Companion Chapter 4

Abstract Classes, Interfaces, and Generics

What You Will Learn in This Chapter

An overarching idea behind object-oriented (OO) programming is the idea of enabling more functionality and actions with less code written and minimizing complexity of logic. The usage of abstract classes, interface classes, and generic data types in Java applications aligns with this idea. Developers can leverage the strengths of class inheritance to ensure common functionality in a family or classes, and they can deploy common functionality across unrelated classes. With generic-type classes and methods, developers can provide flexibility for existing code to be compatible with multiple classes that exist now as well as for those that will be developed in the future. Flexibility is the key. Exploring advanced topics in OO development will help you write more flexible, loosely coupled, and more "intelligent" code that can adapt to data types used at runtime.

Specifically, this chapter will help you do the following:

1. Understand the difference between abstract classes and concrete classes
2. Understand the difference between interfaces and abstract classes and when each is appropriate to be used
3. Learn how to implement and use both abstract classes and interfaces
4. Understand the syntax rules for abstract and interface classes
5. Learn the basics of using generic-type interfaces from the Java API by using interface `Comparable`
6. Learn the basics of generic-type methods and how to use the generic-type parameter
7. Learn the basics of defining a generic-type class and how to use it in an application
8. Learn advanced generic-type concepts such as type parameter bounding and wildcard characters

Opening Scenario

After several client meetings, your team has elicited details about the data kept by the mom-and-pop grocery store and generated extensive information regarding their daily operations (stocking, receiving, personnel, advertising, store displays, shopper programs, etc.). The team has also developed an extensive set of requirements that the new system must implement. You recently met with several team members who work on just the modeling of the data in use in the client's organization and listened to their concerns: "This store sells everything, not just food: flowers; kitchen goods; bathroom cleaning supplies; pharmacy items, both prescription medicines and over-the-counter items; deli products; and even coffee shop items. We have a lot of entities to represent in the system!" One team member described her experiences with a prior team and the system it planned and built where everything was "hard coded" and "tied together." She clarified, "When the first version went online, it worked wonderfully. We had about forty class data types modeling the data in use by the client. After the first version went live, we moved on to phase 2 requirements, which saw an additional twenty class data types added. Integrating those into the current system was a nightmare! Everything was so tightly bound, classes using methods of other classes, that one change trickled throughout the system. We probably had ten changes for each of the twenty classes, and that spurred countless hours of work just fixing the classes. I won't even talk about

the work in the application code we did to accommodate all the new class types!" Hearing this, you hope that a phase 3 is not in their immediate future.

Having heard their concerns loud and clear and after a couple of sleepless nights thinking about that "nightmare" scenario, you decide to research some advanced object-oriented concepts that focus on maximizing the flexibility of code. Particularly, you are interested in class flexibility with existing code and the ability to model new data types later and have them drop into place with the existing system with minimal changes. Once again, helpful team members steer you toward the idea of "generic" classes, abstract classes, interfaces, and something called "generic-type classes." A quick web search shows code syntax completely new to you! Yet all the established advocates for programming, both within and without Java, tout the benefits of abstract classes, interfaces, and generics, stating that "future flexibility" is enabled by a "mastery of these topics." Exactly what you are looking for! You dig in again, hoping to understand these techniques in time to reassure your team that their experienced nightmare will not have a sequel . . .

CC4.1 Abstract and Concrete Classes

Note: This chapter assumes you are familiar with the topics in chapters 8 and 9.

In chapter 8, you learned the basics of creating new data types through the creation of user-defined classes. This yields many programming benefits: bundling actions with data, bundling the data itself through encapsulation, modeling real-world data and entity relationships through effective class design, and much more. Also recall that in chapter 9, you learned that a class can extend the functionality and data of a prior one, enabling compatibility of future class objects with code targeting objects of their superclass. This chapter will build on these ideas with advanced object-oriented (OO) topics that continue to increase the flexibility and usage of classes. "Flexibility" is most definitely the major theme of this chapter.

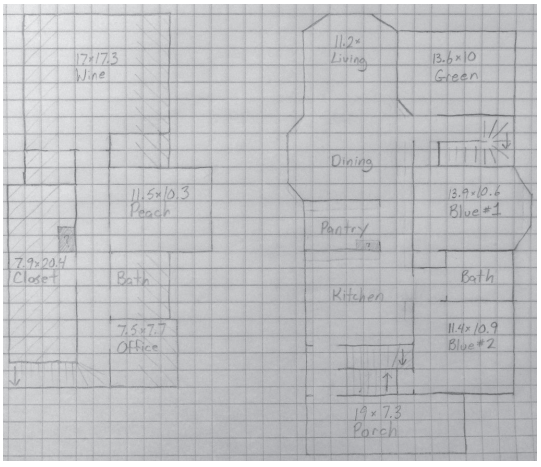

Figure CC4.1. Abstracted Layout Plan for a Home—Details to Be Filled in Later
Source: "16 Radcliffe Rough Layout" by sparr0 is licensed under CC BY-SA 2.0, https://www.flickr.com/photos/22611472@N02/8960537341.

Abstract classes in Java: The idea of an **abstract class** in Java is pretty simple. Recall that via class inheritance, a class can extend the data and the functionality of the parent class / superclass it inherits from. It can also override any inherited `public` methods and replace their functionality for invocations on its class or instance objects. With this in mind, consider the following situation:

- A parent class you have developed is inherited from several subclasses.
- Each of those classes will override one or more of the parent class's methods.
- Your parent class itself is rarely ever instantiated or has any of its methods invoked, as the subclasses are always more useful in their respective contexts.

So instead of attempting to provide a "default" implementation for these methods in your class, you can simply make your parent class **abstract**. An abstract class is considered "incomplete," as some of its code is missing! The developer for an abstract class will only provide **the method header** for certain methods and allow **future** subclasses to implement the code for those methods.

Consider the following example of a class called `Animal`:

Code Snippet CC4.1

```
public class Animal {
    private int age;
    private String skinType;

    public Animal(int age, String skinType)
    {
        this.age = age;
        this.skinType = skinType;
    }

    public int getAge()
    {
        return this.age;
    }

    public String getSkinType()
    {
        return this.skinType;
    }

    public String describeAnimal()
    {
        return "Age: " + this.age
                + ", Skin Type: " + this.skinType;
    }

} // End of class Animal
```

If you think about it, having a class `Animal` is great, but it is not entirely useful in all situations. Critical pieces of information are missing: the species, color, size, and so on. If a zoo was building an information system (IS) to manage the animals under its care, objects of this class would not be entirely useful. How could you generate a list of one particular type of animal? What about their feeding needs? Their range of motion? The developer of a zoo information systems has determined that this class will never be instantiated and will most likely be extended by subclasses that implement the useful information needed. So the developer decides to make this class **abstract** as in the following (changes are in **bold**):

Code Snippet CC4.2

```
public abstract class Animal {
    private int age;
    private String skinType;

    public Animal(int age, String skinType)
    {
        this.age = age;
        this.skinType = skinType;
    }

    public int getAge()
    {
        return this.age;
    }

    public String getSkinType()
    {
        return this.skinType;
    }

    public String describeAnimal()
```

```
    {
        return "Age: " + this.age
                + ", Skin Type: " + this.skinType;
    }

    // Abstract Method
    // Let future subclasses define it later.
    public abstract String howToMove(); // No code!

} // End of class Animal
```

Not much has changed. The keyword **abstract** has been added prior to the `class` keyword. Also, a method has been added that also has the same keyword `abstract` prior to its return type. Notice in particular the semicolon at the end of the method header for `.howToMove()`. This tells the compiler that no code body, no definition, will be provided for this method in **the current** class. Providing code for this method will be the responsibility of **future subclasses** of `Animal`.

Some details about abstract classes to keep in mind:

- Abstract classes **cannot** be instantiated. You cannot create instance objects of class `Animal`, for example. You are not able to use the `new` operator to create instances or arrays of instance references to them. You can create an array of their subclass types though. Consider the following hypothetical example:

  ```
  AbstractClass[] array = new AbstractClass[5];
  array[1] = new ConcreteClass();
  ```

  Though these are hypothetical classes, they show how an abstract class can be used polymorphically as a data type.
- An abstract class can contain zero, one, or more abstract methods. The rule here is that if you declare **any methods** to be abstract in a class, the class itself **must** also be abstract. On the flip side, an abstract class is **not** required to have any abstract methods.
- The fact that an abstract class can have abstract methods means that it is, by definition, incomplete. It will be up to future subclasses to "complete" its definition.
- In addition to abstract methods, abstract classes can contain all the "normal" parts of a class, including data fields (instance and static), constructors, and other fully defined methods.
- The abstract methods of the class **must** be marked with the `public` or `protected` access modifier. They cannot be marked `private` or `final`, which would prevent their full definition in future subclasses.
- Abstract classes are used in a class hierarchy with the assumption that they will **serve as a superclass to other classes**. They are also valid data types and can be used as method parameters, array data types, and so on anywhere class data types can be used.
- If your class extends an abstract class, the compiler forces you to provide concrete implementations of all the abstract methods inherited from the superclass.

The abstract method `.howToMove()` is a great example of abstract method usage. Trying to define how an "average" animal moves is not sensible: some walk, some fly, some crawl, some swim, some dig, some hop, and so on. It makes much more sense to make this method abstract and let it be specifically defined by a future subclass. Consider the following subclass of `Animal` called `Dog`:

Code Snippet CC4.3

```
public class Dog extends Animal {

    private String furColor;

    public Dog(int age, String skinType, String furColor)
    {
        super(age, skinType);
        this.furColor = furColor;
```

```
    }

    // Concrete implementation of howToMove() from
    //  abstract class Animal
    @Override
    public String howToMove()
    {
        return "Run and Play!";
    }

    @Override
    public String toString()
    {
        return "Dog (best friend) Fur Color: "
                + this.furColor + " "
                + super.describeAnimal();
    }
}
```

Notice that the superclass/subclass context still applies here: `Dog` can extend `Animal` and inherits `age` and `skinType` from the parent class (though they are private) and inherits the ability to call the superclass's constructor. Importantly, a **concrete** definition for the method `.howToMove()` has been provided in the `Dog` class. Remember the following:

- An abstract method consists of only a method header in a superclass.
- The method then is fully, concretely implemented in a subclass that inherits from the superclass, consisting of a method header and method code body.

Most dogs can move in the same manner, so it makes sense to provide a concrete implementation of `.howToMove()` here in class `Dog`. The developer of an abstract class leaves it up to other developers and their subclasses to do the work of providing a definition for the abstract methods of the class. Within an abstract class, there are only two method types that **cannot** be abstract:

- Constructors
- Static methods

Abstract classes allow the developer to define **what is common** across all the subclasses in the class inheritance hierarchy but allow each class to **define that commonality differently**. You can create two more classes that inherit from `Animal`—`Budgie` and `Fish`:

Code Snippet CC4.4

```
public class Budgie extends Animal{
    private String name;

    public Budgie(int age, String skinType, String name)
    {
        super(age, skinType);
        this.name = name;
    }

    @Override
    public String howToMove()
    {
        return "Fly!";
    }

    @Override
    public String toString()
    {
        return "Budgie Name: "
```

```
                + this.name
                + super.describeAnimal();
    }
}
```

The `Budgie` class (in North America, these are commonly called "parakeets") also provides a concrete definition of the `Animal` class's abstract method `.howToMove()` and defines how a budgie primarily moves: by flying! The `Fish` class does the same:

Code Snippet CC4.5

```
public class Fish extends Animal {

    private String species;

    public Fish(int age, String skinType, String species)
    {
        super(age, skinType);
        this.species = species;
    }

    @Override
    public String howToMove()
    {
        return "Swim and Glub Glub";
    }

    @Override
    public String toString()
    {
        return "Fish Species: "
                + this.species + " "
                + super.describeAnimal();
    }
}
```

The classes `Dog`, `Budgie`, and `Fish` are called **concrete** classes because they provide full definitions for the `Animal` class by filling in the abstract method gaps. Notice as well that when these subclasses define the `.howToMove()` method, the `abstract` keyword is removed: it is no longer abstract; you are concretely completing the definition!

In a Java `main()` class application, you can test out the polymorphic behavior that is still capable when using concrete classes that extend an abstract one:

Code Snippet CC4.6

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

    Budgie sleepy = new Budgie(3,"Feathers","SleepyBird");
    Dog caylee = new Dog(5,"Fur","Brindle");
    Fish bubbles = new Fish(1,"Scales","Goldfish");

    printAnimalDetails(sleepy);
    printAnimalDetails(caylee);
    printAnimalDetails(bubbles);

}

public static void printAnimalDetails(Animal ani)
{
    System.out.println(ani.toString() + "\n\t" + ani.howToMove());
}
```

This produces the following output:

```
Budgie Name: SleepyBird Age: 3, Skin Type: Feathers
   Fly!
Dog (best friend) Fur Color: Brindle Age: 5, Skin Type: Fur
   Run and Play!
Fish Species: Goldfish Age: 1, Skin Type: Scales
   Swim and Glub Glub
```

Even though these are concrete implementations of an abstract class, dynamic binding still applies as the method `.howToMove()` is invoked: Java looks at the data type of the object referenced by the variable `ani` and invokes the implementation of `.howToMove()` defined in that object's class. Figure CC4.2 shows the relationship between these classes as they stand so far.

Abstract classes are useful when you want to define a concept that represents some entity with a class but does not apply in a specific, concrete way. Subclasses derived from the abstract class provide the additional specifics, the concrete representation of an entity that falls under that context. An animal is a very generic class, but a dog is a specific type of animal with dog-specific characteristics. Same with fish and budgies.

You will find abstract classes used only in inheritance hierarchies. Once a class has decided to extend the abstract class, it has used up its one and only inheritance "ticket," since Java does not permit multiple class inheritance. The characteristics of class `Dog`, for example, will be either inherited from `Animal` or added by the developer. If state and actions from another class are desired within `Dog`, tough luck! Your class can only inherit from one class at a time. In the next section, You will see how to bypass inheritance and adopt actions from other classes that are **outside** the class inheritance hierarchy.

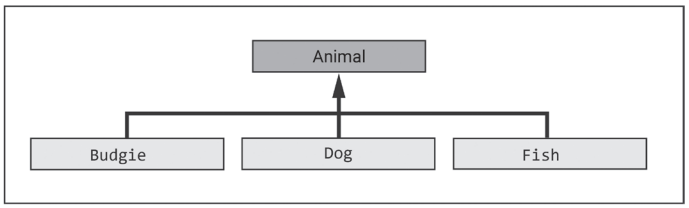

Figure CC4.2. The Relationship among the `Animal`, `Budgie`, `Dog`, and `Fish` Classes

SUMMARY POINTS

- An abstract class is a class with an incomplete implementation. It is expected that a future class that extends the abstract class will complete its implementation.
- The abstract keyword is added to both the class head and the header for any abstract methods defined in the abstract class. Each abstract method consists of only a header, a definition for which will be provided by a future subclass.
- An abstract class can act as a valid data-type reference variable, but abstract class data types themselves cannot be used to instantiate an instance object (i.e., used with the `new` operator).
- If a class extends an abstract class, it is forced by the compiler to provide concrete implementations of any abstract classes defined by the abstract class.

QUICK PROBLEMS

1. **Coding:** Create an array of size five that can hold objects of all three of the data types: `Dog`, `Budgie`, and `Fish`. Write a method that will accept a reference to the array and, for each object, invoke its `.howToMove()` method.
2. **Think:** How can ensuring common functionality across related classes be a good thing for developers?
3. **Coding:** Add an abstract method to the abstract class Animal called `.howToBathe()`. Implement the method in all subclasses, and return a description as you see fit. Then using your method from quick problem #1 earlier, invoke this new method along with `.howToMove()`.

CC4.2 Interface Classes

Abstract classes provide added flexibility to the developer who wants to create a class that generally describes some nonspecific entity. The developer can leave "the specifics" up to future subclasses to define. Flexibility is increased as well by ensuring common actions and functionality across related classes in an inheritance hierarchy. That is also a big weakness of abstract classes: the functionality is common only to related Java classes in an inheritance relationship. Class inheritance, though useful, is typically less common in contemporary development efforts because of the following:

- It increases the amount of **tight coupling** between classes in the inheritance hierarchy as well as other classes that use them.
- Making a **change** to the class hierarchy can be difficult. Imagine replacing the superclass with another: this will spur changes to (most likely all) classes in the hierarchy below it.
- Each class can only inherit from one other class, limiting the adoptable functionality of the classes in the hierarchy.
- Without an alternative, enabling polymorphic behavior among **classes not related to one another** cannot be done. This would sometimes be useful.
- Without an alternative, there is **no guarantee** that classes from separate inheritance hierarchies **have the same functionality**, preventing code already written to be flexibly adapted to work with other data types from other class families.

Figure CC4.3. Automobile Interface Abstracting Details of the Engine from the User
Source: "Classic Car Show" by A S Morton Image is licensed under CC BY 2.0, https://flickr.com/photos/purpleseadonkey/14834486929/.

Seeing these limitations, Java developers created a family of classes called **interfaces**. The biggest benefit of an interface class is that it allows common functionality to be adopted across multiple, **unrelated classes**. Secondly, a class can adopt, or **implement**, functionality from multiple interfaces. And developers who want to add this functionality to their classes that are already in a tightly coupled hierarchy can have those classes implement Java interfaces **without making changes to their classes' core functionality**. Interfaces bring a lot of benefits and of course added flexibility to your development efforts.

There are certain things to keep in mind in regard to Java interfaces versus abstract classes:

- Interfaces allow common functionality to be implemented across several unrelated classes, allowing code that expects this functionality in other classes to find it in yours as well.
- Unlike abstract classes, interface classes **can only** contain constant data field and abstract methods. They **cannot** contain regular data fields, constructors, or nonabstract methods.
- Similar to abstract classes, interface classes **cannot be instantiated** using the new keyword.
- Also similar to abstract classes, an **interface class can contain zero, one, or more abstract methods**. You will frequently see interface classes that contain no abstract methods called "marker" interfaces. `Serializable` in the Java API is a great example of this.*
- Classes use the `extends` keyword to inherit from an abstract class, whereas classes use the `implements` keyword to implement an interface.
- A class that implements an interface is **forced** to provide **concrete definitions** of all abstract methods declared within the interface, similar to the subclass of an abstract class.
- An interface can inherit from **another interface** if needed, using the `extends` keyword. A class that implements the subinterface must provide concrete definitions of all abstract methods declared by both interfaces.
- An interface class in Java **must** be placed in its own `.java` file. Its abstract methods must have the `public` access modifier.

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/Serializable.html.

- Implementation of an interface across multiple, unrelated classes enables the use of polymorphic code among those classes' objects, as the interface behaves just like a data type.
- A class can implement **multiple interfaces**, whereas it can only inherit from one single abstract class.
- If omitted by the developer, the compiler assumes that data field variables in an interface are modified with **public static final**, since interfaces can only contain constants and abstract methods.
- An interface is **implicitly** considered to be abstract also. The Java compiler will add the abstract keyword for you in front of the interface keyword if you omit it. You are not required to place it there. For example, you can create interface `Climbing` like so:

Code Snippet CC4.7

```
public abstract interface Climbing {
    public abstract double climbSpeed();
}
```

The compiler infers `abstract` on an interface class if not manually added. A key thing to remember here is that an interface is **not** also an abstract class but is considered to be abstract in its nature.

To demonstrate how interfaces in Java work, within a new `.java` file, you can type the following code that you will eventually use with the `Animal` class hierarchy exampled earlier:

Code Snippet CC4.8

```
public interface Feeding {

    public abstract String howToFeed();
}
```

For a Java interface, the `class` keyword is replaced with `interface`. Notice that there is only one abstract method defined in the interface, `.howToFeed()`. Where an abstract class is usually leveraged to represent some general, conceptual view of a real-world entity, interfaces are designed to focus on some sort of functionality or common ability that a class can support. Contemporary Java developers have increasingly demonstrated the term "coding to an interface," preferring the use of interfaces over expanded class hierarchies and complex inheritance relationships.

Using Java interfaces allows for **loose coupling** between class objects. When a class implements an interface, it "wraps" a very public and expected interaction ability around itself. Other classes can implement the same interface. Code that expects the functionality defined by the interface can expect to find it in all the classes that implement it and can interact with the objects of those classes. Loose coupling occurs because any code interacting with those class objects does not need to know anything about the methods or internals of those classes. When implementing an interface in your class, you are wrapping a "contractual" agreement around your class. Your class then provides a way for other objects to "interface" with your objects in a loosely coupled fashion. The big benefit here is that any changes made to the internals of your class and the way your objects work will not impact the way the interface provides other classes access to your objects. Very flexible! Figure CC4.4 visualizes this interface wrapping.

The developer who wants to add additional functionality to their class can choose to implement an interface and will not need to make any changes to any existing methods or to the class hierarchy the class sits within. Subclasses of the implementing class will simply inherit the concrete definitions of the abstract methods from the interface or can override them with their own definitions. This minimizes the number of changes needed to the classes in a class hierarchy when additional functionality/compatibility with other code is needed.

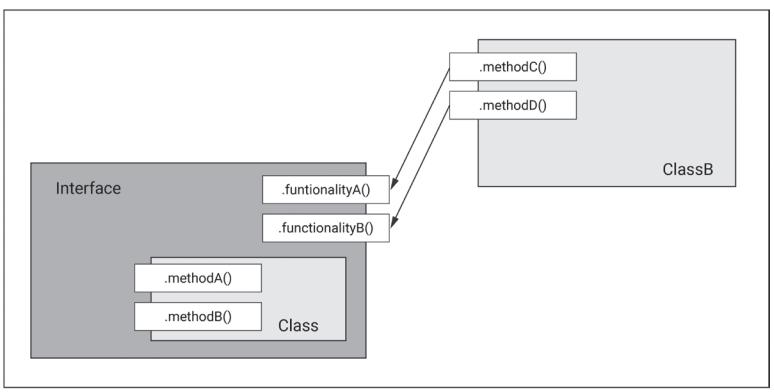

Figure CC4.4. Visualizing a Class with Implemented Interface

These changes can be demonstrated by implementing the `Feeding` interface in class `Budgie` from earlier in the chapter (changes in **bold**):

Code Snippet CC4.9

```
public class Budgie extends Animal implements Feeding {
    private String name;

    public Budgie(int age, String skinType, String name)
    {
        super(age, skinType);
        this.name = name;
    }

    @Override
    public String howToMove()
    {
        return "Fly!";
    }

    @Override
    public String toString()
    {
        return "Budgie Name: "
                + this.name
                + super.describeAnimal();
    }

    // Concrete definition for abstract method
    // from interface Feeding
    @Override
    public String howToFeed()
    {
        return "Budgie - " + this.name
                +  ": Seed, grit, and occasional "
                + "Spray Millet as a treat";
    }
}
```

Note that implementing interface `Feeding` simply requires the addition of the keyword `implements` followed by the interface name at the end of the class header. Note as well that providing the concrete definition for the interface's abstract method required no changes to either the inheritance from `Animal` or the method code that might cause issues for any potential subclasses of `Budgie`. Instance objects of class `Budgie` can self-report how these animals should be fed.

In particular, objects of classes that implement interfaces are able to share common functionality with objects of **unrelated classes**. Consider the creation of the two additional classes `Car` and `Plant` (ideally, you should place each in its own dedicated `.java` file if you are coding along):

Code Snippet CC4.10

```
// Class Car
public class Car implements Feeding {
    private String make;
    private String model;
    private int year;

    public Car(String make, String model, int year)
    {
        this.make = make;
        this.model = model;
        this.year = year;
```

```
    }

    @Override
    public String toString()
    {
        return "Make: " + this.make
                + ", Model: " + this.model
                + ", Year: " + this.year;
    }

    // Concrete implementation of Interface Method
    @Override
    public String howToFeed()
    {
        return this.make + " "
                + this.model
                + ": Fuel it with gasoline or electricity.";
    }
}


// Class Plant
public class Plant implements Feeding {
    private String species;
    private String location;

    public Plant(String species, String location)
    {
        this.species = species;
        this.location = location;
    }

    @Override
    public String toString()
    {
        return "Plant Species: " + this.species
                + ", Location to Grow: " + this.location;
    }

    @Override
    public String howToFeed()
    {
        return this.species + ": Feed with water, "
                + "sunshine, "
                + "and occasional Fertilizer";
    }
}
```

Since an interface can be used polymorphically as a data type, objects from unrelated classes can be used together and polymorphic actions performed upon them. Consider the following application code:

Code Snippet CC4.11

```
Budgie sweetie = new Budgie(2,"Feathers","SweetBaby");
Car forRunner = new Car("Caryoda","ForRunner", 2020);
Plant philo = new Plant("Philodendron","Indoor");

Feeding[] toFeedArray = new Feeding[5]; // Interface as data type
toFeedArray[0] = sweetie;
toFeedArray[1] = forRunner;
toFeedArray[2] = philo;
for (int i=0; i<toFeedArray.length;i++)
```

```
{
    if (toFeedArray[i] != null)
        System.out.println(toFeedArray[i].howToFeed());
}
```

When run, this code produces the following output:

```
Budgie - SweetBaby: Seed, grit, and occasional Spray Millet as a treat
Caryoda ForRunner: Fuel it with gasoline or electricity.
Philodendron: Feed with water, sunshine, and occasional Fertilizer
```

Can you feed a car? Maybe someday . . .

The real benefit here is in implementing common functionality across three unrelated class types (i.e., classes in separate class inheritance hierarchies). Implementing the functionality required minimal changes to the code of each class and did not threaten how those classes functioned internally. Because the array `toFeedArray` is declared with a data type of `Feeding`, you can say that the array **can contain references to classes that have implemented the `Feeding` interface**.

Using the `instanceof` operator: At compile time, the Java compiler will check to ensure that the method `.howToFeed()` is defined in the `Feeding` interface. You will notice as well that no methods of the actual classes themselves are available at compile time. Your code does not need to know anything about how the classes work internally to use them—the safe, external interface to the "public-facing code" is doing its job! If you needed to, as in any polymorphic context, you could cast each object to its actual data type. Consider this alteration to the code (changes in **bold**):

Code Snippet CC4.12

```
Budgie sweetie = new Budgie(2,"Feathers","SweetBaby");
Car forRunner = new Car("Caryoda","ForRunner", 2020);
Plant philo = new Plant("Philodendron","Indoor");

Feeding[] toFeedArray = new Feeding[5];
toFeedArray[0] = sweetie;
toFeedArray[1] = forRunner;
toFeedArray[2] = philo;
for (int i=0; i<toFeedArray.length;i++)
{
    if (toFeedArray[i] != null)
    {
        System.out.println(toFeedArray[i].howToFeed());

        if (toFeedArray[i] instanceof Budgie)
            System.out.println("\t"
                    + ((Budgie)toFeedArray[i]).toString());

        if (toFeedArray[i] instanceof Car)
            System.out.println("\t"
                    + ((Car)toFeedArray[i]).toString());

        if (toFeedArray[i] instanceof Plant)
            System.out.println("\t"
                    + ((Plant)toFeedArray[i]).toString());
    }
}
```

This will produce the following output when executed:

```
Budgie - SweetBaby: Seed, grit, and occasional Spray Millet as a treat
   Budgie Name: SweetBabyAge: 2, Skin Type: Feathers
Caryoda ForRunner: Fuel it with gasoline or electricity.
   Make: Caryoda, Model: ForRunner, Year: 2020
Philodendron: Feed with water, sunshine, and occasional Fertilizer
   Plant Species: Philodendron, Location to Grow: Indoor
```

Checking each array location to see if its object is an `instanceof` each of the classes is awkward. Worse yet, if objects from other classes that implement `Feeding` are used, then this code can get very tedious and defeat the goal of increased code flexibility. This code can be simplified by remembering two important points:

- The `Interface` class, like all other classes, inherits the `.toString()` method from the `Object` class. So you can invoke (i.e., see) `.toString()` on each of the `Interface` references at compile time.
- At runtime, dynamic binding kicks in and checks the actual data type of the object actually referenced from each array location. Because each object has a `.toString()` method implemented in each of its classes, dynamic binding causes that implemented class version to be executed.

Remembering these two points, the `instanceof` code checks can be eliminated, this code can be simplified, and it will produce the same output as seen earlier:

Code Snippet CC4.13

```
Budgie sweetie = new Budgie(2,"Feathers","SweetBaby");
Car forRunner = new Car("Caryoda","ForRunner", 2020);
Plant philo = new Plant("Philodendron","Indoor");

Feeding[] toFeedArray = new Feeding[5];
toFeedArray[0] = sweetie;
toFeedArray[1] = forRunner;
toFeedArray[2] = philo;
for (int i=0; i<toFeedArray.length;i++)
{
    if (toFeedArray[i] != null)
    {
        System.out.println(toFeedArray[i].howToFeed());
        System.out.println("\t" + toFeedArray[i].toString());
    }
}
```

Implementing multiple interfaces: Unlike with class inheritance, Java classes can implement multiple interfaces. Recall the `Climbing` interface displayed earlier (this time leaving off the `abstract` keyword):

Code Snippet CC4.14

```
public interface Climbing {

    public abstract double climbSpeed();

}
```

To implement an additional interface in a class, simply follow the class's first implemented interface with a comma and the name of the additional. Here is the complete listing for the `Budgie` class (including concrete interface method definition for `.climbSpeed()`):

Code Snippet CC4.15

```
public class Budgie extends Animal implements Feeding, Climbing {
    private String name;

    public Budgie(int age, String skinType, String name)
    {
        super(age, skinType);
        this.name = name;
    }

    @Override
    public String howToMove()
    {
        return "Fly!";
```

```
    }

    @Override
    public String toString()
    {
        return "Budgie Name: "
                + this.name
                + super.describeAnimal();
    }

    // Concrete definition for abstract method
    // from interface Feeding
    @Override
    public String howToFeed()
    {
        return "Budgie - " + this.name
                +  ": Seed, grit, and occasional "
                + "Spray Millet as a treat";
    }

    // In class Budgie
    // Concrete method for Climbing interface
    @Override
    public double climbSpeed()
    {
        return 10.0 - (this.getAge() * 0.3);
    }
}
```

The concepts of climbing and feeding are two different activities. You can implement the same methods across `Car` and `Plant` as well (add the new interface for `Climbing` to your classes and add the method to implement `.climbSpeed()` to your classes as well):

Code Snippet CC4.16

```
public class Car implements Feeding, Climbing {

. . .

    // In class Car
    // Concrete method for Climbing interface
    @Override
    public double climbSpeed()
    {
        return 40.0;
    }
}

. . .

public class Plant implements Feeding, Climbing {

. . .

    // In class Plant
    // Concrete method for Climbing interface
    @Override
    public double climbSpeed()
    {
        return 0.04;
    }
}
```

The multiple interfaces implemented in a class do not have to be related to one another through any hierarchy or even conceptually. In this manner, you can add many actions to a class's objects with minimal coding and impact on those classes. Figure CC4.5 shows the structure of the class and interface relationships so far.

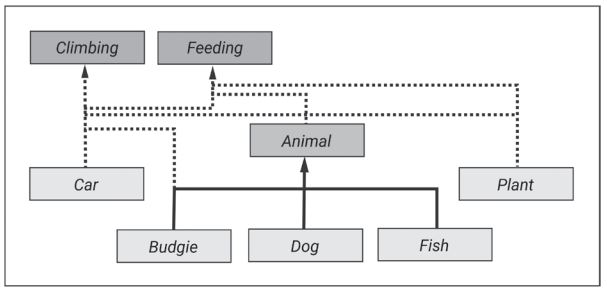

Figure CC4.5. Relationship of Classes with Implemented Interfaces

SUMMARY POINTS

- An interface class allows a developer to define functionality that is common across both related and, importantly, unrelated classes.
- Interface classes in Java contain only constant data fields and abstract methods.
- Whereas a class can only extend one other class, it can implement one or many separate interfaces.
- A class that implements a Java interface is forced to provide concrete implementations of any abstract methods defined in the interface.
- Interfaces can be used as valid data types in reference variables but cannot be used to instantiate any instance objects (i.e., used with the `new` operator).
- It is common to see interfaces with no abstract methods or constants at all. These are commonly called "marker" interfaces.
- Interface classes in Java must be placed in their own `.java` file.

QUICK PROBLEMS

1. **Coding:** Write the code for a small class `Person` that implements the `Feeding` interface. Instantiate an object of `Person`, add it to the `toFeedArray` array in the earlier example, and observe the output.
2. **Think:** Why might the developer of these classes not have implemented the Climbing interface in the classes for `Dog` and `Fish`?
3. **Coding:** Add a new abstract method into the `Feeding` interface called `howMuchToFeed()`. Save your changes to the class. What errors/warnings appear in the other classes that implement `Feeding` afterward?

CC4.3 Generic Interfaces, Classes, and Methods

Implementing an interface from the Java API: The Java language comes with several interfaces that can be implemented in classes to ensure object compatibility with code elsewhere in the Java language's many classes. A common example deals with sorting. The `Comparable` interface is used to give your class the ability for its objects to be compared with one another in some aspect.* Since `Comparable` is an interface, you will decide in your user-defined class how this comparison will happen via your concrete implementation of this interface's only abstract method. Here is the entire source code for interface `Comparable`:

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html.

Code Snippet CC4.17

```
public interface Comparable<T> {

    int compareTo(T o);

}
```

This interface happens to be a **generic-type interface**, meaning that when implemented, the data type for the class objects that comparisons will occur between is provided, replacing the `T` generic type. The Oracle documentation recommends developers implement their `.compareTo()` method to return the following values:

- A value **less than zero** if your object is compared to be less in some value than the object you are comparing it to
- A value **greater than zero** if your object is compared to be greater in some value than the object you are comparing it to
- A value **exactly equal to zero** if the value of your object and the value of the object you are comparing against are exactly the same (or the "same" in whatever way you define it)

To implement this interface, consider the following updated code for class `Budgie` (additions in **bold**):

Code Snippet CC4.18

```
public class Budgie
        extends Animal
        implements Feeding, Climbing, Comparable<Budgie> {

    private String name;

    public Budgie(int age, String skinType, String name)
    {
        super(age, skinType);
        this.name = name;
    }

    @Override
    public String howToMove()
    {
        return "Fly!";
    }

    @Override
    public String toString()
    {
        return "Budgie Name: "
                + this.name + " "
                + super.describeAnimal();
    }

    // Concrete definition for abstract method
    // from interface Feeding
    @Override
    public String howToFeed()
    {
        return "Budgie - " + this.name
                +  ": Seed, grit, and occasional "
                + "Spray Millet as a treat";
    }

    // In class Budgie
    // Concrete method for Climbing interface
```

```
    @Override
    public double climbSpeed()
    {
        return 10.0 - (this.getAge() * 0.3);
    }

    // Concrete implementation for compareTo()
    // Interface Comparable
    @Override
    public int compareTo(Budgie o)
    {
        if (this.getAge() < o.getAge())
            return -1; // This object's age is less
        else if (this.getAge() > o.getAge())
            return 1; // This objects age is greater
        else
            return 0; // This object's age is the same as parameter's
    }
}
```

Notice in the class header that `Comparable` is an additionally implemented interface. Because it is a flexible, generic-type interface, the data type of `Budgie` is provided. This allows the use of the `Budgie` data type as the parameter's data type for the concrete implementation of the `.compareTo()` method. Remember, if you implement an interface in your class, you are forced to provide concrete implementations for all the interface's abstract methods!

To take advantage of this interface implementation, consider the following application code that first simply creates five objects, places them in a generic-type `ArrayList`, and then prints out returned String from each `.toString()` invocation:

Code Snippet CC4.19

```
// Add import for java.util.* to your application class
ArrayList<Budgie> parrotList = new ArrayList<>();

parrotList.add(new Budgie(2, "Feathers", "Pretty Bird"));
parrotList.add(new Budgie(4, "Feathers", "Sassy Squawk"));
parrotList.add(new Budgie(6, "Feathers", "Old Boy"));
parrotList.add(new Budgie(1, "Feathers", "Pip Squeak"));
parrotList.add(new Budgie(5, "Feathers", "Sweet Beak"));

for (Budgie b: parrotList)
{
    System.out.println(b);
}
```

This will produce the following output:

```
Budgie Name: Pretty Bird Age: 2, Skin Type: Feathers
Budgie Name: Sassy Squawk Age: 4, Skin Type: Feathers
Budgie Name: Old Boy Age: 6, Skin Type: Feathers
Budgie Name: Pip Squeak Age: 1, Skin Type: Feathers
Budgie Name: Sweet Beak Age: 5, Skin Type: Feathers
```

The `Collections` class contains several static methods to perform various data operations on collections in Java (similar to how the `Arrays` class in chapter 7 helped perform data operations on arrays).* The `ArrayList<E>` **class** implements the `Collection` **interface,** which means it can be operated upon by the methods in the

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html.

`Collections` **class** (notice here a great reinforcement in your learning: implementing interfaces can make your class compatible with preexisting code that expects that interface's methods to be present).*

You can use the static `.sort()` method of the `Collections` class to sort the contents of the `ArrayList<E>` called `parrotList`. Printed here is the signature of the `.sort()` method from Oracle's official documentation:

```
Static <T extends Comparable<? super T>> void sort(List<T> list)
```

The statements in brackets will be described later. For now, focus on the parameter for the `.sort()` method. `List<T>` is another generic-type interface. So the usage of `.sort()` requires that you provide an object for the parameter whose class has implemented interface `List<T>`. Luckily for us, `ArrayList<E>` has implemented the interface `List<T>`. You can sort these objects by doing the following (additional code in **bold**):

Code Snippet CC4.20

```
// Add import for java.util.* to the application
ArrayList<Budgie> parrotList = new ArrayList<>();
parrotList.add(new Budgie(2, "Feathers", "Pretty Bird"));
parrotList.add(new Budgie(4, "Feathers", "Sassy Squawk"));
parrotList.add(new Budgie(6, "Feathers", "Old Boy"));
parrotList.add(new Budgie(1, "Feathers", "Pip Squeak"));
parrotList.add(new Budgie(5, "Feathers", "Sweet Beak"));

for (Budgie b: parrotList)
{
    System.out.println(b);
}

Collections.sort(parrotList); // invoke .sort()
System.out.println("\n");

// Print the list again . . .
for (Budgie b: parrotList)
{
    System.out.println(b);
}
```

This will now produce the following expanded output:

```
Budgie Name: Pretty Bird Age: 2, Skin Type: Feathers
Budgie Name: Sassy Squawk Age: 4, Skin Type: Feathers
Budgie Name: Old Boy Age: 6, Skin Type: Feathers
Budgie Name: Pip Squeak Age: 1, Skin Type: Feathers
Budgie Name: Sweet Beak Age: 5, Skin Type: Feathers


Budgie Name: Pip Squeak Age: 1, Skin Type: Feathers
Budgie Name: Pretty Bird Age: 2, Skin Type: Feathers
Budgie Name: Sassy Squawk Age: 4, Skin Type: Feathers
Budgie Name: Sweet Beak Age: 5, Skin Type: Feathers
Budgie Name: Old Boy Age: 6, Skin Type: Feathers
```

Notice that with the second print, the `ArrayList<E>` has been sorted! The `Budgie` objects now appear in the list sorted in order of age ascending. The `.sort()` method invokes the `.compareTo()` method that you concretely implemented in the `Budgie` class, comparing each `Budgie` instance object to another. For each comparison, the `.compareTo()` method returned values of `-1`, `0`, or `1` based on ages that are less than, equal to, or greater than the compared objects, respectively. This enabled the `.sort()` method to sort the list based on each object's `age` data field. Very convenient! Other interfaces in the Java API include ones like `Cloneable` and `Serializable` as well as ones covered in companion chapter 5 like `List`, `Set`, and `Queue`.

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html.

Defining generic methods: Now that you have used generic syntax like that found in `ArrayList<E>` and `Comparable<T>`, **generic typing** can be elaborated upon. In the earlier examples, you may have noticed a letter `T` placed where a data-type name would normally be expected. This is called a **generic-type parameter** and is what enables methods of classes that are generically typed to have flexibility in terms of the data types of objects they can accept. For example, if you wanted to create a generic-type method that would invoke the `.toString()` of the objects passed to it, you could do so as follows (generic-type code in **bold**):

Code Snippet CC4.21

```
public static <T> void printObjectDetails(T obj)
{
    System.out.println(obj);
}
```

The type parameter must be specified in angle brackets (<>) prior to the return type of the method. Then the type parameter can be used without the brackets in the parameter of the method to specify the data type of the accepted reference variable. You can use the type parameter for as many parameters as you need:

Code Snippet CC4.22

```
public static <T> void someMethod(T obj, T objTwo)
{
    // Code to act on the objects.
}
```

Also, you can specify multiple type parameters with separate letters to indicate that separate data types should be used for each:

Code Snippet CC4.23

```
public static <T, S> void anotherMethod(T obj, S objTwo)
{
    // Code to act on the objects.
}
```

Going back to the definition of the `printObjectDetails()` generic method earlier, you can test it out like so:

Code Snippet CC4.24

```
printObjectDetails(new Budgie(2, "Feathers", "Nice Bird"));
printObjectDetails(new Dog(4, "Fur", "Spottled"));
printObjectDetails(new String("Hello there!"));
```

The output when this application code is executed is the following:

```
Budgie Name: Nice Bird Age: 2, Skin Type: Feathers
Dog (best friend) Fur Color: Spottled Age: 4, Skin Type: Fur
Hello there!
```

In each case, a reference to an actual data-type object is provided for the parameter when the method is invoked. You would never actually try to provide a `T`. Some things to remember with type parameters:

- A type parameter can consist of a word, but programming convention specifies a **single, uppercase letter**.
- Various letters are used by convention: **T** for a data type; **E** for an element of a data sequence; **K** for a key (as in a Map or Hash); **V** for a value; and **S**, **U**, and **V** for other normal data types if additional are needed.
- In all cases, the instance object of a complex data type **must** be used when you provide a parameter to replace the generic in the usage. In the case of primitive data types, the wrapper class versions of them will **autobox** these values: `Integer` for `int`, `Double` for a simple `double`, and so on.
- A concrete class must be provided as a substitute for the type parameter at compile time.
- You **cannot** use a generic-type parameter in a static context, like with a static method or static data field (this is due to the way the Java compiler resolves data-type names at compile time vs. runtime).
- You cannot use a type parameter with the new operator to create an instance object or use it to create an array. For example, the following uses are not allowed:

```
E aVar = new E();            // Not allowed
E[] anArray = new E[10];     // Not allowed
```

Bounding the type parameter and using wildcard type parameters: To ensure that things work like they should, developers of generic methods can "bound" the type parameter to a range of classes. Consider the `Budgie` class that has implemented the `Comparable<T>` interface. Suppose you want to restrict the objects that can be added to an `ArrayList<E>` to ensure that they have implemented the `Comparable<T>` interface and that only `Budgie` objects are passed to it and used in the `ArrayList<E>`. Consider the following generic method along with application code that uses it:

Code Snippet CC4.25

```
// Application Code
ArrayList<Budgie> birdList = new ArrayList<>();
Budgie aBird = new Budgie(4, "Downy", "Baby Bird");

addToArray(aBird, birdList);

System.out.println(birdList.get(0));

. . .

// Generic Application Method
public static <T extends Comparable<T>>
        void addToArray(T obj, ArrayList<T> array)
{
    array.add(obj);
}
```

This produces the following output:

```
Budgie Name: Baby Bird Age: 4, Skin Type: Downy
```

Notice that the data-type parameter in brackets before the method's return type can be thought of as a "rule" that specifies what ranges of data types will be allowed to be substituted for the type parameter when the method is used. Remember that `Comparable<T>` is a generic-type interface that is implemented in class `Budgie` by giving it the `Budgie` data type: `Comparable<Budgie>`. The data-type parameter `<T extends Comparable<T>>` simply says, "Only data types that have implemented `Comparable` for their data types will be accepted as a substitute for `T` in the parameter list of this method." What has been done here is to restrict the `T` parameter by setting an **upper bound** upon it. Only data types from `Comparable<T>` and extended "downward" will be accepted as a substitute. Every type parameter has an upper bound on it, even if one is not provided. For example, if you see

```
<T>
```

used in a generic method, this has an implied upper bound of `Object`, equivalent to the following:

```
<T extends Object>
```

Type parameters can be specified with a **wildcard** character as well. Consider the following: `Budgie` might be a subclass of abstract class `Animal`, but due to the way the Java compiler resolves data types at compile time, `ArrayList<Budgie>` is not a subclass of `ArrayList<Animal>`. In order to specify that an `ArrayList<E>` can be used for either of these, you can use an upper-bound wildcard character:

Code Snippet CC4.26

```
public static void printListLength(ArrayList<? extends Animal> list)
{
   // Code for method.
}
```

You will use wildcard characters inside the type parameter brackets for generic-type class data types, like `ArrayList<E>` or `Map<K,V>`, for example. In the generic-type method, `<T extends Animal>` would act the same way for a direct parameter generic data type, as seen earlier.

- *Note:* The use of a **lower-bound** wildcard type parameter character is valid in Java but is commonly thought to not be too useful. For example,

```
<? super Animal>
```

would not be too useful, as the only class above `Animal` is `Object`. You will not see these used too often. The use of a lower bound with a regular type parameter is not allowed.

Generic-type classes: Class definitions can take advantage of the flexibility of generic types as well. Consider the following definition for a class `Owner` that can take "possession" of objects of any of the classes you have implemented so far in the chapter:

Code Snippet CC4.27

```
public class Owner<T> {

    private T object;
    private String name;

    public Owner(T object, String name)
    {
        this.object = object;
        this.name = name;
    }

    public String toString()
    {
        return "Owner: " + this.name
                + ", Owns: "
                + this.object.toString();
    }
}
```

Note that upon usage and just like with the methods, it uses **erasure** to get rid of the generic-type parameter `T` and replace it with the actually provided type parameter at runtime. You can test class `Owner` like so:

Code Snippet CC4.28

```
Budgie sweetie = new Budgie(2,"Feathers","SweetBaby");
Car forRunner = new Car("Caryoda","ForRunner", 2020);
Plant philo = new Plant("Philodendron","Indoor");

Owner<Budgie> birdOwner = new Owner<>(sweetie, "Sarah");
Owner<Car> carOwner = new Owner<>(forRunner, "Julie");
Owner<Plant> plantOwner = new Owner<>(philo, "Adara");

System.out.println(birdOwner.toString() + "\n"
        + carOwner.toString()
        + "\n" + plantOwner.toString());
```

This application code will produce the following output:

```
Owner: Sarah, Owns: Budgie Name: SweetBaby Age: 2, Skin Type: Feathers
Owner: Julie, Owns: Make: Caryoda, Model: ForRunner, Year: 2020
Owner: Adara, Owns: Plant Species: Philodendron, Location to Grow: Indoor
```

Notice that since the `Owner<>` constructor requires two parameters, a data type compatible for `T` for the object owned and the name of the owner, you provide these at usage. The ability to upper-bound the data-type parameter applies to a class as well. Consider the following `ZooCareGiver` class:

Code Snippet CC4.29

```
public class ZooCareGiver<T extends Animal> {

    private T animal;
    private String careGiverName;

    public ZooCareGiver (T animal, String careGiverName)
    {
        this.animal = animal;
        this.careGiverName = careGiverName;
    }

    public String toString()
    {
        return "Caregiver: " + this. careGiverName
                + ", Cares for: "
                + this.animal.toString();
    }
}
```

In this way, you can ensure that a `Car` is not accidentally admitted into the zoo and placed into an enclosure. They generally do not like to be cooped up like that; they get temperamental.

SUMMARY POINTS

- Many interfaces and classes in the Java API are generically typed, meaning that they can process objects of many different data types using only one implementation of code. Examples are `Comparable<T>` and `ArrayList<E>`.
- Generic-type classes and methods use a type parameter to indicate the rule or allowance for what range of data types are allowed to be used as a substitute for the parameter.
- An object of a concrete Java class must be used in place of a type parameter in generic-type code.
- General convention calls for a single uppercase letter to be used to represent a generic-type parameter. `T` (for a data type) is commonly used along with `E` (an element of a sequence of a particular data type).
- Generic-type parameters cannot be used to instantiate objects (i.e., used with the `new` operator).
- Bounding a type parameter specifies a range of concrete data types that are allowed for substitution. Typically, upper bounds are much more common and use the `extends` keyword for the definition.
- When a generic-type class or interface data type is used as a parameter in generic-type code, the wildcard character `?` can be used to specify the bounding within its generic typing brackets.

QUICK PROBLEMS

1. **Coding:** Write a small class called `Feeder` that is generically typed to accept a parameter type of any class that has implemented the `Feeding` interface. Accept a `String name` and `String PhoneNumber` for each `Feeder` object. Include a method that prints the `Feeder` object's details along with the return of the `.howToFeed()` method from the `T` object's concrete implementation.
2. **Think:** How does generic typing of methods and classes provide future flexibility and future compatibility of code/logic written today?
3. **Coding:** Implement the `Comparable` interface for the `Car` class. Test the usage out by adding several `Car` objects to an `ArrayList<E>` and sorting the contents based on vehicle year.

Summary

In this chapter, you have learned the basics of using abstract classes, interface classes, and generic-type code in Java. With all three of these syntax techniques, the idea is to increase the flexibility of Java code so it can work with new data types without the need for a major overhaul of the logic. By implementing these techniques correctly, your code can be "future-proofed" to work with classes that may not exist until far in the future. Since one of object-oriented programming's many goals is to help write applications with more functionality in less code, these techniques are a must-learn for any developer wishing to come anywhere near accomplishing this goal in their work. For information systems professionals, the flexibility these techniques bring to modeling current data and adding new entities is extremely valuable. Their value to you can be assured in the fact that not many information systems students study these topics in-depth. With these tools under your belt, you will find yourself thinking far ahead in your data model and application design, resulting in far more sophisticated, flexible, and intelligent Java applications

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

1. Class inheritance	a. Relationship between two classes where one uses the other and depends on the internal interface of that class. Changes to the used class can impact the functionality of the using class.
2. Abstract class	b. Keyword used in the Java syntax to indicate an inheritance relationship between two classes.
3. Superclass	c. Flexibility provided in both methods and classes where the data type used by each can be determined at runtime via application code.
4. Subclass	d. Generic-type interface in the Java API that provides the ability to relate two objects using some characteristic or aspect of each.
5. Concrete definition	e. Syntax that symbolically defines which data types will be "flexible" in a generic-type method or class.
6. Abstract method	f. A runtime operation where the Java compiler replaces generic-type parameters with the actual data type provided by the application's code.
7. Tight coupling	g. In a class inheritance relationship, this class provides public members and methods to other classes that inherit from it.
8. Interface class	h. Automatic operation by the Java compiler that "wraps" a primitive data type in its complex data-type equivalent class.
9. Implicit abstract	i. Generic-type class in the Java API that provides a dynamically resizable array whose elements can contain any data type provided to it by the application code at runtime.
10. `extends`	j. Specifying a range of data types that fall in a class relationship hierarchy that are allowed for use when substituting for generic-type parameters.
11. `implements`	k. Specifying a range of data types that fall in a class relationship hierarchy that are allowed for use when substituting for generic-type parameters in a preexisting generic-type class or interface.
12. Loose coupling	l. Relationship between two classes where one uses another class's adopted interface instead of directly accessing that class's internal methods. Changes to the target class's internals will not impact the functionality of the using class.
13. Generic typing	m. Class with an "incomplete" implementation, where some methods are missing a full definition that will be provided by future, inheriting subclasses.
14. `Comparable<T>`	n. The implied nature of an interface class, provided if the developer omits the `abstract` keyword from the interface header.
15. Type parameter	o. Keyword used in the Java syntax to indicate the adoption of an interface definition in a class.

16. `ArrayList<E>`	p. A full definition for a class method is provided to fill in the gaps present from an inherited class's method definition.
17. Erasure	q. Technique where a class adopts the public members and methods of another class through a parent/child class relationship.
18. Autoboxing	r. Class that contains only constants and abstract methods. Intended to provide functional commonality across both related and unrelated classes.
19. Bounding	s. In a class inheritance relationship, this class adopts the public members and methods of another class higher than it in the hierarchy.
20. Wildcard character	t. Member method of a class that is missing a code body, causing its definition to be incomplete.

Find the Error

In each of the following problems, carefully examine the code given, and determine the error(s)/issue(s) with each. Keep in mind, the error(s) could be syntax (code) or logic (intended outcome) based or both! For the following, assume that any classes used have already been properly defined and/or imported.

1.

```
ArrayList[Person] personList = new Person[30];
personList.add[new Person()];
```

2.

```
ArrayList[Person] personList = new Person[30];
personList[2] = new Person();
```

3.

```
private interface Locomotion
{
     private String howToMove();
}
```

4.

```
public class Fish implements ArrayList<> extends Animal
{
   // Members of class…
}
```

5.

```
public class ShippingBox(T)
{
  private String contents;
}
```

Think about It

1. What are the similarities between classes and abstract classes when it comes to class inheritance? What are the differences?
2. What are some of the rules for implementing an abstract class?
3. What happens if an abstract class extends another abstract class?
4. Why does it make sense that an abstract class cannot be instantiated?
5. What is unique about abstract methods?
6. Why are abstract methods not able to have the private or final modifiers applied to them?
7. How do abstract classes serve as valid data types?
8. What is a concrete class?
9. How do abstract classes and interface classes differ?
10. What is the difference in how an interface class is used with a "normal" class versus how an abstract class is used?

11. What are the benefits of implementing interfaces in your Java classes?
12. How do interfaces enable the concept of "loose coupling" between classes in Java?
13. What changes are required of a class that implements an interface class?
14. How do class inheritance and interface implementation differ?
15. How is a generic-type class or method useful, and how does it add flexibility to your application's code?
16. What is the generic-type parameter, and how is it used in generic-type code?
17. What are the general conventions for naming a generic-type parameter?
18. When can a generic-type parameter not be used?
19. What will happen if a primitive data type is used where a generic-type parameter is expected at runtime?
20. Describe what generic-type parameter bounding is. How are wildcard characters related to this? How are they different?

Short Syntax Problems

1. Write an abstract class called `ConsumerItem`. The consumer item class will have the following members and data fields:
   - `double price`
   - `String ownerName`
   - `int age`
   - `abstract double value()`

   Also, write a concrete class called `Vehicle` that extends `ConsumerItem`. Provide a concrete definition for the inherited method (value returned is 5% of price reduced for every year of age). Test out class `Vehicle` in an application.
2. Write an interface class called `AuctionProduct`. The interface will have one abstract method: `.startingBid()`. Have the `ConsumerItem` class from short syntax problem #1 implement this interface. Provide a concrete implementation of the interface's method where appropriate (starting bid = 80% of value for a `Vehicle`). Test everything out in an application.
3. Write an interface class called `AuctionFurniture` that extends `AuctionProduct` from short syntax problem #2. This interface has one abstract method: `.bidIncrements()`. Next, write a concrete class called `Furniture` that extends the `ConsumerGood` abstract class from short syntax problem #1 and implements the `AuctionFurniture` interface. Provide concrete definitions for any inherited methods (starting bid for furniture is 60% of value; bid increments for auction furniture is 10% of price, meaning the auctioneer will go up by 10% of the original price every time a bid is given). Test everything out in an application.
4. Write a static, generic-type method called `printAuctionDetails()` that accepts references to objects that implement the `AuctionProduct` interface. Have the method print the starting bid value, and if the object is a piece of furniture, have it also print the bid increment value. Use a loop in an application, and send several objects of `Vehicle` and `Furniture` types to it.

Full Problems

1. Fully expand the auction system from short syntax problems #1 through #4. Add `Car` and `Truck` as subclasses of `Vehicle`. Add `Couch` and `Bed` as subclasses of `Furniture`. Also, add a new class hierarchy with an abstract class for `RealEstate`. Add classes `ResidentialHome` and `Condo` as subclasses that extend `RealEstate`. Implement real-world appropriate data fields and methods for each of these classes. Provide concrete implementations for the `.startingBid()` method as you implement `AuctionProduct` across all these classes.
   i. Add a menu system that has two main options: Enter Items and Run Auction.
   ii. Enter Items will allow the user to create a new auction item of any of the above types: `Car`, `Truck`, `Couch`, `Bed`, `ResidentialHome`, or `Condo`.
   iii. Once items have been entered, the option "Run Auction" will loop upon each item in the inventory. (*Hint:* You need a way to store all these in some sort of data structure as the "inventory.") For each item, bidding will start at the starting bid price and go up by a predefined amount, if specified for a class. The user can enter new bids or can enter `-100` when

the auction is over. Use generic typing whenever appropriate throughout your implementation.

iv. Once the auction has finished, print out each item along with its value versus the amount paid for it in the auction. Print the profit per item.

2. Implement a bank account system. Create an abstract class called `BankAccount` with three subclasses: `CheckingAccount`, `SavingsAccount`, `CDAccount`. Implement an abstract method in `BankAccount` called `.calculateInterest()`. Each account type will have a separate interest rate, frequency of interest accrual (monthly, quarterly, or yearly). Implement an interface called `InterestInformation` that will define an abstract method `.reportInterestInformation()` where any implementing classes can report the nature of their interest-bearing characteristics (rate, frequency, and type of account).
   i. Create an application that will allow a user to create a new account. Represent this account with a generic-type `CustomerAccount` class that can accept an account object of any of the three bank account class types listed earlier.
   ii. Each account object should be capable of storing a list of deposits and withdrawals and capable of reporting the balance. Each account will have different withdrawal rules: checking: unlimited withdrawals down to balance of $0; savings: ten withdrawals a year; CD: only one withdrawal a year.
   iii. For every deposit made, invoke the `.calculateInterest()` method, and print to the screen what the next interest payment will be for the account.
   iv. For every account, give the user the ability to print the last ten activities on the account.
   v. Use generic typing wherever it is appropriate throughout your implementation.