Introduction to Object Oriented Programming in Java

1.1. What is Object Oriented Programming (OOP)?

- A software design method that models the characteristics of real or abstract objects using software classes and objects.
- Characteristics of objects:
    - State (what the objects have)
    - Behavior (what the objects do)
    - Identity (what makes them unique)
- Definition: an object is a software bundle of related *fields* (variables) and *methods*.
- In OOP, a program is a collection of objects that act on one another (vs. procedures).

For example, a car is an object. Its state includes current:

- Speed
- RPM
- Gear
- Direction
- Fuel level
- Engine temperature

Its behaviors include:

- Change Gear
- Go faster/slower
- Go in reverse
- Stop
- Shut-off

Its identity is:

- VIN
- License Plate

1.2. Why OOP?

- *Modularity* — Separating entities into separate logical units makes them easier to code, understand, analyze, test, and maintain.
- *Data hiding* (*encapsulation*) — The implementation of an object's private data and actions can change without affecting other objects that depend on it.
- Code reuse through:
    - *Composition* — Objects can contain other objects
    - *Inheritance* — Objects can inherit state and behavior of other objects
- Easier design due to natural modeling

Even though OOP takes some getting used to, its main benefit is to make it easier to solve real-world problems by modeling natural objects in software objects. The OO thought process is more intuitive than procedural, especially for tackling complex problems.

Although a lot of great software is implemented in procedural languages like C, OO languages typically scale better for taking on medium to large software projects.

1.3. Class vs. Object

- A *class* is a template or blueprint for how to build an object.
    - A class is a prototype that defines state placeholders and behavior common to all objects of its kind.
    - Each object is a member of a single class — there is no multiple inheritance in Java.
- An *object* is an instance of a particular class.
    - There are typically many object instances for any one given class (or type).
    - Each object of a given class has the same built-in behavior but possibly a different state (data).
    - Objects are *instantiated* (created).

For example, each car starts of with a design that defines its features and properties.

It is the design that is used to build a car of a particular type or class.

When the physical cars roll off the assembly line, those cars are *instances* (concrete objects) of that class.

Many people can have a 2007 BMW 335i, but there is typically only one design for that particular class of cars.

As we will see later, classification of objects is a powerful idea, especially when it comes to inheritance — or classification hierarchy.

1.4. Classes in Java

- Everything in Java is defined in a class.
- In its simplest form, a class just defines a collection of data (like a record or a C `struct`). For example:

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;
}
```

- The order of data fields and methods in a class is not significant.

If you recall, each class must be saved in a file that matches its name, for example: `Employee.java`

There are a few exceptions to this rule (for non-`public` classes), but the accepted convention is to have one class defined per source file.

Note that in Java, `Strings` are also classes rather than being implemented as primitive types.

Unlike local variables, the state variables (known as *fields*) of objects do not have to be explicitly initialized. Primitive fields (such as `yearOfBirth`) are automatically set to primitive defaults (`0` in this case), whereas objects (`name`, `ssn`, `emailAddress`) are automatically set to `null` — meaning that they do not point to any object.

1.5. Objects in Java

- To create an object (instance) of a particular class, use the `new` operator, followed by an invocation of a *constructor* for that class, such as:

  ```
  new MyClass()
  ```

  - The constructor method initializes the state of the new object.
  - The `new` operator returns a *reference* to the newly created object.
- As with primitives, the variable type must be compatible with the value type when using object references, as in:

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

- To access member data or methods of an object, use the dot (`.`) notation: *`variable.field`* or *`variable.method`*`()`

We'll explore all of these concepts in more depth in this module.

Consider this simple example of creating and using instances of the `Employee` class:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "456-78-901";
        e2.yearOfBirth = 1974;

        System.out.println("Name: " + e1.name);
        System.out.println("SSN: " + e1.ssn);
        System.out.println("Email Address: " + e1.emailAddress);
```

```
        System.out.println("Year Of Birth: " + e1.yearOfBirth);

        System.out.println("Name: " + e2.name);
        System.out.println("SSN: " + e2.ssn);
        System.out.println("Email Address: " + e2.emailAddress);
        System.out.println("Year Of Birth: " + e2.yearOfBirth);

    }
}
```

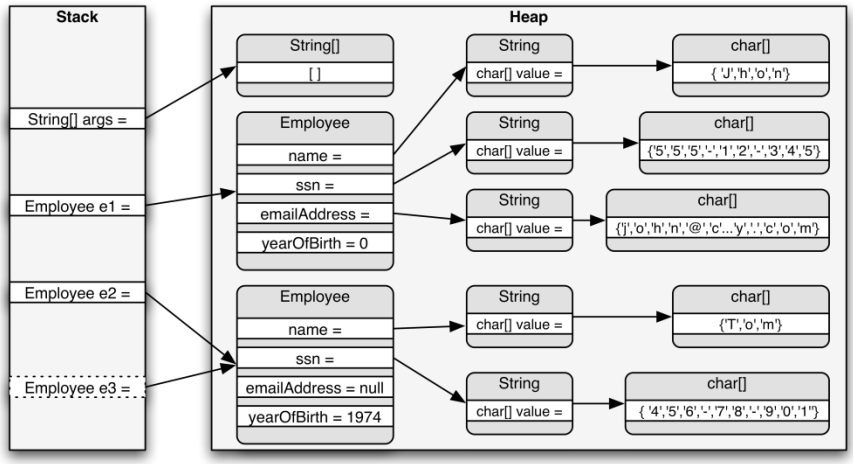
Running this code produces:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 451-78-901
Email Address: null
Year Of Birth: 1974
```

1.6. Java Memory Model

- Java variables do not contain the actual objects, they contain *references* to the objects.
    - The actual objects are stored in an area of memory known as the *heap*.
    - Local variables referencing those objects are stored on the stack.
    - More than one variable can hold a reference to the same object.

Figure 4. Java Memory Model



As previously mentioned, the *stack* is the area of memory where local variables (including method parameters) are stored. When it comes to object variables, these are merely *references* (pointers) to the actual objects on the heap.

Every time an object is instantiated, a chunk of *heap memory* is set aside to hold the data (state) of that object. Since objects can contain other objects, some of this data can in fact hold references to those nested objects.

In Java:

- Object references can either point to an actual object of a compatible type, or be set to `null` (`0` is not the same as `null`).
- It is not possible to instantiate objects on the stack. Only local variables (primitives and object references) can live on the stack, and everything else is stored on the heap, including classes and static data.

1.7. Accessing Objects through References

```
Employee e1 = new Employee();
Employee e2 = new Employee();

// e1 and e2 refer to two independent Employee objects on the heap

Employee e3 = e1;

// e1 and e3 refer to the *same* Employee object

e3 = e2;

// Now e2 and e3 refer to the same Employee object

e1 = null;

// e1 no longer refers to any object. Additionally, there are no references
// left to the Employee object previously referred to by e1. That "orphaned"
// object is now eligible for garbage collection.
```

Note

The statement `Employee e3 = e2;` sets `e3` to point to the same physical object as `e2`. It does **not** duplicate the object. Changes to `e3` are reflected in `e2` and vice-versa.

1.8. Garbage Collection

- Unlike some OO languages, Java does not support an explicit *destructor* method to delete an object from memory.
  - Instead, unused objects are deleted by a process known as *garbage collection*.
- The JVM automatically runs garbage collection periodically. Garbage collection:
  - Identifies objects no longer in use (no references)
  - Finalizes those objects (deconstructs them)
  - Frees up memory used by destroyed objects
  - Defragments memory
- Garbage collection introduces overhead, and can have a major affect on Java application performance.
  - The goal is to avoid how often and how long GC runs.

    - Programmatically, try to avoid unnecessary object creation and deletion.
    - Most JVMs have tuning parameters that affect GC performance.

Benefits of garbage collection:

- Frees up programmers from having to manage memory. Manually identifying unused objects (as in a language such as C++) is not a trivial task, especially when programs get so complex that the responsibility of object destruction and memory deallocation becomes vague.
- Ensures integrity of programs:
    - Prevents memory leaks — each object is tracked down and disposed off as soon as it is no longer used.
    - Prevents deallocation of objects that are still in use or have already been released. In Java it is impossible to explicitly deallocate an object or use one that has already been deallocated. In a language such as C++ dereferencing null pointers or double-freeing objects typically crashes the program.

Through Java command-line switches (`java -X`), you can:

- Set minimum amount of memory (e.g. `-Xmn`)
- Set maximum amount of memory (e.g. `-Xmx`, `-Xss`)
- Tune GC and memory integrity (e.g. `-XX:+UseParallelGC`)

For more information, see: http://java.sun.com/docs/hotspot/VMOptions.html and http://www.petefreitag.com/articles/gctuning/

1.9. Methods in Java

- A *method* is a set of instructions that defines a particular behavior.
    - A method is also known as a *function* or a *procedure* in procedural languages.
- Java allows procedural programming through its *static* methods (e.g. the `main()` method).
    - A static method belongs to a class independent of any of the class's instances.
- It would be possible to implement an entire program through static methods, which call each other procedurally, but that is not OOP.

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "456-78-901";
        e2.yearOfBirth = 1974;
```

```
        printEmployee(e1);
        printEmployee(e2);
    }

    static void printEmployee(Employee e) {
        System.out.println("Name: " + e.name);
        System.out.println("SSN: " + e.ssn);
        System.out.println("Email Address: " + e.emailAddress);
        System.out.println("Year Of Birth: " + e.yearOfBirth);
    }
}
```

Running this code produces the same output as before:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 451-78-901
Email Address: null
Year Of Birth: 1974
```

1.10. Methods in Java (cont.)

- In true OOP, we combine an object's state and behavior together.
    - For example, rather than having external code access the individual fields of an Employee object and print the values, an Employee object could know how to print itself:

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;

    void print() {
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
    }
}
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "451-78-901";
        e2.yearOfBirth = 1974;
```

```
        e1.print();
        e2.print();
    }
}
```

Running this code produces the same output as before:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 456-78-901
Email Address: null
Year Of Birth: 1974
```

1.11. Method Declarations

Each method has a *declaration* of the following format:

modifiers returnType name(*params*) *throws-clause* { *body* }

modifiers

`public`, `private`, `protected`, `static`, `final`, `abstract`, `native`, `synchronized`

returnType

A primitive type, object type, or `void` (no return value)

name

The name of the method

params

paramType paramName, ...

throws-clause

`throws` *ExceptionType*, ...

body

The method's code, including the declaration of local variables, enclosed in braces

Note that abstract methods do not have a body (more on this later).

Here are some examples of method declarations:

```
public static void print(Employee e) { ... }
public void print() { ... }
public double sqrt(double n) { ... }
public int max(int x, int y) { ... }
public synchronized add(Employee e) throws DuplicateEntryException { ... }
public int read() throws IOException { ... }
public void println(Object o) { ... }
protected void finalize() throws Throwable { ... }
public native void write(byte[] buffer, int offset, int length) throws
IOException { ... }
public boolean equals(Object o) { ... }
private void process(MyObject o) { ... }
```

```
void run() { ... }
```

1.12. Method Signatures

- The *signature* of a method consists of:
  - The method name
  - The parameter list (that is, the parameter types and their order)
- The signature does **not** include:
  - The parameter names
  - The return type
- Each method defined in a class must have a unique signature.
- Methods with the same name but different signatures are said to be *overloaded*.

We'll discuss examples of overloading constructors and other methods later in this module.

1.13. Invoking Methods

- Use the dot (`.`) notation to invoke a method on an object: *`objectRef.method(params)`*
- Parameters passed into methods are always copied ("pass-by-value").
  - Changes made to parameter variables within the methods do no affect the caller.
  - Object references are also copied, but they still point to the *same object*.

For example, we add the following method to our Employee class:

```
void setYearOfBirth(int year) {
    yearOfBirth = year;
    year = -1;      // modify local variable copy
}
```

We invoke it from `EmployeeDemo.main()` as:

```
int y = 1974;
e2.setYearOfBirth(y);
System.out.println(e2.yearOfBirth); // prints 1974
System.out.println(y);          // prints 1974
```

On the other hand, we add this method to our EmployeeDemo class:

```
static void printYearOfBirth(Employee e) {
    System.out.println(e.yearOfBirth);
    e.yearOfBirth = -1; // modify object's copy
}
```

We invoke it from `EmployeeDemo.main()` as:

```
printYearOfBirth(e2);           // prints 1974
System.out.println(e2.yearOfBirth); // prints -1
```

1.14. Static vs. Instance Data Fields

- Static (or class) data fields
  - Unique to the entire class
  - Shared by all instances (objects) of that class
  - Accessible using *`ClassName.fieldName`*
  - The class name is optional within static and instance methods of the class, unless a local variable of the same name exists in that scope
  - Subject to the declared access mode, accessible from outside the class using the same syntax
- Instance (object) data fields
  - Unique to each instance (object) of that class (that is, each object has its own set of instance fields)
  - Accessible within instance methods and constructors using `this.`*`fieldName`*
  - The `this.` qualifier is optional, unless a local variable of the same name exists in that scope
  - Subject to the declared access mode, accessible from outside the class from an object reference using *`objectRef.fieldName`*

Say we add the following to our Employee class:

```
static int vacationDays = 10;
```

and we print this in the Employee's `print()` method:

```
System.out.println("Vacation Days: " + vacationDays);
```

In the EmployeeDemo's `main()` method, we change `vacationDays` to 15:

```
Employee.vacationDays = 15;
```

Now, `e1.print()` and `e2.print()` will both show the vacation days set to 15. This is because both `e1` and `e2` (and any other Employee object) share the static `vacationDays` integer field.

The field `vacationDays` is part of the Employee class, and this is also stored on the heap, where it is shared by all objects of that class.

Static fields that are not protected (which we will soon learn how to do) are almost like global variables — accessible to anyone.

Note that it is possible to access static fields through instance variables (e.g., `e1.vacationDays = 15;` will have the same effect), however this is discouraged. You should always access static fields by *`ClassName.staticFieldName`*, unless you are within the same class, in which case you can just say *`staticFieldName`*.

1.15. Static vs. Instance Methods

- Static methods can access only static data and invoke other static methods.
    - Often serve as helper procedures/functions
    - Use when the desire is to provide a utility or access to class data only
- Instance methods can access both instance and static data and methods.
    - Implement behavior for individual objects
    - Use when access to instance data/methods is required
- An example of static method use is Java's Math class.
    - All of its functionality is provided as static methods implementing mathematical functions (e.g., `Math.sin()`).
    - The Math class is designed so that you don't (and can't) create actual Math instances.
- Static methods also are used to implement *factory methods* for creating objects, a technique discussed later in this class.

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;
    int extraVacationDays = 0;
    static int baseVacationDays = 10;

    Employee(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }

    static void setBaseVacationDays(int days) {
        baseVacationDays = days < 10? 10 : days;
    }

    static int getBaseVacationDays() {
        return baseVacationDays;
    }

    void setExtraVacationDays(int days) {
        extraVacationDays = days < 0? 0 : days;
    }

    int getExtraVacationDays() {
        return extraVacationDays;
    }

    void setYearOfBirth(int year) {
        yearOfBirth = year;
    }

    int getVacationDays() {
        return baseVacationDays + extraVacationDays;
    }

    void print() {
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
```

```
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
        System.out.println("Vacation Days: " + getVacationDays());
    }
}
```

To change the company vacation policy, do `Employee.setBaseVacationDays(15);`

To give one employee extra vacation, do `e2.setExtraVacationDays(5);`

1.16. Method Overloading

- A class can provide multiple definitions of the same method. This is known as *overloading*.
- Overloaded methods *must* have distinct signatures:
    - The parameter type list must be different, either different number or different order.
    - Only parameter *types* determine the signature, not parameter *names*.
    - The return type is not considered part of the signature.

We can overload the print method in our Employee class to support providing header and footer to be printed:

```
public class Employee {
    …
    public void print(String header, String footer) {
        if (header != null) {
            System.out.println(header);
        }
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
        System.out.println("Vacation Days: " + getVacationDays());
        if (footer != null) {
            System.out.println(footer);
        }
    }

    public void print(String header) {
        print(header, null);
    }

    public void print() {
        print(null);
    }
}
```

In our `EmployeeDemo.main()` we can then do:

```
e1.print("COOL EMPLOYEE");
e2.print("START OF EMPLOYEE", "END OF EMPLOYEE");
```

1.17. Variable Argument Length Methods

- Java 5 introduced syntax supporting methods with a variable number of argument (also known as *varargs*).
    - The last parameter in the method declaration must have the format `Type... varName` (literally three periods following the type).
    - The arguments corresponding to the parameter are presented as an *array* of that type.
    - You can also invoke the method with an explicit array of that type as the argument.
    - If no arguments are provided corresponding to the parameter, the result is an array of length 0.
- There can be at most one varargs parameter in a method declaration.
- The varargs parameter must be the last parameter in the method declaration.

For example, consider the following implementation of a `max()` method:

```
public int max(int... values) {
    int max = Integer.MIN_VALUE;
    for (int i: values) {
        if (i > max) max = i;
    }
    return max;
}
```

You can then invoke the `max()` method with any of the following:

```
max(1, -2, 3, -4);
max(1);
max();
int[] myValues = {5, -7, 26, -13, 42, 361};
max(myValues);
```

1.18. Constructors

- Constructors are like special methods that are called implicitly as soon as an object is instantiated (i.e. on `new ClassName()`).
    - Constructors have no return type (not even `void`).
    - The constructor name must match the class name.
- If you don't define an explicit constructor, Java assumes a default constructor
    - The default constructor accepts no arguments.
    - The default constructor automatically invokes its base class constructor with no arguments, as discussed later in this module.
- You can provide one or more explicit constructors to:
    - Simplify object initialization (one line of code to create and initialize the object)
    - Enforce the state of objects (require parameters in the constructor)
    - Invoke the base class constructor with arguments, as discussed later in this module.

- Adding *any* explicit constructor disables the implicit (no argument) constructor.

We can add a constructor to our Employee class to allow/require that it be constructed with a name and a social security number:

```
class Employee {
    String name;
    String ssn;
    …
    Employee(String name, String ssn) {
        this.name = name; // "this." helps distinguish between
        this.ssn = ssn; // instance and parameter variables
    }
    …
}
```

Then we can modify `EmployeeDemo.main()` to call the specified constructor:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee("John", "555-12-345");
        e1.emailAddress = "john@company.com";
        Employee e2 = new Employee("Tom", "456-78-901");
        e2.setYearOfBirth(1974);
        …
    }
}
```

1.19. Constructors (cont.)

- As with methods, constructors can be overloaded.
- Each constructor must have a unique signature.
    - The parameter type list must be different, either different number or different order.
    - Only parameter *types* determine the signature, not parameter *names*.
- One constructor can invoke another by invoking `this(`*`param1, param2, …`*`)` as the *first* line of its implementation.

It is no longer possible to do `Employee e = new Employee();` because there is no constructor that takes no parameters.

We could add additional constructors to our class Employee:

```
Employee(String ssn) { // employees must have at least a SSN
    this.ssn = ssn;
}

Employee(String name, String ssn) {
    this(ssn);
    this.name = name;
}
```

```
Employee(String name, String ssn, String emailAddress) {
    this(name, ssn);
    this.emailAddress = emailAddress;
}

Employee(String ssn, int yearOfBirth) {
    this(ssn);
    this.yearOfBirth = yearOfBirth;
}
```

Now we can construct Employee objects in different ways:

```
Employee e1 = new Employee("John", "555-12-345", "john@company.com");
Employee e2 = new Employee("456-78-901", 1974);
e2.name = "Tom";
```

1.20. Constants

- "Constant" fields are defined using the `final` keyword, indicating their values can be assigned only once.
    - Final instance fields must be initialized by the end of object construction.
    - Final static fields must be initialized by the end of class initialization.
    - Final local variables must be initialized only once before they are used.
    - Final method parameters are initialized on the method call.

Important

Declaring a reference variable as `final` means only that once initialized to refer to an object, it can't be changed to refer to another object. It does **not** imply that the state of the object referenced cannot be changed.

- Final static field can be initialized through direct assignment or by using a *static initializer*.
    - A static initializer consists of the keyword `static` followed by a block, for example:
    - `private static int[] values = new int[10];`
    - `static {`
    - `    for (int i = 0; i < values.length; i++) {`
    - `        values[i] = (int) (100.0 * Math.random());`
    - `    }`
    - `}`

If we declare `ssn` as final, we then must either assign it right away or initialize it in all constructors. This can also be done indirectly via constructor-to-constructor calls. Once initialized, final instance fields (e.g. `ssn`) can no longer be changed.

```
class Employee {
    final String ssn;
    …
    Employee(String ssn) {
```

```
        this.ssn = ssn;
    }
    …
}
```

Local variables can also be set as final, to indicate that they should not be changed once set:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        final Employee e1 = new Employee(…);
        final Employee e2 = new Employee("456-78-901", 1974);
        final Employee e3;
        e3 = e2;
        …
    }
}
```

1.21. Encapsulation

- The principle of *encapsulation* is that all of an object's data is contained and hidden in the object and access to it restricted to methods of that class.
    - Code outside of the object cannot (or at least should not) directly access object fields.
    - All access to an object's fields takes place through its methods.
- Encapsulation allows you to:
    - Change the way in which the data is actually stored without affecting the code that interacts with the object
    - Validate requested changes to data
    - Ensure the consistency of data — for example preventing related fields from being changed independently, which could leave the object in an inconsistent or invalid state
    - Protect the data from unauthorized access
    - Perform actions such as notifications in response to data access and modification

More generally, encapsulation is a technique for *isolating change*. By hiding internal data structures and processes and publishing only well-defined methods for accessing your objects,

1.22. Access Modifiers: Enforcing Encapsulation

- *Access modifiers* are Java keywords you include in a declaration to control access.
- You can apply access modifiers to:
    - Instance and static fields
    - Instance and static methods
    - Constructors
    - Classes
    - Interfaces (discussed later in this module)
- Two access modifiers provided by Java are:

    `private`

visible only within the same class
`public`
visible everywhere

Note

There are two additional access levels that we'll discuss in the next module.

1.23. Accessors (Getters) and Mutators (Setters)

- A common model for designing data access is the use of *accessor* and *mutator* methods.
- A mutator — also known as a *setter* — changes some property of an object.
  - By convention, mutators are usually named `set`*`PropertyName`*.
- An accessor — also known as a *getter* — returns some property of an object.
  - By convention, accessors are usually named `get`*`PropertyName`*.
  - One exception is that accessors that return a `boolean` value are commonly named `is`*`PropertyName`*.
- Accessors and mutators often are declared `public` and used to access the property outside the object.
  - Using accessors and mutators from code within the object also can be beneficial for side-effects such as validation, notification, etc.
  - You can omit implementing a mutator — or mark it `private` — to implement *immutable* (unchangeable) object properties.

We can update our Employee class to use access modifiers, accessors, and mutators:

```
public class Employee {
    private String name;
    private final String ssn;
    …
    public void setName(String name) {
        if (name != null && name.length() > 0) {
            this.name = name;
        }
    }

    public String getName() {
        return this.name;
    }

    public String getSsn() {
        return this.ssn;
    }
    …
}
```

Now, to set the name on an employee in `EmployeeDemo.main()` (i.e., from the outside), you must call:

```
e2.setName("Tom");
```

as opposed to:

```
e2.name = "Tom"; // won't compile, name is hidden externally
```

1.24. Inheritance

- *Inheritance* allows you to define a class based on the definition of another class.
    - The class it inherits from is called a *base class* or a *parent class*.
    - The derived class is called a *subclass* or *child class*.
- Subject to any access modifiers, which we'll discuss later, the subclass gets access to the fields and methods defined by the base class.
    - The subclass can add its own set of fields and methods to the set it inherits from its parent.
- Inheritance simplifies modeling of real-world hierarchies through generalization of common features.
    - Common features and functionality is implemented in the base classes, facilitating code reuse.
    - Subclasses can extended, specialize, and override base class functionality.

Inheritance provides a means to create specializations of existing classes. This is referred to as *sub-typing*. Each subclass provides specialized state and/or behavior in addition to the state and behavior inherited by the parent class.

For example, a manager is also an employee but it has a responsibility over a department, whereas a generic employee does not.

1.25. Inheritance, Composition, and Aggregation

- Complex class structures can be built through inheritance, composition, and aggregation.
- Inheritance establishes an "is-a" relationship between classes.
    - The subclass has the same features and functionality as its base class, with some extensions.
    - A Car **is-a** Vehicle. An Apple **is-a** Food.
- Composition and aggregation are the construction of complex classes that incorporate other objects.
    - They establish a "has-a" relationship between classes.
    - A Car **has-a** Engine. A Customer **has-a** CreditCard.
- In *composition*, the component object exists solely for the use of its composite object.
    - If the composite object is destroyed, the component object is destroyed as well.
    - For example, an Employee **has-a** name, implemented as a String object. There is no need to retain the String object once the Employee object has been destroyed.
- In *aggregation*, the component object can (but isn't required to) have an existence independent of its use in the aggregate object.
    - Depending on the structure of the aggregate, destroying the aggregate may or may not destroy the component.

- For example, let's say a Customer **has-a** BankAccount object. However, the BankAccount might represent a joint account owned by multiple Customers. In that case, it might not be appropriate to delete the BankAccount object just because one Customer object is deleted from the system.

Notice that composition and aggregation represent another aspect of encapsulation. For example, consider a Customer class that includes a customer's birthdate as a property. We could define a Customer class in such a way that it duplicates all of the fields and methods from and existing class like Date, but duplicating code is almost always a bad idea. What if the data or methods associated with a Date were to change in the future? We would also have to change the definitions for our Customer class. Whenever a set of data or methods are used in more than one place, we should look for opportunities to encapsulate the data or methods so that we can reuse the code and isolate any changes we might need to make in the future.

Additionally when designing a class structure with composition or aggregation, we should keep in mind the principle of encapsulation when deciding where to implement functionality. Consider implementing a method on Customer that would return the customer's age. Obviously, this depends on the birth date of the customer stored in the Date object. But should we have code in Customer that calculates the elapsed time since the birth date? It would require the Customer class to know some very Date-specific manipulation. In this case, the code would be *tightly coupled* –- a change to Date is more likely to require a change to Customer as well. It seems more appropriate for Date objects to know how to calculate the elapsed time between two instances. The Customer object could then *delegate* the age request to the Date object in an appropriate manner. This makes the classes *loosely coupled* –- so that a change to Date is unlikely to require a change to Customer.

1.26. Inheritance in Java

- You define a subclass in Java using the `extends` keyword followed by the base class name. For example:

```
class Car extends Vehicle { // ... }
```

  - In Java, a class can extend at most one base class. That is, multiple inheritance is not supported.
  - If you don't explicitly extend a base class, the class inherits from Java's Object class, discussed later in this module.
- Java supports multiple *levels* of inheritance.
  - For example, Child can extend Parent, which in turn extends GrandParent, and so on.

```
class A {
    String a = null;
    void doA() {
        System.out.println("A says " + a);
    }
}
class B extends A {
```

```
    String b = null;
    void doB() {
        System.out.println("B says " + b);
    }
}
class C extends B {
    String c = null;
    void doA() {
        System.out.println("Who cares what A says");
    }
    void doB() {
        System.out.println("Who cares what B says");
    }
    void doC() {
        System.out.println("C says " + a + " " + b + " " + c);
    }
}
public class ABCDemo {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();

        a.a = "AAA";
        b.a = "B's A";
        b.b = "BBB";
        c.a = "Who cares";
        c.b = "Whatever";
        c.c = "CCC";

        a.doA();
        b.doB();
        c.doA();
        c.doB();
        c.doC();
    }
}
```

The output of running ABCDemo is:

```
A says AAA
B says BBB
Who cares what A says
Who cares what B says
C says Who cares Whatever CCC
```

1.27. Invoking Base Class Constructors

- By default, Java automatically invokes the base class's constructor with **no** arguments before invoking the subclass's constructor.
    - This might not be desirable, especially if the base class doesn't have a no-argument constructor. (You get a compilation error in that case.)
- You can explicitly invoke a base class constructor with any arguments you want with the syntax `super(arg1, arg2, ...)`. For example:

```
class Subclass extends ParentClass {
    public Subclass(String name, int age) {
        super(name);
        // Additional Subclass initialization...
    }
}
```

- The call to `super()` must be the **first** statement in a subclass constructor.

Note

A single constructor cannot invoke both `super()` and `this()`. However, a constructor can use `this()` to invoke an overloaded constructor, which in turn invokes `super()`.

1.28. Overriding vs. Overloading

- The subclass can *override* its parent class definition of fields and methods, replacing them with its own definitions and implementations.
  - To successfully override the base class method definition, the subclass method must have the same signature.
  - If the subclass defines a method with the same name as one in the base class but a different signature, the method is overloaded not overridden.
- A subclass can explicitly invoked an ancestor class's implementation of a method by prefixing `super.` to the method call. For example:

```
class Subclass extends ParentClass {
    public String getDescription() {
        String parentDesc = super.getDescription();
        return "My description\n" + parentDesc;
    }
}
```

Consider defining a Manager class as a subclass of Employee:

```
public class Manager extends Employee {
    private String responsibility;

    public Manager(String name, String ssn, String responsibility) {
        super(name, ssn);
        this.responsibility = responsibility;
    }

    public void setResponsibility(String responsibility) {
        this.responsibility = responsibility;
    }

    public String getResponsibility() {
        return this.responsibility;
    }

    public void print(String header, String footer) {
        super.print(header, null);
```

```
        System.out.println("Responsibility: " + responsibility);
        if (footer != null) {
            System.out.println(footer);
        }
    }
}
```

Now with code like this:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        …
        Manager m1 = new Manager("Bob", "345-11-987", "Development");
        Employee.setBaseVacationDays(15);
        m1.setExtraVacationDays(10);
        …
        m1.print("BIG BOSS");
    }
}
```

The output is:

```
BIG BOSS
Name: Bob
SSN: 345-11-987
Email Address: null
Year Of Birth: 0
Vacation Days: 25
Responsibility: Development
```

Note that the Manager class must invoke one of super's constructors in order to be a valid Employee. Also observe that we can invoke a method like `setExtraVacationDays()` that is defined in Employee on our Manager instance `m1`.

1.29. Polymorphism

- *Polymorphism* is the ability for an object of one type to be treated as though it were another type.
- In Java, inheritance provides us one kind of polymorphism.
    - An object of a subclass can be treated as though it were an object of its parent class, or any of its ancestor classes. This is also known as *upcasting*.
    - For example, if Manager is a subclass of Employee:

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

    - Or if a method accepts a reference to an Employee object:
    - `public void giveRaise(Employee e) { // ... }`
    - `// ...`
    - `Manager m = new Manager(...);`
      `giveRaise(m);`

Why is polymorphism useful? It allows us to create more generalized programs that can be extended more easily.

Consider an online shopping application. You might need to accept multiple payment methods, such as credit cards, debit card, direct bank debit through ACH, etc. Each payment method might be implemented as a separate class because of differences in the way you need to process credits, debits, etc.

If you were to handle each object type explicitly, the application would be very complex to write. It would require `if-else` statements everywhere to test for the different types of payment methods, and overloaded methods to pass different payment type objects as arguments.

On the other hand, if you define a base class like PaymentMethod and then derive subclasses for each type, then it doesn't matter if you've instantiated a CreditCard object or a DebitCard object, you can treat it as a PaymentMethod object.

1.30. More on Upcasting

- Once you have upcast an object reference, you can access only the fields and methods declared by the base class.
    - For example, if Manager is a subclass of Employee:

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

    - Now using `e` you can access only the fields and methods declared by the Employee class.
- However, if you invoke a method on `e` that is defined in Employee but overridden in Manager, the Manager version is executed.
    - For example:

```
public class A {
    public void print() {
        System.out.println("Hello from class A");
    }
}
public class B extends A {
    public void print() {
        System.out.println("Hello from class B");
    }
}
// ...
A obj = new B();
obj.print();
```

      In the case, the output is "Hello from class B".

From within a subclass, you can explicitly invoke a base class's version of a method by using the `super.` prefix on the method call.

1.31. Downcasting

- An upcast reference can be *downcast* to a subclass through explicit casting. For example:
- `Employee e = new Manager(...);`
- `// ...`
  `Manager m = (Manager) e;`

    - The object referenced must actually be a member of the downcast type, or else a `ClassCastException` run-time exception occurs.
- You can test if an object is a member of a specific type using the `instanceof` operator, for example:

```
      if (obj instanceof Manager) { // We've got a Manager object }
public class EmployeeDemo {
    public static void main(String[] args) {
        final Employee e1 = new Employee("John", "555-12-345",
"john@company.com");
        final Employee e2 = new Employee("456-78-901", 1974);
        e2.setName("Tom");
        Employee em = new Manager("Bob", "345-11-987", "Development");

        Employee.setBaseVacationDays(15);
        e2.setExtraVacationDays(5);
        em.setExtraVacationDays(10);

        if (em instanceof Manager) {
            Manager m = (Manager) em;
            m.setResponsibility("Operations");
        }

        e1.print("COOL EMPLOYEE");
        e2.print("START OF EMPLOYEE", "END OF EMPLOYEE");
        em.print("BIG BOSS");
    }
}
```

This would print:

```
BIG BOSS
Name: Bob
SSN: 345-11-987
Email Address: null
Year Of Birth: 0
Vacation Days: 25
Responsibility: Operations
```

1.32. Abstract Classes and Methods

- An *abstract class* is a class designed solely for subclassing.
    - You can't create actual instances of the abstract class. You get a compilation error if you attempt to do so.

    - You design abstract classes to implement common sets of behavior, which are then shared by the *concrete* (instantiable) classes you derive from them.
    - You declare a class as abstract with the `abstract` modifier:

      ```
      public abstract class PaymentMethod { // ... }
      ```

- An *abstract method* is a method with no body.
    - It declares a method signature and return type that a concrete subclass must implement.
    - You declare a method as abstract with the `abstract` modifier and a semicolon terminator:

      ```
      public abstract boolean approveCharge(float amount);
      ```

    - If a class has any abstract methods declared, the class itself must also be declared as abstract.

For example, we could create a basic triangle class:

```
public abstract class Triangle implements Shape {
  public abstract double getA();
  public abstract double getB();
  public abstract double getC();
  public double getPerimeter() {
    return getA() + getB() + getC();
  }
  // getArea() is also abstract since it is not implemented
}
```

Now we can create concrete triangle classes based on their geometric properties. For example:

```
public class RightAngledTriangle extends Triangle {
  private double a, b, c;
  public RightAngledTriangle(double a, double b) {
    this.a = a;
    this.b = b;
    this.c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
  }
  public double getA() { return a; }
  public double getB() { return b; }
  public double getC() { return c; }
  public double getArea() { return (a * b) / 2; }
}
```

1.33. Interfaces

- An *interface* defines a set of methods, without actually defining their implementation.
    - A class can then *implement* the interface, providing actual definitions for the interface methods.

- In essence, an interface serves as a "contract" defining a set of capabilities through method signatures and return types.
    - By implementing the interface, a class "advertises" that it provides the functionality required by the interface, and agrees to follow that contract for interaction.

The concept of an interface is the cornerstone of object oriented (or modular) programming. Like the rest of OOP, interfaces are modeled after real world concepts/entities.

For example, one must have a driver's license to drive a car, regardless for what kind of a car that is (i.e., make, model, year, engine size, color, style, features etc.).

However, the car must be able to perform certain operations:

- Go forward
- Slowdown/stop (break light)
- Go in reverse
- Turn left (signal light)
- Turn right (signal light)
- Etc.

These operations are defined by an interface (contract) that defines what a car is from the perspective of **how it is used**. The interface does not concern itself with how the car is **implemented**. That is left up to the car manufacturers.

1.34. Defining a Java Interface

- Use the `interface` keyword to define an interface in Java.
    - The naming convention for Java interfaces is the same as for classes: CamelCase with an initial capital letter.
    - The interface definition consists of public abstract method declarations. For example:
    - ```
      public interface Shape {
          double getArea();
          double getPerimeter();
      }
      ```

- All methods declared by an interface are implicitly `public abstract` methods.
    - You can omit either or both of the `public` and `static` keywords.
    - You must include the semicolon terminator after the method declaration.

Rarely, a Java interface might also declare and initialize `public static final` fields for use by subclasses that implement the interface.

- Any such field must be declared **and** initialized by the interface.
- You can omit any or all of the `public`, `static`, and `final` keywords in the declaration.
      ```

1.35. Implementing a Java Interface

- You define a class that implements a Java interface using the `implements` keyword followed by the interface name. For example:

  ```
  class Circle implements Shape { // ... }
  ```

- A concrete class must then provide implementations for all methods declared by the interface.
  - Omitting any method declared by the interface, or not following the same method signatures and return types, results in a compilation error.
- An abstract class can omit implementing some or all of the methods required by an interface.
  - In that case concrete subclasses of that base class must implement the methods.
- A Java class can implement as many interfaces as needed.
  - Simply provide a comma-separated list of interface names following the `implements` keyword. For example:

    ```
    class ColorCircle implements Shape, Color { // ... }
    ```

- A Java class can extend a base class **and** implement one or more interfaces.
  - In the declaration, provide the `extends` keyword and the base class name, followed by the `implements` keywords and the interface name(s). For example:

    ```
    class Car extends Vehicle implements Possession { // ... }
    ```

```
public class Circle implements Shape {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }

    public double getPerimeter() {
        return Math.PI * this.radius * 2;
    }
}

/**
 * Rectange shape with a width and a height.
 * @author sasa
 * @version 1.0
 */
public class Rectangle implements Shape {
```

```
    private double width;
    private double height;

    /**
     * Constructor.
     * @param width the width of this rectangle.
     * @param height the height of this rectangle.
     */
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    /**
     * Gets the width.
     * @return the width.
     */
    public double getWidth() {
        return width;
    }

    /**
     * Gets the height.
     * @return the height.
     */
    public double getHeight() {
        return height;
    }

    public double getArea() {
        return this.width * this.height;
    }

    public double getPerimeter() {
        return 2 * (this.width + this.height);
    }
}

public class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }
}
```

1.36. Polymorphism through Interfaces

- Interfaces provide another kind of polymorphism in Java.
    - An object implementing an interface can be assigned to a reference variable typed to the interface.
    - For example, if Circle implements the Shape interface:

```
Shape s = new Circle(2);
```

- Or you could define a method with an interface type for a parameter:

```
public class ShapePrinter {
    public void print(Shape shape) {
        System.out.println("AREA: " + shape.getArea());
        System.out.println("PERIMETER: " + shape.getPerimeter());
    }
}
```

- When an object reference is upcast to an interface, you can invoke only those methods declared by the interface.

The following illustrates our shapes and ShapePrinter classes:

```
public class ShapeDemo {
    public static void main(String[] args) {
        Circle c = new Circle(5.0);
        Rectangle r = new Rectangle(3, 4);
        Square s = new Square(6);

        ShapePrinter printer = new ShapePrinter();
        printer.print(c);
        printer.print(r);
        printer.print(s);

    }
}
```

This would print:

```
AREA: 78.53981633974483
CIRCUMFERENCE: 31.41592653589793
AREA: 12.0
CIRCUMFERENCE: 14.0
AREA: 36.0
CIRCUMFERENCE: 24.0
```

1.37. `Object`: Java's Ultimate Superclass

- Every class in Java ultimately has the Object class as an ancestor.
    - The Object class implements basic functionality required by all classes.
    - Often you'll want to override some of these methods to better support your custom classes.
- Behaviors implemented by Object include:
    - Equality testing and hash code calculation
    - String conversion
    - Cloning
    - Class introspection
    - Thread synchronization
    - Finalization (deconstruction)

1.38. Overriding `Object.toString()`

- The `Object.toString()` method returns a String representation of the object.
- The `toString()` method is invoked automatically:
  - When you pass an object as an argument to `System.out.println(obj)` and some other Java utility methods
  - When you provide an object as a String concatenation operand
- The default implementation returns the object's class name followed by its hash code.
  - You'll usually want to override this method to return a more meaningful value.

For example:

```
public class Circle implements Shape {
    // ...
    public String toString() {
        return "Circle with radius of " + this.radius;
    }
}
```

1.39. Object Equality

- When applied to object references, the equality operator (==) returns `true` only if the references are to the same object. For example:

```
Circle a = new Circle(2);
Circle b = new Circle(2);
Circle c = a;
if { a == b } { // false }
if { a == c } { // true }
```

One exception to this equality principle is that Java supports *string interning* to save memory (and speed up testing for equality). When the `intern()` method is invoked on a String, a lookup is performed on a table of interned Strings. If a String object with the same content is already in the table, a reference to the String in the table is returned. Otherwise, the String is added to the table and a reference to it is returned. The result is that after interning, all Strings with the same content will point to the same object.

String interning is performed on String literals automatically during compilation. At run-time you can invoke `intern()` on any String object that you want to add to the intern pool.

1.40. Object Equivalence

- The Object class provides an `equals()` method that you can override to determine if two objects are equivalent.
  - The default implementation by Object is a simple == test.
  - You should include all `` `significant'' fields for your class when overriding `equals(). ``
  - We could define a simple version of `Circle.equals` as follows:
  - `public booleans equals(Object obj) {`
  - `    if (obj == this) {`

```
        o           // We're being compared to ourself
        o           return true;
        o       }
        o       else if (obj == null || obj.getClass() != this.getClass()) {
        o           // We can only compare to another Circle object
        o           return false;
        o       }
        o       else {
        o           // Compare the only significant field
        o           Circle c = (Circle) obj;
        o           return c.radius == this.radius;
        o       }
           }
```

A somewhat more complex example of an equality test is shown in this class:

```
public class Person {
    private String name;
    private final int yearOfBirth;
    private String emailAddress;

    public Person(String name, int yearOfBirth) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getYearOfBirth() {
        return this.yearOfBirth;
    }

    public String getEmailAddress() {
        return this.emailAddress;
    }

    public void setEmailAddress() {
        this.emailAddress = emailAddress;
    }

    public boolean equals(Object o) {
        if (o == this) {
            // we are being compared to ourself
            return true;
        } else if (o == null || o.getClass() != this.getClass()) {
            // can only compare to another person
            return false;
        } else {
            Person p = (Person) o; // cast to our type
            // compare significant fields
```

```
            return p.name.equals(this.name) &&
                p.yearOfBirth == this.yearOfBirth;
        }
    }

    public int hashCode() {
        // compute based on significant fields
        return 3 * this.name.hashCode() + 5 * this.yearOfBirth;
    }
}
```

1.41. Object Equivalence (cont.)

- Overriding `equals()` requires care. Your equality test must exhibit the following properties:
  - Symmetry: For two references, `a` and `b`, `a.equals(b)` if and only if `b.equals(a)`
  - Reflexivity: For all non-`null` references, `a.equals(a)`
  - Transitivity: If `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
  - Consistency with `hashCode()`: Two equal objects must have the same `hashCode()` value

Note

The `hashcode()` method is used by many Java Collections Framework classes like HashMap to identify objects in the collection. If you override the `equals()` method in your class, you **must** override `hashcode()` as well.

1.42 Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3 ×, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.

class Factorial {

  // this is a recursive method

  int fact(int n) {
```

```
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of fact( ) may seem a bit confusing. Here is how it works. When fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(n–1)*n. To evaluate this expression,

fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning.

To better understand how the fact( ) method works, let's go through a short example. When you compute the factorial of 3, the first call to fact( ) will cause a second call to be made with an argument of 2. This invocation will cause fact( ) to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of n in the second invocation). This result (which is 2) is then returned to the original invocation of fact( ) and multiplied by 3 (the original value of n). This yields the answer, 6. You might find it interesting to insert println( ) statements into fact( ), which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This is a very common error in working with recursion. Use println( ) statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method printArray( ) prints the first i elements in the array values.

```
// Another example that uses recursion.

class RecTest {
  int values[];
  RecTest(int i) {
    values = new int[i];
  }

  // display array -- recursively
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i−1) + "] " + values[i−1]);
  }
}

class Recursion2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;
```

```
    ob.printArray(10);
  }
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

1.43 Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the access modifier attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java's access modifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved. The other access modifiers are described next.

Let's begin by defining public and private. When a member of a class is modified by public, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. Now you can understand why main( ) has always been preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)

In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;

private double j;


private int myMethod(int a, char b) { //…To understand the effects of public and private access, consider the following program:


/* This program demonstrates the difference between

   public and private.

*/

class Test {

  int a; // default access
```

```
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
```

```
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

As you can see, inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods: setc( ) and getc( ). If you were to remove the comment symbol from the beginning of the following line,

// ob.c = 100; // Error!then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved versionof the Stack class shown at the end of Chapter 6.

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  /* Now, both stck and tos are private. This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;

  // Initialize top-of-stack
  Stack() {
```

```
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

As you can see, now both stck, which holds the stack, and tos, which is the index of the top of the stack, are specified as private. This means that they cannot be accessed or altered except through push( ) and pop( ). Making tos private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the stck array.

The following program demonstrates the improved Stack class. Try removing the commented-out lines to prove to yourself that the stck and tos members are, indeed, inaccessible.

```
class TestStack {

  public static void main(String args[]) {

    Stack mystack1 = new Stack();

    Stack mystack2 = new Stack();


    // push some numbers onto the stack

    for(int i=0; i<10; i++) mystack1.push(i);

    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack

    System.out.println("Stack in mystack1:");

    for(int i=0; i<10; i++)

      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");

    for(int i=0; i<10; i++)

      System.out.println(mystack2.pop());

    // these statements are not legal

    // mystack1.tos = -2;

    // mystack2.stck[3] = 100;

  }

}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

1.44. Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.

Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

• They can only directly call other static methods.

• They can only directly access static data.

• They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
```

}As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a static method from outside its class, you can do so using the following general form:

classname.method( )

Here, classname is the name of the class in which the static method is declared. As you can see, this format is similar to that used to call non-static methods through object-reference variables. A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside main( ), the static method callme( ) and the static variable b are accessed through their class name StaticDemo.

```
class StaticDemo {

  static int a = 42;

  static int b = 99;



  static void callme() {

    System.out.println("a = " + a);

  }

}



class StaticByName {

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

```
    StaticDemo.callme();

    System.out.println("b = " + StaticDemo.b);

  }

}
```
Here is the output of this program:

```
a = 42

b = 99
```

1.45. Introducing final

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```
final int FILE_NEW = 1;

final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;
```
Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of final is described in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its length instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.

class Length {

  public static void main(String args[]) {

    int a1[] = new int[10];

    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};

    int a3[] = {4, 3, 2, 1};



    System.out.println("length of a1 is " + a1.length);

    System.out.println("length of a2 is " + a2.length);

    System.out.println("length of a3 is " + a3.length);

  }

}
```

This program displays the following output:

```
length of a1 is 10

length of a2 is 8

length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the length member to good use in many situations. For example, here is an improved version of the Stack class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of stck.length is used to prevent the stack from overflowing.

1.46 Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```
// Demonstrate an inner class.

class Outer {

  int outer_x = 100;



  void test() {

    Inner inner = new Inner();

    inner.display();

  }
```

```
  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Output from this application is shown here:

display: outer_x = 100In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display( ) is defined inside Inner. This method displays outer_x on the standard output stream. The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method. That method creates an instance of class Inner and the display( ) method is called.

It is important to realize that an instance of Inner can be created only within the scope of class Outer. The Java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner. In general, an inner class instance must be created by an enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.

class Outer {

  int outer_x = 100;


  void test() {

    Inner inner = new Inner();

    inner.display();

  }


  // this is an inner class

  class Inner {

    int y = 10; // y is local to Inner


    void display() {

      System.out.println("display: outer_x = " + outer_x);

    }

  }


  void showy() {

    System.out.println(y); // error, y not known here!
```

```
  }
}


class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Here, y is declared as an instance variable of Inner. Thus, it is not known outside of that class and it cannot be used by showy( ).

Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a for loop, as this next program shows:

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
```

```
    }
    Inner inner = new Inner();
    inner.display();
  }
 }
}

class InnerClassDemo {
 public static void main(String args[]) {
  Outer outer = new Outer();
  outer.test();
 }
}
```
The output from this version of the program is shown here:

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

display: outer_x = 100While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 22. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about anonymous inner classes, which are inner classes that don't have a name.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.