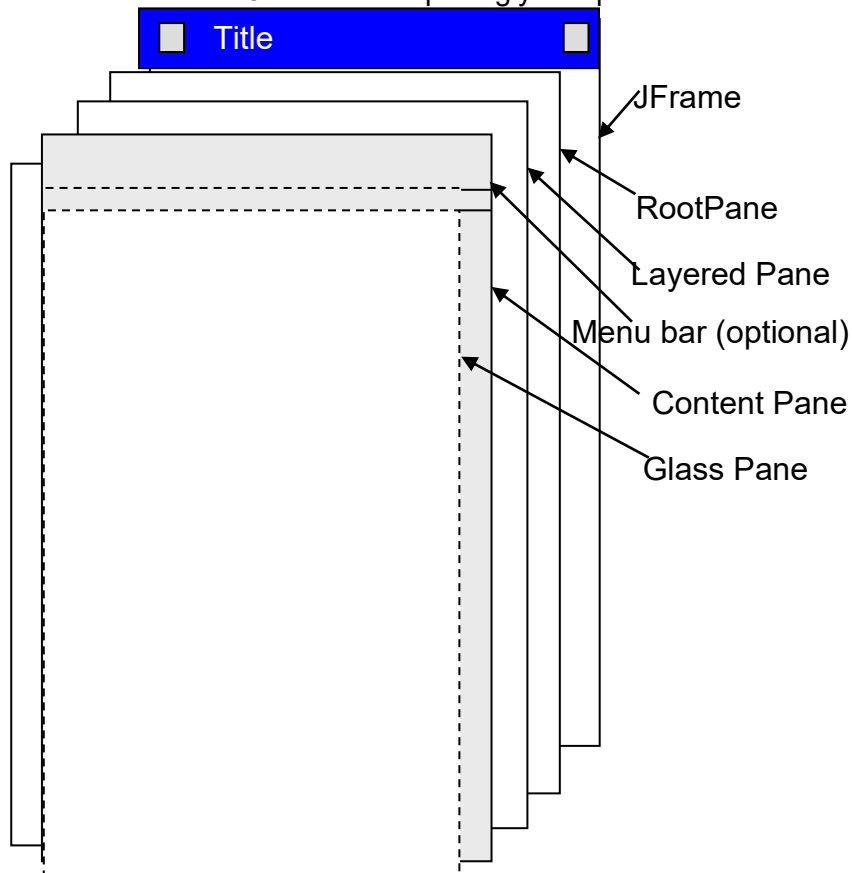
```

Displaying Information in a Panel

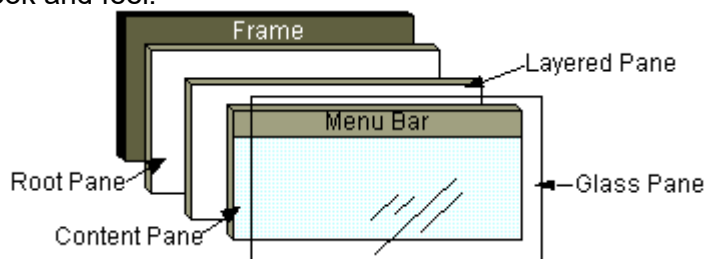
In this section, we'll show you how to display information inside a frame. For example, rather than displaying "Not a Hello, World program" in text mode in a console window, we will display the message in a frame.

It would be possible to draw the message string directly onto a frame, but that is not considered good programming practice. In Java, frames are really designed to be containers for *components* such as a menu bar and other user interface elements. You normally draw on another component, called a *panel*, which you add to the frame.

The structure of a JFrame is surprisingly complex.



Four panes are layered in a JFrame. The root pane, layered pane, and glass pane are of no interest to us; they are required to organize the menu bar and content pane and to implement the look and feel.



As the preceding figure shows, a root pane has four parts:

The glass pane

Hidden, by default. If you make the glass pane visible, then it's like a sheet of glass over all the other parts of the root pane. It's completely transparent unless you implement the glass

pane's `paintComponent` method so that it does something, and it intercepts input events for the root pane.

The layered pane

Serves to position its contents, which consist of the content pane and the optional menu bar. Can also hold other components in a specified Z order.

The content pane

The container of the root pane's visible components, excluding the menu bar.

The optional menu bar

The home for the root pane's container's menus. If the container has a menu bar, you generally use the container's `setJMenuBar` method to put the menu bar in the appropriate place.

The part that most concerns Swing programmers is the *content pane*. When designing a frame, you add components into the content pane, using code such as the following:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

Alternatively, if you just want to show a single Swing component in the frame, then you can set the content pane to that component:

```
JComponent c = . . .;
frame.setContentPane(c);
```

In our case, we want to add a single *panel* to the content pane onto which we will draw our message.

Panels are implemented by the `JPanel` class. They are user interface elements with two useful properties:

- ❑ They have a surface onto which you can draw;
- ❑ They themselves are containers.

Thus, they can hold other user interface components such as buttons, sliders, and so on. However, there is no sense in adding a plain `JPanel` to the content pane—it doesn't do anything interesting. To make it more interesting, you use inheritance to create a new class and then override or add methods to get the extra functionality you want.

In particular, to draw on a panel, you will need to:

- ❑ Define a class that extends `JPanel`;
- ❑ Override the `paintComponent` method in that class.

The `paintComponent` method is actually in `JComponent`—the parent class for all nonwindow Swing components. It takes one parameter of type `Graphics`. A `Graphics` object remembers a collection of settings for drawing images and text, such as the font you set or the current color. All drawing in Java must go through a `Graphics` object. It has methods that draw patterns, images, and text.

Here's how to make a panel onto which you can draw:

```
class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
        . . . // code for drawing will go here
    }
}
```

Each time a window needs to be redrawn, no matter what the reason, the event handler notifies the component. This causes the `paintComponent` methods of all components to be executed. Never call the `paintComponent` method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process. What sorts of actions trigger this automatic response? For example, painting occurs because the user increased the size of the window or minimized and then restored the window. If the user popped up another window and it covered an existing window and then made the overlaid window disappear, the application window that was covered is now corrupted and will need to be repainted. (The graphics system does not save the pixels underneath.) And, of course, when the window is displayed for the first time, it needs to process the code that specifies how and where it should draw the initial elements.

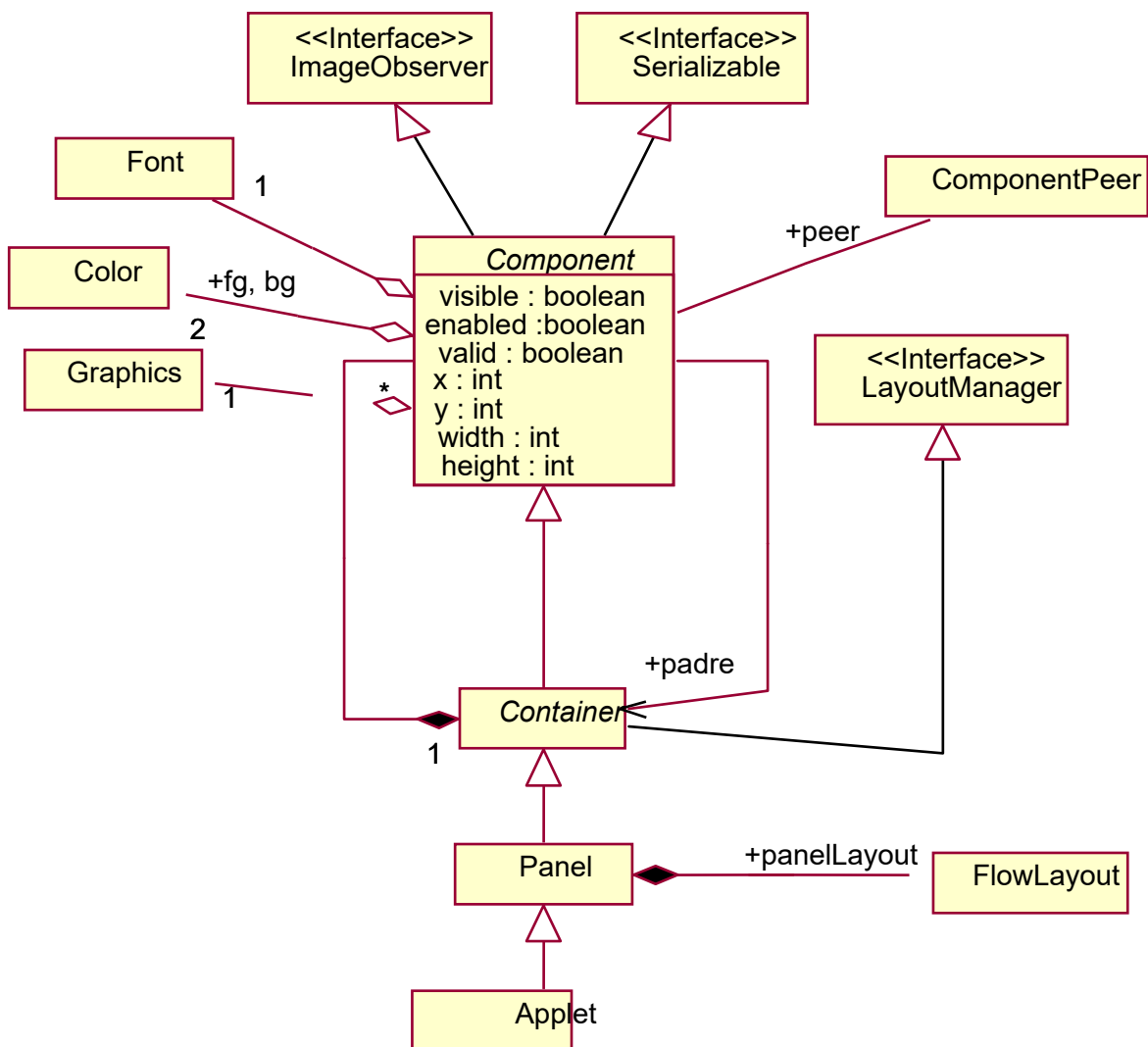
As you saw in the code fragment above, the `paintComponent` method takes a single parameter of type `Graphics`. Measurement on a `Graphics` object for screen display is done in pixels. The (0, 0) coordinate denotes the top-left corner of the component on whose surface you are drawing. Displaying text (usually called *rendering text*) is considered a special kind of drawing. The `Graphics` class has a `drawString` method that has the following syntax:

```
g.drawString(text, x, y)
```

In our case, we want to draw the string "Hello, World !" in our original window, roughly one-quarter of the way across and halfway down. Although we don't yet know how to measure the size of the string, we'll start the string at coordinates (75, 100). This means the first character in the string will start at a position 75 pixels to the right and 100 pixels down. (Actually, it is the baseline for the text that is 100 pixels down—see below for more on how text is measured.) Thus, our `paintComponent` method looks like this:

```
class HelloWorldPanel extends JPanel {
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;
    public void paintComponent(Graphics g) {
        . . . // see below
        g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
    }
}
```

However, this `paintComponent` method is not complete. The `HelloWorldPanel` class extends the `JPanel` class, which has its own idea how to paint the panel, namely, to fill it with the background color.



To make sure that the superclass does its part of the job, we must call `super.paintComponent` before doing any painting on our own.

```
class HelloWorldPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        . . . // code for drawing will go here
    }
}
```

```

}

The complete code:

import javax.swing.*;
import java.awt.*;

public class HelloWorld{
    public static void main(String[] args){
        HelloWorldFrame frame = new HelloWorldFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

/**
A frame that contains a message panel
*/

class HelloWorldFrame extends JFrame {
    public static final int DEFAULT WIDTH = 300;

```

```

public static final int DEFAULT_HEIGHT = 200;
public HelloWorldFrame(){
    setTitle("NotHelloWorld");
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    // add panel to frame
    HelloWorldPanel panel = new HelloWorldPanel();
    Container contentPane = getContentPane();
    contentPane.add(panel);
}
}
/**
A panel that displays a message.
*/
class HelloWorldPanel extends JPanel{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;
    public void paintComponent(Graphics g){
        super.paintComponent(g);
        g.drawString("Hello, World !", MESSAGE_X, MESSAGE_Y);
    }
}

```

```

java.awt.Component
    boolean isVisible()
    void setVisible(boolean b)
    boolean isShowing()
    boolean isEnabled()
    Font getFont()
    void setEnabled(boolean b)
    Point getLocation()
    Point getLocationOnScreen()
    Graphics getGraphics()
    void setLocation(int x, int y)
    void setLocation(Point p)
    void setFont(Font f)
    void setBackground(Color c)
    void setForeground(Color c)
    Dimension getSize
    void setSize(int w, int h)
    void setSize(Dimension d)
java.awt.Container
    Component add(Component c)
    Component add(String name, Component c)
    Component getComponentAt(int x, int y)
    Component getComponentAt(Point p)
    Component[] getComponents()
    LayoutManager getLayout()
    void setLayout(LayoutManager mgr)
    void paint(Graphics g)
    void print(Graphics g)
java.awt.Window
    void toFront()
    void toBack()
    Toolkit getToolkit()
    void show()
java.awt.Frame
    void setResizable(boolean b)
    void setTitle(String s)

```

2D Shapes

To draw shapes in the Java 2D library, you need to obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. If you use a version of the SDK that is Java 2D enabled, methods such as `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    . . .
}
```

The Java 2D library organizes geometric shapes in an object-oriented fashion. In particular, there are classes to represent lines, rectangles and ellipses:

1. `Line2D`
2. `Rectangle2D`
3. `Ellipse2D`

These classes all implement the `Shape` interface.

To draw a shape, you first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class. For example,

```
Rectangle2D rect = . . .;
g2.draw(rect);
```

the Java 2D shapes use floating-point coordinates. In many cases, that is a great convenience because it allows you to specify your shapes in coordinates that are meaningful to you (such as millimeters or inches) and then translate to pixels. The Java 2D library uses single-precision float quantities for many of its internal floating-point calculations.

Because the suffixes and casts are a bit of a pain, the designers of the 2D library decided to supply *two versions* of each shape class: one with float coordinates for frugal programmers, and one with double coordinates for the lazy ones. (In this book, we'll fall into the second camp and use double coordinates whenever we can.)

The library designers chose a curious, and initially confusing, method for packaging these choices. Consider the `Rectangle2D` class. This is an abstract class with two concrete subclasses, which are also static inner classes:

```
Rectangle2D.Float
Rectangle2D.Double
```

`Rectangle2D` and `Ellipse2D` objects are simple to construct. You need to specify:

- The x- and y-coordinates of the top-left corner;
- The width and height.

For ellipses, these refer to the bounding rectangle. For example,

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

constructs an ellipse that is bounded by a rectangle with the top-left corner at (150, 200), width 100, and height 50.

However, sometimes you don't have the top-left corner readily available. It is quite common to have two diagonal corner points of a rectangle, but perhaps they aren't the top-left and bottom-right corners. You can't simply construct a rectangle as

```
Rectangle2D rect = new Rectangle2D.Double(px, py,
qx - px, qy - py); // Error
```

If `p` isn't the top-left corner, one or both of the coordinate differences will be negative and the rectangle will come out empty. In that case, first create a blank rectangle and use the `setFrameFromDiagonal` method.

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Or, even better, if you know the corner points as `Point2D` objects `p` and `q`,

```
rect.setFrameFromDiagonal(p, q);
```

When constructing an ellipse, you usually know the center, width, and height, and not the corner points of the bounding rectangle (which don't even lie on the ellipse). There is a `setFrameFromCenter` method that uses the center point, but it still requires one of the four corner points. Thus, you will usually end up constructing an ellipse as follows:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2,
```

```
centerY - height / 2, width, height);
```

To construct a line, you supply the start and end points, either as Point2D objects or as pairs of numbers:

```
Line2D line = new Line2D.Double(start, end);
```

or

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

The following program draws

A rectangle;

The ellipse that is enclosed in the rectangle;

A diagonal of the rectangle;

A circle that has the same center as the rectangle.

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class DrawTest{
    public static void main(String[] args) {
        DrawFrame frame = new DrawFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

/**
 * A frame that contains a panel with drawings
 */
class DrawFrame extends JFrame {
    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 400;
    public DrawFrame() {
        setTitle("DrawTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        // add panel to frame
        DrawPanel panel = new DrawPanel();
        Container contentPane = getContentPane();
        contentPane.add(panel);
    }
}

/**
 * A panel that displays rectangles and ellipses.
 */
class DrawPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        // draw a rectangle
        double leftX = 100;
        double topY = 100;
        double width = 200;
        double height = 150;
        Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width,
            height);
        g2.draw(rect);
        // draw the enclosed ellipse
        Ellipse2D ellipse = new Ellipse2D.Double();
        ellipse.setFrame(rect);
        g2.draw(ellipse);
        // draw a diagonal line
```

```

        g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY +
        height));
        // draw a circle with the same center
        double centerX = rect.getCenterX();
        double centerY = rect.getCenterY();
        double radius = 150;
        Ellipse2D circle = new Ellipse2D.Double();
        circle.setFrameFromCenter(centerX, centerY, centerX + radius,
        centerY + radius);
        g2.draw(circle);
    }
}

```

Colors

The `setPaint` method of the `Graphics2D` class lets you select a color that is used for all subsequent drawing operations on the graphics context or component. To draw in multiple colors, you select a color, draw, then select another color, and draw again. You define colors with the `Color` class. The `java.awt.Color` class offers predefined constants for the 13 standard colors listed

```

BLACK GREEN RED BLUE LIGHT_GRAY WHITE CYAN MAGENTA YELLOW DARK_GRAY
ORANGE GRAY PINK

```

For example,

```

g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);

```

You can specify a custom color by creating a `Color` object by its red, green, and blue components. Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the `Color` constructor like this:

```

Color(int redness, int greenness, int blueness)

```

Here is an example of setting a custom color:

```

g.setPaint(new Color(0, 128, 128)); // a dull blue-green
g.drawString("Welcome!", 75, 125);

```

To set the *background color*, you use the `setBackground` method of the `Component` class, an ancestor of `JPanel`. In fact, you should set the background before displaying the frame for the first time.

```

MyPanel p = new MyPanel();
p.setBackground(Color.WHITE);
contentPane.add(p);

```

There is also a `setForeground` method. It specifies the default color that is used for drawing on the component.

Filling Shapes

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color (or, more generally, the current paint setting). Simply call `fill` instead of `draw`:

```

Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red color

```

The complete example:

```

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class FillTest {
    public static void main(String[] args) {
        FillFrame frame = new FillFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

```

```

}
/**
A frame that contains a panel with drawings
*/
class FillFrame extends JFrame{
    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 400;
    public FillFrame() {
        setTitle("FillTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        // add panel to frame
        FillPanel panel = new FillPanel();
        Container contentPane = getContentPane();
        contentPane.add(panel);
    }
}
/**
A panel that displays filled rectangles and ellipses
*/
class FillPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        // draw a rectangle
        double leftX = 100;
        double topY = 100;
        double width = 200;
        double height = 150;
        Rectangle2D rect = new Rectangle2D.Double(leftX, topY, 54. width,
        height);
        g2.setPaint(Color.RED);
        g2.fill(rect);
        // draw the enclosed ellipse
        Ellipse2D ellipse = new Ellipse2D.Double();
        ellipse setFrame(rect);
        g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
        g2.fill(ellipse);
    }
}

```

Text and Fonts

The "Not a Hello, World" program at the beginning of this chapter displayed a string in the default font. Often, you want to show text in a different font. You specify a font by its *font face name* (or font name for short). A font face name is composed of a *font family name*, such as "Helvetica," and an optional suffix such as "Bold."

For example, the font faces "Helvetica" and "Helvetica Bold" are both considered to be part of the family named "Helvetica."

To find out which fonts are available on a particular computer, call the `getAvailableFontFamilyNames`

method of the `GraphicsEnvironment` class. The method returns an array of strings that contains the names of all available fonts. To obtain an instance of the `GraphicsEnvironment` class that describes the graphics environment of the user's system, use the static `getLocalGraphicsEnvironment` method. Thus, the following program gives you a printout of the names of all fonts on your system:

```

import java.awt.*;
public class ListFonts {

```

```

    public static void main(String[] args) {
        String[] fontNames = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();
        for (int i = 0; i < fontNames.length; i++)
            System.out.println(fontNames[i]);
    }
}

```

On one system, the list starts out like this:

```

Abadi MT Condensed Light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic

```

To establish a common baseline, the AWT defines five *logical* font names:

```

SansSerif
Serif
Monospaced
Dialog
DialogInput

```

These font names are always mapped to fonts that actually exist on the client machine. For example, on a Windows system, SansSerif is mapped to Arial.

To draw characters in a font, you must first create an object of the class Font. You specify the font name, the font style, and the point size. Here is an example of how you construct a Font object:

```
Font helvb14 = new Font("Helvetica", Font.BOLD, 14);
```

The third argument is the point size.

You can use a logical font name in the place of a font face name in the Font constructor. You specify the style (plain, **bold**, *italic*, or ***bold italic***) by setting the second Font constructor argument to one of the following values:

```

Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC

```

Here is an example:

```

Font sansbold14 = new Font("SansSerif", Font.BOLD, 14)
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);

```

Event Handling

Basics of Event Handling

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running. Each program then decides what, if anything, to do in response to these events. In languages like Visual Basic, the correspondence between events and code is obvious. One writes code for each specific event of interest and places the code in what is usually called an *event procedure*. For example, a Visual Basic button named HelpButton would have a HelpButton_Click event procedure associated with it, and VB would activate the code in this procedure in response to that button being clicked. Each Visual Basic user interface component responds to a fixed set of events, and it is impossible to change the events to which a Visual Basic component responds. Java takes an approach somewhat between the Visual Basic approach and the raw C approach in terms of power and, therefore, in resulting complexity. Within the limits of the events that the AWT knows about, you completely control how events are transmitted from the *event sources* (such as buttons or scrollbars) to *event listeners*. You can designate *any* object to be an event listener—in practice, you pick an object that can conveniently carry out the desired response to the event. This *event delegation model* gives you much more flexibility than is possible with Visual Basic, where the listener is predetermined, but it requires more code and is more difficult to untangle (at least until you get used to it).

Event sources have methods that allow you to register event listeners with them. When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event.

As one would expect in an object-oriented language like Java, the information about the event is encapsulated in an *event* object. In Java, all event objects ultimately derive from the class `java.util.EventObject`. Of course, there are subclasses for each event type, such as `ActionEvent` and `WindowEvent`.

Different event sources can produce different kinds of events. For example, a button can send `ActionEvent` objects, whereas a window can send `WindowEvent` objects.

To sum up, here's an overview of how event handling in the AWT works.

1. A listener object is an instance of a class that implements a special interface called (naturally enough) a *listener interface*.
2. An event source is an object that can register listener objects and send them event objects.
3. The event source sends out event objects to all registered listeners when that event occurs.
4. The listener objects will then use the information in the event object to determine their reaction to the event.

You register the listener object with the source object by using lines of code that follow the model: `eventSourceObject.addEventListner(eventListenerObject);`

For example,

```
ActionListener listener = . . . ;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

Now the listener object is notified whenever an "action event" occurs in the button. For buttons, as you might expect, an action event is a button click.

Code like the above requires that the class to which the listener object belongs implements the appropriate interface (which in this case is the `ActionListener` interface). As with all interfaces in Java, implementing an interface means supplying methods with the right signatures. To implement the `ActionListener` interface, the listener class must have a method called `actionPerformed` that receives an `ActionEvent` object as a parameter.

```
class MyListener implements ActionListener {
    . . .
    public void actionPerformed(ActionEvent event) {
        // reaction to button click goes here
    }
}
```

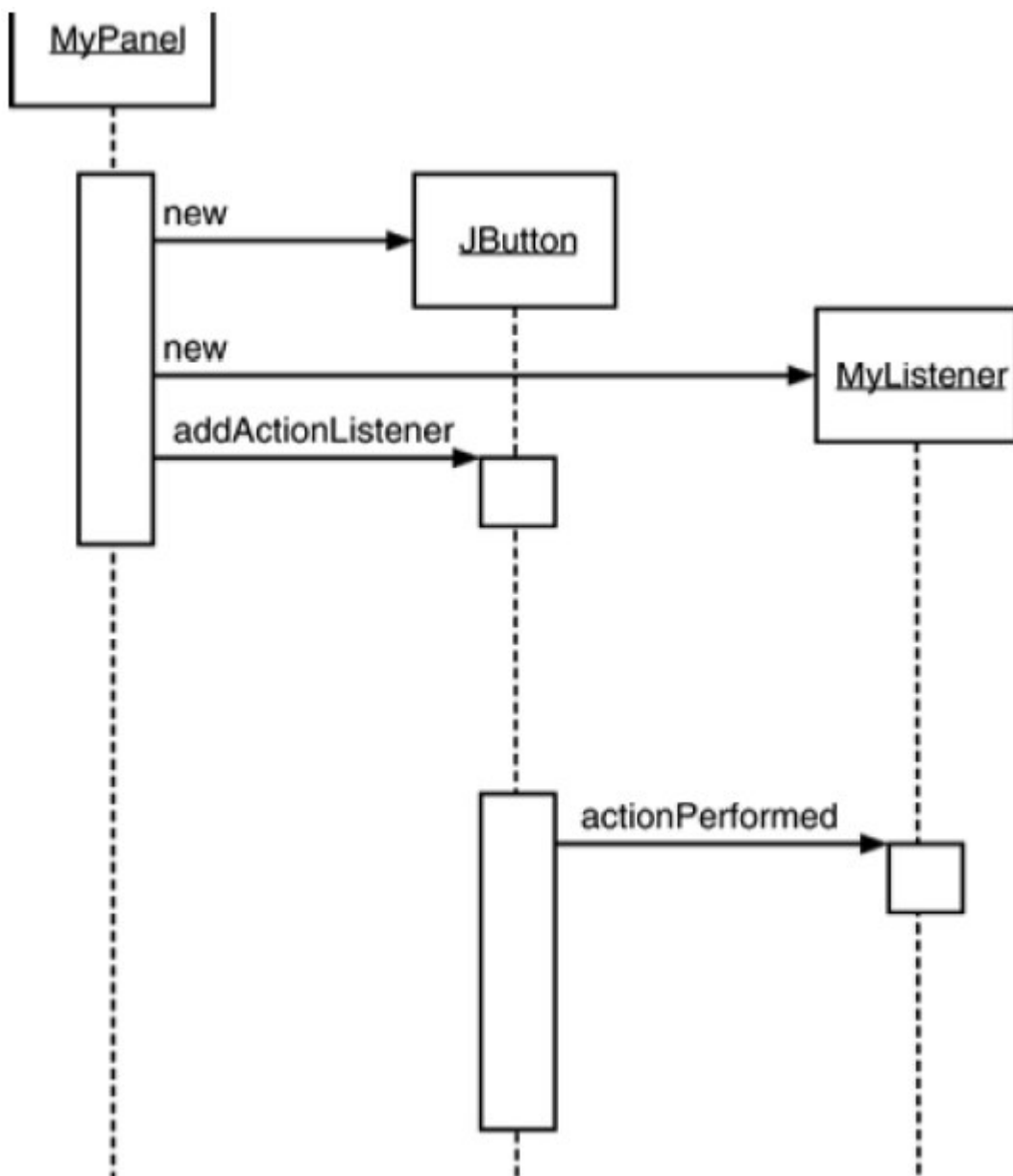
```

    . . .
}
}

```

Whenever the user clicks the button, the JButton object creates an(ActionEvent) object and calls listener.actionPerformed(event) passing that event object. It is possible for multiple objects to be added as listeners to an event source such as a button. In that case, the button calls the actionPerformed methods of all listeners whenever the user clicks the button.

The following figure shows the interaction between the event source, event listener, and event object.



Handling a button click

As a way of getting comfortable with the event delegation model, let's work through all details needed for the simple example of responding to a button click. For this example, we will want:

- ❑ A panel populated with three buttons;
- ❑ Three listener objects that are added as action listeners to the buttons.

With this scenario, each time a user clicks on any of the buttons on the panel, the associated listener object then receives an(ActionEvent) that indicates a button click. In our sample program, the listener object will then change the background color of the panel.

Before we can show you the program that listens to button clicks, we first need to explain how to create buttons and how to add them to a panel.

You create a button by specifying a label string, an icon, or both in the button constructor. Here are two examples:

```
JButton yellowButton = new JButton("Yellow");  
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Adding buttons to a panel occurs through a call to a method named (quite mnemonically) `add`. The `add` method takes as a parameter the specific component to be added to the container. For example,

```
class ButtonPanel extends JPanel {  
    public ButtonPanel() {  
        JButton yellowButton = new JButton("Yellow");  
        JButton blueButton = new JButton("Blue");  
        JButton redButton = new JButton("Red");  
        add(yellowButton);  
        add(blueButton);  
        add(redButton);  
    }  
}
```

With this scenario, each time a user clicks on any of the buttons on the panel, the associated listener object then receives an `ActionEvent` that indicates a button click. In our sample program, the listener object will then change the background color of the panel.

Before we can show you the program that listens to button clicks, we first need to explain how to create buttons and how to add them to a panel.

You create a button by specifying a label string, an icon, or both in the button constructor. Here are two examples:

```
JButton yellowButton = new JButton("Yellow");  
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Adding buttons to a panel occurs through a call to a method named (quite mnemonically) `add`. The `add` method takes as a parameter the specific component to be added to the container. For example,

```
class ButtonPanel extends JPanel {  
    public ButtonPanel() {  
        JButton yellowButton = new JButton("Yellow");  
        JButton blueButton = new JButton("Blue");  
        JButton redButton = new JButton("Red");  
        add(yellowButton);  
        add(blueButton);  
        add(redButton);  
    }  
}
```

Now that you know how to add buttons to a panel, you'll need to add code that lets the panel listen to these buttons. This requires classes that implement the `ActionListener` interface, which, as we just mentioned, has one method: `actionPerformed`, whose signature looks like this:

```
public void actionPerformed(ActionEvent event)
```

When a button is clicked, then we want to set the background color of the panel to a particular color. We store the desired color in our listener class.

```
class ColorAction implements ActionListener {  
    private Color backgroundColor;  
    public ColorAction(Color c) {  
        backgroundColor = c;  
    }  
    public void actionPerformed(ActionEvent event) {  
        // set panel background color  
        . . .  
    }  
}
```

```
}
```

We then construct one object for each color and set the objects as the button listeners.

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
```

```
ColorAction blueAction = new ColorAction(Color.BLUE);
```

```
ColorAction redAction = new ColorAction(Color.RED);
```

```
yellowButton.addActionListener(yellowAction);
```

```
blueButton.addActionListener(blueAction);
```

```
redButton.addActionListener(redAction);
```

For example, if a user clicks on the button marked "Yellow," then the `actionPerformed` method of the `yellowAction` object is called. Its `backgroundColor` instance field is set to `Color.YELLOW`, and it can now proceed to set the panel's background color.

There is just one remaining issue. The `ColorAction` object doesn't have access to the `panel` variable. You can solve this problem in two ways. You can store the `panel` in the `ColorAction` object and set it in the `ColorAction` constructor. Or, more conveniently, you can make `ColorAction` into an inner class of the `ButtonPanel` class. Then its methods can access the outer panel automatically.

We will follow the latter approach. Here is how you place the `ColorAction` class inside the `ButtonPanel` class.

```
class ButtonPanel extends JPanel {  
    . . .  
    private class ColorAction implements ActionListener {  
        private Color backgroundColor;  
        . . .  
        public void actionPerformed(ActionEvent event) {  
            setBackground(backgroundColor);  
            // i.e. outer.setBackground(...)  
        }  
    }  
}
```

Look closely at the `actionPerformed` method. The `ColorAction` class doesn't have a `setBackground` method. But the outer `ButtonPanel` class does. The methods are invoked on the `ButtonPanel` object that constructed the inner class objects. (Note again that *outer* is not a keyword in the Java programming language. We just use it as an intuitive symbol for the invisible outer class reference in the inner class object.)

This situation is very common. Event listener objects usually need to carry out some action that affects other objects. You can often strategically place the listener class inside the class whose state the listener should modify.

The complete program:

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
public class ButtonTest{  
    public static void main(String[] args){  
        ButtonFrame frame = new ButtonFrame();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.show();  
    }  
}  
/**  
A frame with a button panel  
*/  
class ButtonFrame extends JFrame {  
    public static final int DEFAULT_WIDTH = 300;  
    public static final int DEFAULT_HEIGHT = 200;  
    public ButtonFrame(){  
        setTitle("ButtonTest");  
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
        // add panel to frame
```

```

        ButtonPanel panel = new ButtonPanel();
        Container contentPane = getContentPane();
        contentPane.add(panel);
    }
}
/**
A panel with three buttons.
*/
class ButtonPanel extends JPanel{
    public ButtonPanel(){
        // create buttons
        JButton yellowButton = new JButton("Yellow");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");
        // add buttons to panel
        add(yellowButton);
        add(blueButton);
        add(redButton);
        // create button actions
        ColorAction yellowAction = new ColorAction(Color.YELLOW);
        ColorAction blueAction = new ColorAction(Color.BLUE);
        ColorAction redAction = new ColorAction(Color.RED);
        // associate actions with buttons
        yellowButton.addActionListener(yellowAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);
    }
}
/**
An action listener that sets the panel's background color.
*/
private class ColorAction implements ActionListener{
    private Color backgroundColor;
    public ColorAction(Color c){
        backgroundColor = c;
    }
    public void actionPerformed(ActionEvent event){
        setBackground(backgroundColor);
    }
}
}

```

Becoming Comfortable with Inner Classes

Some people dislike inner classes because they feel that a proliferation of classes and objects makes their programs slower. Let's have a look at that claim. First of all, you don't need a new class for every user interface component. In our example, all three buttons share the same listener class. Of course, each of them has a separate listener object. But these objects aren't large. They each contain a color value and a reference to the panel. And the traditional solution, with if . . . else statements, also references the same color objects that the action listeners store, just as local variables and not as instance variables.

We believe the time has come to get used to inner classes. We recommend that you use dedicated inner classes for event handlers rather than turning existing classes into listeners. We think that even anonymous inner classes have their place.

Here is a good example how anonymous inner classes can actually simplify your code. If you look at the code of the example so far, you will note that each button requires the same treatment:

1. Construct the button with a label string.
2. Add the button to the panel.

3. Construct an action listener with the appropriate color.
4. Add that action listener.

Let's implement a helper method to simplify these tasks:

```
void makeButton(String name, Color backgroundColor) {
    JButton button = new JButton(name);
    add(button);
    ColorAction action = new ColorAction(backgroundColor);
    button.addActionListener(action);
}
```

Then the ButtonPanel constructor simply becomes

```
public ButtonPanel() {
    makeButton("yellow", Color.YELLOW);
    makeButton("blue", Color.BLUE);
    makeButton("red", Color.RED);
}
```

Now you can make a further simplification. Note that the ColorAction class is only needed *once*: in the makeButton method. Therefore, you can make it into an anonymous class:

```
void makeButton(String name, final Color backgroundColor) {
    JButton button = new JButton(name);
    add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            setBackground(backgroundColor);
        }
    });
}
```

The action listener code has become quite a bit simpler. The actionPerformed method simply refers to the parameter variable backgroundColor. (As with all local variables that are accessed in the inner class, the parameter needs to be declared as final.)

No explicit constructor is needed. The inner class mechanism automatically generates a constructor that stores all local final variables that are used in one of the methods of the inner class.

Turning Components into Event Listeners

You are completely free to designate *any* object of a class that implements the ActionListener interface as a button listener. We prefer to use objects of a new class that was expressly created for carrying out the desired button actions. However, some programmers are not comfortable with inner classes and choose a different strategy. They locate the component that changes as a result of the event, make that component implement the ActionListener interface, and add an actionPerformed method. In our example, you can turn the ButtonPanel into an action listener:

```
class ButtonPanel extends JPanel implements ActionListener {
    . . .
    public void actionPerformed(ActionEvent event) {
        // set background color
        . . .
    }
}
```

Then the panel sets *itself* as the listener to all three buttons:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

Note that now the three buttons no longer have individual listeners. They share a single listener object, namely the button panel. Therefore, the actionPerformed method must figure out *which* button was clicked.

The `getSource` method of the `EventObject` class, the superclass of all other event classes, will tell you the *source* of every event. The event source is the object that generated the event and notified the listener:

```
Object source = event.getSource();
```

The `actionPerformed` method can then check which of the buttons was the source:

```
if (source == yellowButton) . . .
else if (source == blueButton) . . .
else if (source == redButton ) . . .
```

Of course, this approach requires that you keep references to the buttons as instance fields in the surrounding panel.

As you can see, turning the button panel into the action listener isn't really any simpler than defining an inner class. It also becomes *really* messy when the panel contains multiple user interface elements.

Capturing Window Events

Not all events are as simple to handle as button clicks. Here is an example of a more complex case that we already briefly noted in Chapter 7. Prior to the appearance of the `EXIT_ON_CLOSE` option in the Java SDK 1.3, programmers had to manually exit the program when the main frame was closed. In a non-toy program, you will want to do that as well, because you only want to close the program after you checked that the user won't lose work. For example, when the user closes the frame, you may want to put up a dialog to warn the user if unsaved work is about to be lost, and only exit the program when the user agrees.

When the program user tries to close a frame window, the `JFrame` object is the source of a `WindowEvent`. If you want to catch that event, you must have an appropriate listener object and add it to the list of window listeners.

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

The window listener must be an object of a class that implements the `WindowListener` interface. There are actually seven methods in the `WindowListener` interface. The frame calls them as the responses to seven distinct events that could happen to a window. The names are self-explanatory, except that "iconified" is usually called "minimized" under Windows. Here is the complete `WindowListener` interface:

```
public interface WindowListener {
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

As is always the case in Java, any class that implements an interface must implement all its methods; in this case, this means implementing *seven* methods. Recall that we are only interested in one of these seven methods, namely the `windowClosing` method.

Of course, we can define a class that implements the interface, add a call to `System.exit(0)` in the `windowClosing` method, and write do-nothing functions for the other six methods:

```
class Terminator implements WindowListener {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

```
}
```

Adapter Classes

Typing code for six methods that don't do anything is the kind of tedious busywork that nobody likes. To simplify this task, each of the AWT listener interfaces that has more than one method comes with a companion *adapter* class that implements all the methods in the interface but does nothing with them. For example, the `WindowAdapter` class has seven do-nothing methods. This means the adapter class automatically satisfies the technical requirements that Java imposes for implementing the associated listener interface. You can extend the adapter class to specify the desired reactions to some, but not all, of the event types in the interface. (An interface such as `ActionListener` that has only a single method does not need an adapter class.)

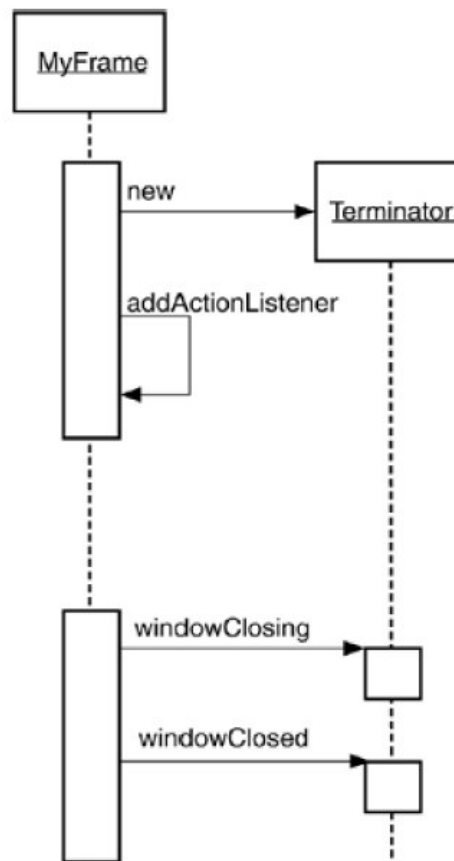
Let us make use of the window adapter. We can extend the `WindowAdapter` class, inherit six of the do-nothing methods, and override the `windowClosing` method:

```
class Terminator extends WindowAdapter {  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
}
```

Now you can register an object of type `Terminator` as the event listener:

```
WindowListener listener = new Terminator();  
frame.addWindowListener(listener);
```

Now, whenever the frame generates a window event, it passes it to the listener object by calling one of its seven methods



Creating a listener class that extends the `WindowAdapter` is an improvement, but we can go even further. There is no need to give a name to the listener object. Simply write

```
frame.addWindowListener(new Terminator());
```

But why stop there? We can make the listener class into an anonymous inner class of the frame.

```
frame.addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {
```

```

        System.exit(0);
    }
} );

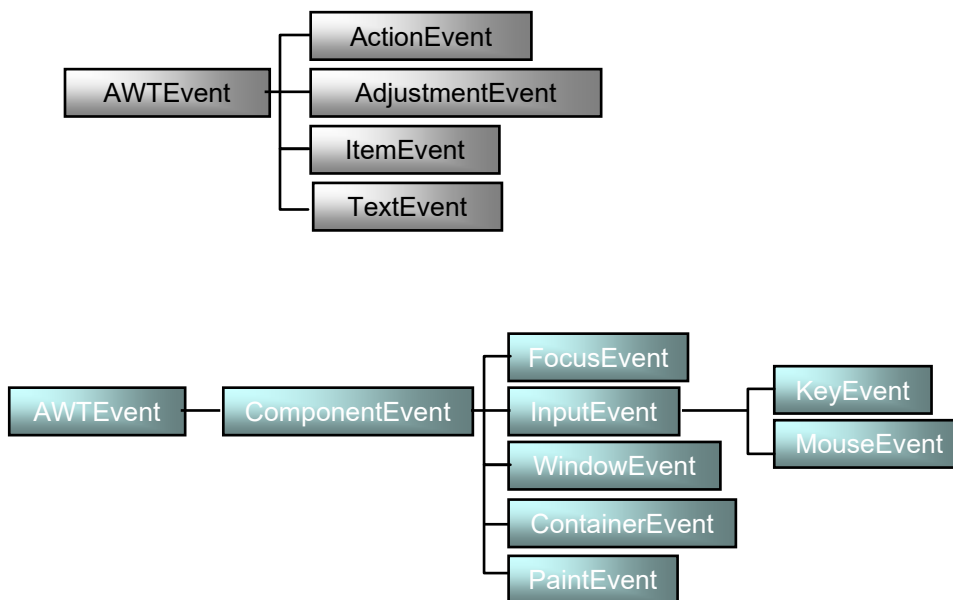
```

This code does the following:

1. Defines a class without a name that extends the WindowAdapter class. Adds a windowClosing method to that anonymous class. (As before, this method exits the program.)
2. Inherits the remaining six do-nothing methods from WindowAdapter.
3. Creates an object of this class. That object does not have a name, either.
4. Passes that object to the addWindowListener method.

The AWT Event Hierarchy

Having given you a taste of how event handling works, we want to turn to a more general discussion of event handling in Java. As we briefly mentioned earlier, event handling in Java is object oriented, with all events descending from the EventObject class in the java.util package. The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes.



WT Component	Tipi di Eventi Generati										
	action	adjustment	component	container	focus	item	key	mouse	mouse motion	text	window
Button	X		X		X		X	X	X		
Canvas			X		X		X	X	X		
Checkbox			X		X	X	X	X	X		
CheckboxMenuItem Nota: Non è una sottoclasse Component						X					
Choice			X		X	X	X	X	X		
Component			X		X		X	X	X		
Container			X	X	X		X	X	X		
Dialog			X	X	X		X	X	X		X
Frame			X	X	X		X	X	X		X
Label			X		X		X	X	X		
List	X		X		X	X	X	X	X		
MenuItem Nota: Non è una sottoclasse Component	X										
Panel			X	X	X		X	X	X		
Scrollbar		X	X		X		X	X	X		
ScrollPane			X	X	X		X	X	X		
TextArea			X		X		X	X	X	X	
TextComponent			X		X		X	X	X	X	
TextField	X		X		X		X	X	X	X	
Window			X	X	X		X	X	X		X

Listener Interface	Adapter Class	Methods	Generated Events:
ActionListener	No one	actionPerformed	AbstractButton JComboBox JTextField Timer
AdjustmentListener	No one	adjustmentValueChanged	JScrollbar
ComponentListener	ComponentAdapter	componentHidden componentMoved componentResized componentShown	Component
ContainerListener	ContainerAdapter	componentAdded componentRemoved	Container
FocusListener	FocusAdapter	focusGained focusLost	Component
ItemListener	no one	itemStateChanged	JComboBox, AbstractButton
KeyListener	KeyAdapter	keyPressed keyReleased keyTyped	Component
MouseListener	MouseAdapter	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	Component
MouseMotionListener	MouseMotionAdapter	mouseDragged mouseMoved	Component
TextListener	No one	textValueChanged	TextComponent
WindowListener	WindowAdapter	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened	Window

Border Layout

Java comes with several layout managers, and you can also make your own layout managers: We will cover all of them later on in this chapter. However, to enable us to give you more interesting examples right away, we need to briefly describe another layout manager called the *border layout manager*. This is the default layout manager of the content pane of every JFrame. Unlike the flow layout manager, which completely controls the position of each component, the border layout manager lets you choose where you want to place each component.

For example:

```
class MyPanel extends JPanel {
    setLayout(new BorderLayout());
    . . .
    add(yellowButton, BorderLayout.SOUTH);
}
```

The edge components are laid out first, and the remaining available space is occupied by the center. When the container is resized, the thickness of the edge components is unchanged, but the center component changes its size. You add components by specifying a constant CENTER, NORTH, SOUTH, EAST, or WEST of the BorderLayout class. Not all of the positions need to be occupied. If you don't supply any value, CENTER is assumed.

Unlike the flow layout, the border layout grows all components to fill the available space. (The flow layout leaves each component at its preferred size.)

As with flow layouts, if you want to specify a gap between the regions, you can do so in the constructor of the BorderLayout.

As previously noted, the content pane of a JFrame uses a border layout. Up to now, we never took advantage of this—we simply added panels into the default (center) area. But you can add components into the other areas as well:

```
Container contentPane = getContentPane();
contentPane.add(yellowButton, BorderLayout.SOUTH);
```

Panels

A BorderLayout is not very useful by itself. When you use the code fragment above. The button has grown to fill the entire southern region of the frame. And, if you were to add another button to the southern region, it would just displace the first button.

One common method to overcome this problem is to use additional *panels*. Panels act as (smaller) containers for interface elements and can themselves be arranged inside a larger panel under the control of a layout manager. For example, you can have one panel in the southern area for the buttons and another in the center for text. By nesting panels and using a mixture of border layouts and flow layouts, you can achieve fairly precise positioning of components. This approach to layout is certainly enough for prototyping, and it is the approach that we will use for the example programs in the first part of this chapter.

As you might expect, you first create a new instance of a JPanel object before you add the individual buttons to it. The default layout manager for a panel is a FlowLayout, which is a good choice for this situation. Finally, you add the individual buttons, using the add method you have seen before. Since you are adding buttons to a panel and haven't changed the default layout manager, the position of the buttons is under the control of the FlowLayout manager. This means the buttons stay centered within the panel and they do not expand to fill the entire panel area. Here's a code fragment that adds a panel containing three buttons in the south end of a container.

```
Container contentPane = getContentPane();
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
contentPane.add(panel, BorderLayout.SOUTH);
```

Grid Layout

The grid layout arranges all components in rows and columns like a spreadsheet. However, for a grid layout, cells are always the same size. The calculator program in Figure 9-11 uses a grid layout to arrange the calculator buttons. When you resize the window, the buttons grow and shrink, but all buttons have identical sizes.

In the constructor of the grid layout object, you specify how many rows and columns you need.

```
panel.setLayout(new GridLayout(5, 4));
```

As with the border layout and flow layout managers, you can also specify the vertical and horizontal gaps you want.

```
panel.setLayout(new GridLayout(5, 4, 3, 3));
```

The last two parameters of this constructor specify the size of the horizontal and vertical gaps (in pixels) between the components.

You add the components, starting with the first entry in the first row, then the second entry in the first row, and so on.

```
panel.add(new JButton("1"));
```

```
panel.add(new JButton("2"));
```

Of course, few applications have as rigid a layout as the face of a calculator. In practice, small grids (usually with just one row or one column) can be useful to organize partial areas of a window. For example, if you want to have a row of buttons with identical size, then you can put the buttons inside a panel that is governed by a grid layout with a single row.

Text Input

We are finally ready to start introducing the Swing user interface components. We'll start with components that let a user input and edit text. In Java, two components are used to get text input: *text fields* and *text areas*.

The difference between them is that a text field can accept only one line of text and a text area can accept multiple lines of text. The classes are called `JTextField` for single-line input and `JTextArea` for multiple lines of text.

Both of these classes inherit from a class called `JTextComponent`. You will not be able to construct a `JTextComponent` yourself since it is an abstract class. On the other hand, as is so often the case in Java, when you go searching through the API documentation, you may find that the methods you are looking for are actually in the parent class `JTextComponent` rather than in the derived class. For example, the methods that get or set the text in a text field or text area are actually methods in `JTextComponent`.

Text Fields

The usual way to add a text field to a window is to add it to a panel or other container—just as you would a button:

```
JPanel panel = new JPanel();
```

```
JTextField textField = new JTextField("Default input", 20);
```

```
panel.add(textField);
```

This code adds a text field and initializes the text field by placing the string "Default input" inside it. The second parameter of this constructor sets the width. In this case, the width is 20 "columns."

Unfortunately, a column is a rather imprecise measurement. One column is the expected width of one character in the font you are using for the text. The idea is that if you expect the inputs to be n characters or less, you are supposed to specify n as the column width. In practice, this measurement doesn't work out too well, and you should add 1 or 2 to the maximum input length to be on the safe side. Also, keep in mind that the number of columns is only a hint to the AWT that gives the *preferred* size. If the layout manager needs to grow or shrink the text field, it can adjust its size. The column width that you set in the `JTextField` constructor is not an upper limit on the number of characters the user can enter. The user can still type in longer strings, but the input scrolls when the text exceeds the length of the field. Users tend to find scrolling text fields irritating, so you should size the fields generously. If you need to reset the number of columns at run time, you can do that with the `setColumns` method.

In general, you want to let the user add text (or edit the existing text) in a text field. Quite often these text fields start out blank. To make a blank text field, just leave out the string as a parameter for the `JTextField` constructor:

```
JTextField textField = new JTextField(20);
```

You can change the contents of the text field at any time by using the `setText` method from the `TextComponent` parent class mentioned in the previous section. For example:

```
textField.setText("Hello!");
```

And, as was also mentioned in the previous section, you can find out what the user typed by calling the `getText` method. This method returns the exact text that the user typed. To trim any extraneous leading and trailing spaces from the data in a text field, apply the `trim` method to the return value of `getText`:

```
String text = textField.getText().trim();
```

To change the font in which the user text appears, use the `setFont` method.

Text Areas

Sometimes, you need to collect user input that is more than one line long. As mentioned earlier, you use the `JTextArea` component for this collection. When you place a text area component in your program, a user can enter any number of lines of text, using the ENTER key to separate them. Each line ends with a '\n' as far as Java is concerned. If you need to break up what the user enters into separate lines, you can use the `StringTokenizer` class. In the constructor for the `JTextArea` component, you specify the number of rows and columns for the text area. For example:

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each
contentPane.add(textArea);
```

where the columns parameter works as before—and you still need to add a few more columns for safety's sake. Also, as before, the user is not restricted to the number of rows and columns; the text simply scrolls when the user inputs too much. You can also use the `setColumns` method to change the number of columns, and the `setRows` method to change the number of rows. These numbers only indicate the preferred size—the layout manager can still grow or shrink the text area. If there is more text than the text area can display, then the remaining text is simply clipped. You can avoid clipping long lines by turning on line-wrapping:

```
textArea.setLineWrap(true); // long lines are wrapped
```

This wrapping is a visual effect only; the text in the document is not changed—no '\n' characters are inserted into the text.

In Swing, a text area does not have scroll bars. If you want scroll bars, you have to insert the text area inside a *scroll pane*. Then, insert the scroll pane inside the content pane.

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
contentPane.add(scrollPane, BorderLayout.CENTER);
```

The scroll pane now manages the view of the text area. Scroll bars automatically appear if there is more text than the text area can display, and they vanish again if text is deleted and the remaining text fits inside the area. The scrolling is handled internally in the scroll pane—your program does not need to process scroll events.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextAreaTest{
    public static void main(String[] args){
        TextAreaFrame frame = new TextAreaFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.show();
    }
}

/**
A frame with a text area and buttons for text editing
*/
class TextAreaFrame extends JFrame{
```

```

public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 300;
private JTextArea textArea;
private JScrollPane scrollPane;
private JPanel buttonPanel;
private JButton wrapButton;
public TextAreaFrame() {
    setTitle("TextAreaTest");
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    Container contentPane = getContentPane();
    buttonPanel = new JPanel();
    // add button to append text into the text area
    JButton insertButton = new JButton("Insert");
    buttonPanel.add(insertButton);
    insertButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            textArea.append("The quick brown fox " + "jumps over the lazy
            dog. ");
        }
    });
    // add button to turn line wrapping on and off
    wrapButton = new JButton("Wrap");
    buttonPanel.add(wrapButton);
    wrapButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            boolean wrap = !textArea.getLineWrap();
            textArea.setLineWrap(wrap);
            scrollPane.validate();
            wrapButton.setText(wrap ? "No Wrap" : "Wrap");
        }
    });
    contentPane.add(buttonPanel, BorderLayout.SOUTH);
    // add a text area with scroll bars
    textArea = new JTextArea(8, 40);
    scrollPane = new JScrollPane(textArea);
    contentPane.add(scrollPane, BorderLayout.CENTER);
}
}

```

Labels and Labeling Components

Labels are components that hold text. They have no decorations (for example, no boundaries). They also do not react to user input. You can use a label to identify components. For example, unlike buttons, text components have no label to identify them. To label a component that does not itself come with an identifier:

1. Construct a JLabel component with the correct text.
2. Place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

The constructor for a JLabel lets you specify the initial text or icon, and optionally, the alignment of the contents. You use constants from the SwingConstants interface to specify alignment. That interface defines a number of useful constants such as LEFT, RIGHT, CENTER, NORTH, EAST, and so on. The JLabel class is one of several Swing classes that implements this interface.

Therefore, you can specify a left-aligned label either as:

```
JLabel label = new JLabel("Text", SwingConstants.LEFT);
```

or

```
JLabel label = new JLabel("Text", JLabel.LEFT);
```

The setText and setIcon methods let you set the text and icon of the label at run time.

Labels can be positioned inside a container like any other component. This means you can use the techniques you have seen before to place labels where you need them.

Check Boxes

You now know how to collect text input from users, but there are many occasions where you would rather give users a finite set of choices than have them enter the data in a text component. Using a set of buttons or a list of items tells your users what choices they have. (It also saves you the trouble of error checking.) In this section, you will learn how to program check boxes, radio buttons, lists of choices, and sliders.

If you want to collect just a "yes" or "no" input, use a check box component. Check boxes automatically come with labels that identify them. The user usually checks the box by clicking inside it, and turns off the check mark by clicking inside the box again. To toggle the check mark, the user can also press the space bar when the focus is in the check box.

Let consider a simple program with two check boxes, one to turn on or off the italic attribute of a font, and the other for boldface. Note that the second check box has focus, as indicated by the rectangle around the label. Each time the user clicks one of the check boxes, we refresh the screen, using the new font attributes.

Check boxes need a label next to them to identify their purpose. You give the label text in the constructor.

```
bold = new JCheckBox("Bold");
```

You use the `setSelected` method to turn a check box on or off. For example,

```
bold.setSelected(true);
```

The `isSelected` method then retrieves the current state of each check box. It is false if unchecked; true if checked.

When the user clicks on a check box, this triggers an action event. As always, you attach an action listener. In our program, the two buttons share the same action listener.

```
ActionListener listener = . . .
```

```
bold.addActionListener(listener);
```

```
italic.addActionListener(listener);
```

Its `actionPerformed` method queries the state of the bold and italic check boxes and sets the font of the panel to plain, bold, italic, or both.

```
public void actionPerformed(ActionEvent event) {
    int mode = 0;
    if (bold.isSelected()) mode += Font.BOLD;
    if (italic.isSelected()) mode += Font.ITALIC;
    label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

Radio Buttons

In the previous example, the user could check either, both, or none of the two check boxes. In many cases, we want to require the user to check only one of several boxes. When another box is checked, the previous box is automatically unchecked. Such a group of boxes is often called a *radio button group* because the buttons work like the station selector buttons on a radio. When you push in one button, the previously depressed button pops out. We allow the user to select a font size among the choices—Small, Medium, Large, and Extra large—but, of course, we will allow the user to select only one size at a time.

Implementing radio button groups is easy in Swing. You construct one object of type `ButtonGroup` for every group of buttons. Then, you add objects of type `JRadioButton` to the button group. The button group object is responsible for turning off the previously set button when a new button is clicked.

```
ButtonGroup group = new ButtonGroup();
```

```
JRadioButton smallButton = new JRadioButton("Small", false);
```

```
group.add(smallButton);
```

```
JRadioButton mediumButton = new JRadioButton("Medium", true);
```

```
group.add(mediumButton);
```

```
. . .
```

The second argument of the constructor is true for the button that should be checked initially, and false for all others.

Note that the button group controls only the *behavior* of the buttons; if you want to group the buttons together for layout purposes, you also need to add them to a container such as a JPanel. The event notification mechanism for radio buttons is the same as for any other buttons. When the user checks a radio button, the radio button generates an action event. In our example program, we define an action listener that sets the font size to a particular value:

```
ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        // size refers to the final parameter of the
        // addRadioButton method
        label.setFont(new Font("Serif", Font.PLAIN, size));
    }
};
```

Combo Boxes

If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead, you can use a combo box. When the user clicks on the component, a list of choices drops down, and the user can then select one of them.

If the drop-down list box is set to be *editable*, then you can edit the current selection as if it was a text field. For that reason, this component is called a *combo box*—it combines the flexibility of an edit box with a set of predefined choices. The JComboBox class provides a combo box component.

You call the `setEditable` method to make the combo box editable. Note that editing affects only the current item. It does not change the contents of the list.

You can obtain the current selection or edited text by calling the `getSelectedItem` method.

In the example program, the user can choose a font style from a list of styles (Serif, SansSerif, Monospaced, etc.).

The user can also type in another font.

You add the choice items with the `addItem` method. In our program, `addItem` is called only in the constructor, but you can call it any time.

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

This method adds the string at the end of the list. You can add new items anywhere in the list with the `insertItemAt` method:

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

You can add items of any type—the combo box invokes each item's `toString` method to display it.

If you need to remove items at run time, you use the `removeItem` or `removeItemAt` method, depending on whether you supply the item to be removed or its position.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

There is also a `removeAllItems` method to remove all items at once.

When the user selects an item from a combo box, the combo box generates an action event. To find out which item was selected, call `getSource` on the event parameter to get a reference to the combo box that sent the event. Then call the `getSelectedItem` method to retrieve the currently selected item. You need to cast the returned value to the appropriate type, usually `String`.

```
public void actionPerformed(ActionEvent event) {
    label.setFont(new Font((String) faceCombo.getSelectedItem(), Font.PLAIN,
        DEFAULT_SIZE));
}
```

Mouse Events

You do not need to handle mouse events explicitly if you just want the user to be able to click on a button or menu. These mouse operations are handled internally by the various components in the user interface and then translated into the appropriate semantic event. However, if you want to enable the user to draw with the mouse, you will need to trap mouse move, click, and drag events. In this section, we will show you a simple graphics editor application that intercept the mouse events and print them in a text area.

When the user clicks a mouse button, three listener methods are called: `mousePressed` when the mouse is first pressed, `mouseReleased` when the mouse is released, and, finally, `mouseClicked`. If you are only interested in complete clicks, you can ignore the first two methods. By using the `getX` and `getY` methods on the `MouseEvent` argument, you can obtain the *x*- and *y*-coordinates of the mouse pointer when the mouse was clicked. If you want to distinguish between single, double and triple (!) clicks, use the `getClickCount` method.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
 * Window that captures mouse events launched by a Canvas
 * and visualises them in a TextArea component
 */
public class MouseEvents extends JFrame implements MouseListener {
    private JTextArea ta;
    public MouseEvents (String title){
        super(title);
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(1,2));
        Canvas cv = new Canvas();
        cv.addMouseListener(this);
        contentPane.add(cv);
        ta = new JTextArea(20, 40);
        contentPane.add(ta);
    }
    public void mouseClicked(MouseEvent e) {
        ta.append("Mouse click. x=" + e.getX() + " y=" + e.getY() + "\n");
    }
    public void mouseEntered(MouseEvent e) {
        ta.append("Mouse entered. x=" + e.getX() + " y=" + e.getY() + "\n");
    }
    public void mouseExited(MouseEvent e) {
        ta.append("Mouse exited. x=" + e.getX() + " y=" + e.getY() + "\n");
    }
    public void mousePressed(MouseEvent e) {
        ta.append("Mouse pressed. x=" + e.getX() + " y=" + e.getY() + "\n");
    }
    public void mouseReleased(MouseEvent e) {
        ta.append("Mouse released. x=" + e.getX() + " y=" + e.getY() +
"\n");
    }
    public static void main(String[] args) {
        JFrame f = new MouseEvents ("Demo");
        f.setBounds(40, 60, 450, 350);
        f.setVisible(true);
    }
}
```

Some user interface designers inflict mouse click and keyboard modifier combinations, such as **CONTROL + SHIFT + CLICK**, on their users. We find this practice reprehensible, but if you

disagree, you will find that checking for mouse buttons and keyboard modifiers is a mess. In the original API, two of the button masks equal two keyboard modifier masks, namely:

```
BUTTON2_MASK == ALT_MASK  
BUTTON3_MASK == META_MASK
```

Presumably this was done so that users with a one-button mouse could simulate the other mouse buttons by holding down modifier keys instead. However, as of SDK 1.4, a different approach is recommended. There are now masks

```
BUTTON1_DOWN_MASK  
BUTTON2_DOWN_MASK  
BUTTON3_DOWN_MASK  
SHIFT_DOWN_MASK  
CTRL_DOWN_MASK  
ALT_DOWN_MASK  
ALT_GRAPH_DOWN_MASK  
META_DOWN_MASK
```

The `getModifiersEx` method accurately reports the mouse buttons and keyboard modifiers of a mouse event.

Applet Basics

Before Java, you used HTML (the hypertext markup language) to describe the layout of a web page. HTML is simply a vehicle to indicate elements of a hypertext page. For example, `<title>` indicates the title of the page, and any text that follows this tag becomes the title of the page. You indicate the end of the title with the `</title>` tag. (This is one of the general rules for tags: a slash followed by the name of the element indicates the end of the element.) The basic idea of how to use applets in a web page is simple: the HTML page must tell the browser which applets to load and then where to put each applet on the web page. As you might expect, the tag needed to use an applet must tell the browser the following:

- From where to get the class files;
- How the applet sits on the web page (size, location, and so on).

The browser then retrieves the class files from the internet (or from a directory on the user's machine) and automatically runs the applet, using an external Java Runtime Environment or its built-in Java Virtual Machine.

In addition to the applet itself, the web page can contain all the other HTML elements you have seen in use on web pages: multiple fonts, bulleted lists, graphics, links, and so on. Applets are just one part of the hypertext page. It is always worth keeping in mind that the Java programming language is *not* a tool for designing HTML pages; it is a tool for *bringing them to life*. This is not to say that the GUI design elements in a Java applet are not important, but that they must work with (and, in fact, are subservient to) the underlying HTML design of the web page.

When applets were first developed, you had to use Sun's HotJava browser to view web pages that contain applets. Naturally, few users were willing to use a separate browser just to enjoy a new web feature. Java applets became really popular when Netscape included a Java Virtual Machine in its Navigator browser.

Microsoft Internet Explorer soon followed suit. Unfortunately, then two problems happened. Netscape didn't keep up with more modern versions of Java, and Microsoft vacillated between reluctantly supporting outdated Java versions and dropping Java support altogether.

To overcome this problem, Sun released a tool called the "Java Plug-In". Using the various extension mechanisms of Internet Explorer or Navigator, it seamlessly plugs into both Netscape and Internet Explorer and allows both browsers to execute Java applets by using an external Java Runtime Environment that Sun supplies. By keeping the Plug-In up-to-date, you can always take advantage of the latest and greatest features of Java.

A Simple Applet

For tradition's sake, let's write a `NotHelloWorld` program as an applet. Before we do that, we want to point out that, from a programming point of view, an applet isn't very strange. An applet is simply

a Java class that (ultimately) extends the `java.applet.Applet` class. Note that although the `java.applet` package is not part of the AWT package, an applet is an AWT component. In this book, we will use the Swing set to implement applets. All of our applets will extend the `JApplet` class, the superclass for Swing applets. As you can see in Figure 10-1, `JApplet` is an immediate subclass of the ordinary `Applet` class.

```
import java.awt.*;
import javax.swing.*;
public class NotHelloWorldApplet extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        JLabel label = new JLabel("Not a Hello, World applet", SwingConstants.CENTER);
        contentPane.add(label);
    }
}
```

Viewing an Applet

To execute the applet, you need to carry out two steps:

1. Compile your source files into class files.
2. Create an HTML file that tells the browser which class file to load first and how to size the applet.

It is customary (but not necessary) to give the HTML file the same name as that of the applet class inside. So, following this tradition, we will call the file `NotHelloWorldApplet.html`. Here are the contents of the file:

```
<applet code="NotHelloWorldApplet.class" width="300" height="300">
</applet>
```

Before viewing the applet in a browser, it is a good idea to test it in the *applet viewer* program that is a part of the Java SDK. To use the applet viewer in our example, enter

```
appletviewer NotHelloWorldApplet.html
```

at the command line. The command line for the applet viewer program is the name of the HTML file, not the class file.

To properly view the applet, you need a Java 2 enabled browser. After installing and configuring the Java Plug-In, simply load the HTML file into the browser (see Figure 10-3). If the applet doesn't show up, your browser probably uses its built-in virtual machine, and you need to configure it to use the Java Plug-In instead.

Life Cycle of an Applet

Four methods in the `Applet` class give you the framework on which you build any serious applet: `init`, `start`, `stop`, `destroy`. What follows is a short description of these methods, when these methods are called, and what code you should place into them.

init

This method is used for whatever initialization is needed for your applet. This works much like a constructor—it is automatically called by the system when Java launches the applet for the first time. Common actions in an applet include processing `param` values and adding user interface components.

Applets can have a default constructor, but it is customary to perform all initialization in the `init` method instead of the default constructor.

start

This method is automatically called *after* Java calls the `init` method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages. This means that the `start` method can be called repeatedly, unlike the `init` method. For this reason, put the code that you want executed only once in the `init` method, rather than in the `start` method. The `start` method is where you usually restart a thread for your applet, for example, to resume an animation. If your applet does nothing that needs to be suspended when the user leaves the current web page, you do not need to implement this method (or the `stop` method).

stop

This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet. Its purpose is to give you a chance to stop a time-consuming activity from slowing down the system when the user is not paying attention to the applet. You should not call this method directly. If your applet does not perform animation, play audio files, or perform calculations in a thread, you do not usually need this method.

destroy

This method is only called when the browser shuts down normally. Since applets are meant to live on an HTML page, you should not normally leave resources behind after a user stops viewing the page that contains the applet. But if you do, then you can close down the resources by overriding the destroy method.

Security Basics

Because applets are designed to be loaded from a remote site and then executed locally, security becomes vital. If a user enables Java in the browser, the browser will download all the applet code on the web page and execute it immediately. The user never gets a chance to confirm or to stop individual applets from running. For this reason, applets (unlike applications) are restricted in what they can do. The *applet security manager* throws a `SecurityException` whenever an applet attempts to violate one of the access rules.

What *can* applets do on all platforms? They can show images and play sounds, get keystrokes and mouse clicks from the user, and send user input back to the host from which they were loaded. That is enough functionality to show facts and figures or to get user input for placing an order. The restricted execution environment for applets is often called the "sandbox." Applets playing in the "sandbox" cannot alter the user's system or spy on it. In this chapter, we will look only at applets that run inside the sandbox.

In particular, when running in the sandbox:

- ❑ Applets can *never* run any local executable program.
- ❑ Applets cannot communicate with any host other than the server from which they were downloaded; that server is called the *originating host*. This rule is often called "applets can only phone home." This protects applet users from applets that might try to spy on intranet resources.
- ❑ Applets cannot read from or write to the local computer's file system.
- ❑ Applets cannot find out any information about the local computer, except for the Java version used, the name and version of the operating system, and the characters used to separate files (for instance, / or \), paths (such as : or ;), and lines (such as \n or \r\n). In particular, applets cannot find out the user's name, e-mail address, and so on.
- ❑ All windows that an applet pops up carry a warning message.

All this is possible only because applets are *interpreted* by the Java Virtual Machine and not directly executed by the CPU on the user's computer. Because the interpreter checks all critical instructions and program areas, a hostile (or poorly written) applet will almost certainly not be able to crash the computer, overwrite system memory, or change the privileges granted by the operating system.

These restrictions are too strong for some situations. For example, on a corporate intranet, you can certainly imagine an applet wanting to access local files. To allow for different levels of security under different situations, you can use *signed applets*. A signed applet carries with it a certificate that indicates the identity of the signer. Cryptographic techniques ensure that such a certificate cannot be forged. If you trust the signer, you can choose to give the applet additional rights. The point is that if you trust the signer of the applet, you can tell the browser to give the applet more privileges.

You can, for example, give applets in your corporate intranet a higher level of trust than those from www.hacker.com. The configurable Java security model allows the continuum of privilege levels you need.

You can give completely trusted applets the same privilege levels as local applications. Programs from vendors that are known to be somewhat flaky can be given access to some, but not all, privileges. Unknown applets can be restricted to the sandbox.

To sum up, Java has three separate mechanisms for enforcing security:

1. Program code is interpreted by the Java Virtual Machine, not executed directly.
2. A security manager checks all sensitive operations in the Java runtime library.
3. Applets can be signed to identify their origin.