Interfaces & Inheritance

Interfaces

There are a number of situations in software engineering when it's NB for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Interfaces are such contracts.

EG: imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.
The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know how the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

An interface is a reference type, similar to a class, that *can* contain only **constants**, **method signatures**, **default methods**, **static methods**, & **nested types**.

Method bodies exist only for default methods and static methods.

Interfaces cannot be **instantiated**—they can only be implemented by classes or extended by other interfaces.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures

   // An enum with values RIGHT, LEFT
   int turn(Direction direction,
            double radius,
            double startSpeed,
            double endSpeed);
   int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
   int signalTurn(Direction direction,
                  boolean signalOn);
   int getRadarFront(double distanceToCar,
                     double speedOfCar);
   int getRadarRear(double distanceToCar,
                    double speedOfCar);
         ......
   // more method signatures
}
```

Note: Method signatures have **no** braces and are terminated with a semicolon.

@Romina Podolo

Interfaces

To use an interface, you write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

Example:

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    public int signalTurn(Direction direction, boolean signalOn) {
       // code to turn BMW's LEFT turn indicator lights on
       // code to turn BMW's LEFT turn indicator lights off
       // code to turn BMW's RIGHT turn indicator lights on
       // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. Doing so the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate etc.

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard Application Programming Interface (API). APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs.

The image processing company writes its classes to implement an interface, which it makes public to its customers.
The graphics company then invokes the image processing methods using the signatures and return types defined in the interface.

While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded **secret**—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

@Romina Podolo

Defining an Interface

An interface declaration consists of modifiers, the keyword *interface*, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

Example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body

The interface body can contain abstract methods, default methods, and static methods. An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the default modifier, and static methods with the static keyword. All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.

An interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final. Again, you can omit these modifiers.

@Romina Podolo

Implementing an Interface

To declare a class that implements an interface, you include an *implements* clause in the class declaration. Your class can implement more than one interface, so the *implements* keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the implements clause follows the extends clause, if there is one.

A Sample Interface, Relatable:

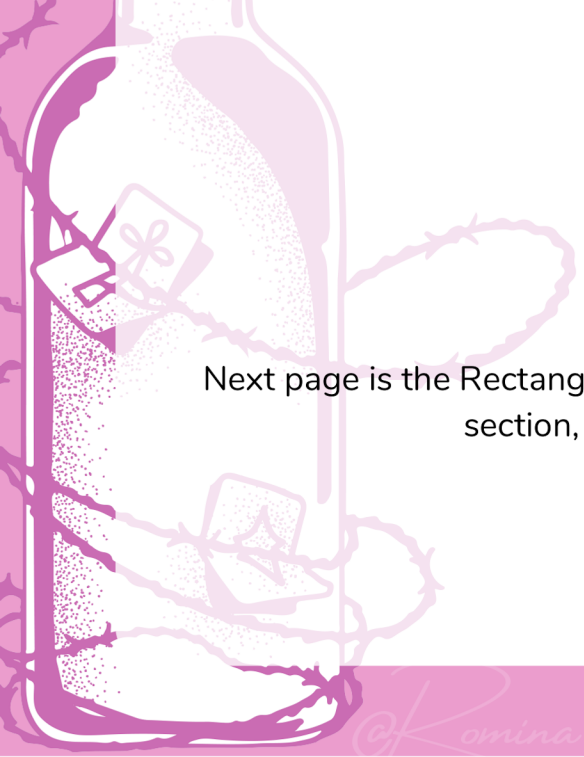
Consider an interface that defines how to compare the size of objects.

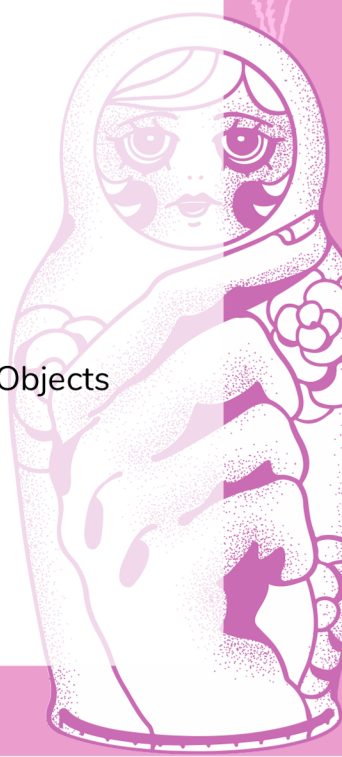
```
public interface Relatable {

    // this (object calling isLargerThan)
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater than,
    // equal to, or less than other
    public int isLargerThan(Relatable other);
}
```

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement Relatable.
Any class can implement Relatable if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth.

For planar geometric objects, area would be a good choice (see the RectanglePlus class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the isLargerThan() method.

If you know that a class implements Relatable, then you know that you can compare the size of the objects instantiated from that class.





Next page is the Rectangle class that was presented in the Creating Objects section, rewritten to implement Relatable.

@Romina Podolo

Example:

Implementing the Relatable Interface

```
public class RectanglePlus
    implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing
    // the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement
    // the Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect
            = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Because RectanglePlus implements Relatable, the size of any two RectanglePlus objects can be compared.

Note: The `isLargerThan` method, as defined in the `Relatable` interface, takes an object of type `Relatable`. The line of code, shown in bold in the previous example, casts `other` to a `RectanglePlus` instance. Type casting tells the compiler what the object really is. Invoking `getArea` directly on the `other` instance (`other.getArea()`) would fail to compile because the compiler does not understand that `other` is actually an instance of `RectanglePlus`.

@Komina Podolo

Using an Interface as a Type

When you define a new interface, you are defining a new **reference** data type.

You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

Eg: A method for finding the largest object in a pair of objects, for any objects that are instantiated from a class that implements Relatable:

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

By casting object1 to a Relatable type, it can invoke the isLargerThan method.

If you make a point of implementing Relatable in a wide variety of classes, the objects instantiated from any of those classes can be compared with the findLargest() method—provided that both objects are of the same class.

Similarly, they can all be compared with the following methods:

```
public Object findSmallest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) < 0)
      return object1;
   else
      return object2;
}

public boolean isEqual(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) == 0)
      return true;
   else
      return false;
}
```

These methods work for any "relatable" objects, no matter what their class inheritance is. When they implement Relatable, they can be of both their own class (or superclass) type and a Relatable type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.

@Romina Podolo

Evolving Interfaces

Eg: Interface called DoIt:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Perhaps at a later time, you want to add a third method to DoIt, so that the interface now becomes:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

If you make this change, then all classes that implement the old *DoIt* interface will **break** because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

Anticipate ALL uses for your interface and **specify** it completely from the beginning. If you want to add additional methods to an interface, you have 7 options.

1

Create a DoItPlus interface that **extends** DoIt:

```
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Now users of your code can **choose** to continue to use the old interface or to upgrade to the new interface.

2

Define new methods as **default** methods.

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }
}
```

Eg: defines a default method named didItWork

Note

You must provide an implementation for default methods. You could also define new *static* methods to existing *interfaces*. Users who have classes that implement interfaces enhanced with new default or static methods do NOT have to modify or recompile them to accommodate the additional methods.

@Romina Podolo

Default Methods

The section, Interfaces, describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars?

These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Interface: TimeClient

```
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

The following class, SimpleTimeClient, implements TimeClient:

Example on page 9 (following page)

@Romina Podolo

Default Methods

```
package defaultmethods;

import java.time.*;
import java.lang.*;
import java.util.*;

public class SimpleTimeClient implements TimeClient {

    private LocalDateTime dateAndTime;

    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }

    public void setTime(int hour, int minute, int second) {
        LocalDate currentDate = LocalDate.from(dateAndTime);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(currentDate, timeToSet);
    }

    public void setDate(int day, int month, int year) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime currentTime = LocalTime.from(dateAndTime);
        dateAndTime = LocalDateTime.of(dateToSet, currentTime);
    }

    public void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
    }

    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }

    public String toString() {
        return dateAndTime.toString();
    }

    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println(myTimeClient.toString());
    }
}
```

Adding new functionality to the TimeClient interface, such as the ability to specify a time zone through a ZonedDateTimeobject
(Like a LocalDateTime object except that it stores time zone information)

The following class, SimpleTimeClient, implements TimeClient:

```
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
    ZonedDateTime getZonedDateTime(String zoneString);
}
```

Following this modification to the TimeClient interface, you would also have to modify the class SimpleTimeClient & implement the method getZonedDateTime. However, rather than leaving getZonedDateTime as abstract (as in the previous example), you can instead define a default implementation.

Remember that an abstract method is a method declared without an implementation.

@Komina Todolo

Default Methods

```
package defaultmethods;

import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDpackage defaultmethods;

import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                                int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

You specify that a method definition in an interface is a **default** method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly **public**, so you can omit the **public** modifier.

With this interface, you do not have to modify the class *SimpleTimeClient*, and this class (and any class that implements the interface *TimeClient*), will have the method *getZonedDateTime* already defined.

The following example, *TestSimpleTimeClient*, invokes the method *getZonedDateTime* from an instance of *SimpleTimeClient*:

```
package defaultmethods;

import java.time.*;
import java.lang.*;
import java.util.*;

public class TestSimpleTimeClient {
    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + myTimeClient.toString());
        System.out.println("Time in California: " +
            myTimeClient.getZonedDateTime("Blah blah").toString());
    }
}
```

@Romina Podolo

Default Methods

Extending Interfaces That Contain Default Methods

When you extend an interface that contains a default method, you can do the following:

- Not mention the default method at all, which lets your extended interface inherit the default method.
- Redeclare the default method, which makes it abstract.
- Redefine the default method, which overrides it.

Extend the interface TimeClient as follows: `public interface AnotherTimeClient extends TimeClient { }`

Any class that implements the interface AnotherTimeClient will have the implementation specified by the default method TimeClient.getZonedDateTime.

Extend the interface TimeClient as follows:

```
public interface AbstractZoneTimeClient extends TimeClient {
    public ZonedDateTime getZonedDateTime(String zoneString);
}
```

Any class that implements the interface AbstractZoneTimeClient will have to implement the method getZonedDateTime; this method is an abstract method like all other non-default (and non-static) methods in an interface.

Extend the interface TimeClient as follows:

```
public interface HandleInvalidTimeZoneClient extends TimeClient {
    default public ZonedDateTime getZonedDateTime(String zoneString) {
        try {
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));
        } catch (DateTimeException e) {
            System.err.println("Invalid zone ID: " + zoneString +
                "; using the default time zone instead.");
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());
        }
    }
}
```

Any class that implements the interface HandleInvalidTimeZoneClient will use the implementation of getZonedDateTime specified by this interface instead of the one specified by the interface TimeClient.

Static Methods

Defining static methods in interfaces:

A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods. This makes it easier for you to organise helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.

Example:

Defines a static method that retrieves a ZoneId object corresponding to a time zone identifier; it uses the system default time zone if there is no ZoneId object corresponding to the given identifier. (Resulting in simplifying the method getZonedDateTime):

```
public interface TimeClient {
    // ...
    static public ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default public ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Like static methods in classes, you specify that a method definition in an interface is a static method with the static keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

@Romina Podolo

Default Methods

Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces. This section demonstrates how theComparator interface has been enhanced with default and static methods.

Consider the Card and Deck classes as described in Questions and Exercises: Classes.
Example rewrites the Card and Deck classes as interfaces. The Cardinterface contains two enum types (Suit and Rank) and two abstract methods (getSuit and getRank):

```
package defaultmethods;

public interface Card extends Comparable<Card> {

    public enum Suit {
        DIAMONDS (1, "Diamonds"),
        CLUBS    (2, "Clubs"   ),
        HEARTS   (3, "Hearts"  ),
        SPADES   (4, "Spades"  );

        private final int value;
        private final String text;
        Suit(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public enum Rank {
        DEUCE  (2 , "Two"  ),
        THREE  (3 , "Three"),
        FOUR   (4 , "Four" ),
        FIVE   (5 , "Five" ),
        SIX    (6 , "Six"  ),
        SEVEN  (7 , "Seven"),
        EIGHT  (8 , "Eight"),
        NINE   (9 , "Nine" ),
        TEN    (10, "Ten"  ),
        JACK   (11, "Jack" ),
        QUEEN  (12, "Queen"),
        KING   (13, "King" ),
        ACE    (14, "Ace"  );
        private final int value;
        private final String text;
        Rank(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public Card.Suit getSuit();
    public Card.Rank getRank();
}
```

@Romina Podolo

Default Methods

Integrating Default Methods into Existing Libraries

The Deck interface contains various methods that manipulate cards in a deck:

```
package defaultmethods;

import java.util.*;
import java.util.stream.*;
import java.lang.*;

public interface Deck {

    List<Card> getCards();
    Deck deckFactory();
    int size();
    void addCard(Card card);
    void addCards(List<Card> cards);
    void addDeck(Deck deck);
    void shuffle();
    void sort();
    void sort(Comparator<Card> c);
    String deckToString();

    Map<Integer, Deck> deal(int players, int numberOfCards)
        throws IllegalArgumentException;

}
```

The class *PlayingCard* implements the interface Card, and the class *StandardDeck* implements the interface *Deck*.

The class *StandardDeck* implements the abstract method Deck.sort as follows:

```
public class StandardDeck implements Deck {

    private List<Card> entireDeck;

    // ...

    public void sort() {
        Collections.sort(entireDeck);
    }

    // ...
}
```

The method *Collections.sort* sorts an instance of *List* whose element type implements the interface *Comparable*. The member *entireDeck* is an instance of *List* whose elements are of the type Card, which extends Comparable. The class *PlayingCard* implements the *Comparable.compareTo* method as follows:

```
public int hashCode() {
    return ((suit.value()-1)*13)+rank.value();
}

public int compareTo(Card o) {
    return this.hashCode() - o.hashCode();
}
```

The method *compareTo* causes the method ***StandardDeck.sort()*** to sort the deck of cards first by suit, then by rank.

What if you want to sort the deck first by rank, then by suit? You would need to implement the Comparator interface to specify new sorting criteria, and use the *methodsort(List<T> list, Comparator<? super T> c)* (the version of the sort method that includes a Comparator parameter). You can define the following method in the class StandardDeck:

```
public void sort(Comparator<Card> c) {
    Collections.sort(entireDeck, c);
}
```

With this method, you can specify how the method Collections.sort sorts instances of the Card class. One way to do this is to implement the Comparator interface to specify how you want the cards sorted

@Romina Podolo

Default Methods

Integrating Default Methods into Existing Libraries

The example *SortByRankThenSuit* does this:

```
package defaultmethods;

import java.util.*;
import java.util.stream.*;
import java.lang.*;

public class SortByRankThenSuit implements Comparator<Card> {
    public int compare(Card firstCard, Card secondCard) {
        int compVal =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compVal != 0)
            return compVal;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
}
```

The following invocation sorts the deck of playing cards first by rank, then by suit:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(new SortByRankThenSuit());
```

Very verbose approach therefore it's better to specify just the sort criteria and avoid creating multiple sorting implementations. Suppose that you are the developer who wrote the Comparator interface. What default or static methods could you add to the Comparator interface to enable other developers to more easily specify sort criteria?

To start, suppose that you want to sort the deck of playing cards by rank, regardless of suit. Invoking StandardDeck.sort method as follows:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) ->
        firstCard.getRank().value() - secondCard.getRank().value()
);
```

Because the interface Comparator is a functional interface, you can use a lambda expression as an argument for the sort method. In this example, the lambda expression compares two integer values.
Simpler for developers if they could create a Comparator instance by invoking the method Card.getRank only.
Helpful if developers could create a Comparator instance that compares any object that can return a numerical value from a method such as getValue or hashCode. The Comparator interface has been enhanced with this ability with the static method comparing:

```
myDeck.sort(Comparator.comparing((card) -> card.getRank()));
```

Example: use a method reference instead: `myDeck.sort(Comparator.comparing(Card::getRank));`

This invocation better demonstrates how to specify different sort criteria and avoid creating multiple sorting implementations.
The Comparator interface has been enhanced with other versions of the static method comparing such as comparingDouble and comparingLong that enable you to create Comparator instances that compare other data types.
Suppose that your developers would like to create a Comparator instance that could compare objects with more than one criteria.

For example, how would you sort the deck of playing cards first by rank, and then by suit?

@Romina Podolo

Default Methods

Integrating Default Methods into Existing Libraries

As before, you could use a lambda expression to specify these sort criteria:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) -> {
        int compare =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compare != 0)
            return compare;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
);
```

It would be simpler for your developers if they could build a Comparator instance from a series of Comparator instances. The Comparator interface has been enhanced with this ability with the default method thenComparing:

```
myDeck.sort(
    Comparator
        .comparing(Card::getRank)
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

The Comparator interface has been enhanced with other versions of the default method thenComparing (such as thenComparingDouble and thenComparingLong) that enable you to build Comparator instances that compare other data types.

Suppose that your developers would like to create a Comparator instance that enables them to sort a collection of objects in reverse order.
Example:
How would you sort the deck of playing cards first by descending order of rank, from Ace to Two (instead of from Two to Ace)?
You could specify another *lambda* expression. However, it would be simpler for your developers if they could reverse an existing Comparator by invoking a method. The Comparator interface has been enhanced with this ability with the default method reversed:

```
myDeck.sort(
    Comparator.comparing(Card::getRank)
        .reversed()
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

This example demonstrates how the Comparator interface has been enhanced with default methods, static methods, lambda expressions, and method references to create more expressive library methods whose functionality programmers can quickly deduce by looking at how they are invoked. Use these constructs to enhance the interfaces in your libraries.

Summary of Interfaces

- An interface declaration can contain method signatures, default methods, static methods and constant definitions. The only methods that have implementations are default and static methods.
- A class that implements an interface must implement all the methods declared in the interface.
- An interface name can be used anywhere a type can be used.

@Romina Podolo

Inheritance

Classes can be *derived* from other classes, thereby inheriting fields and methods from those classes.

Definition

A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).

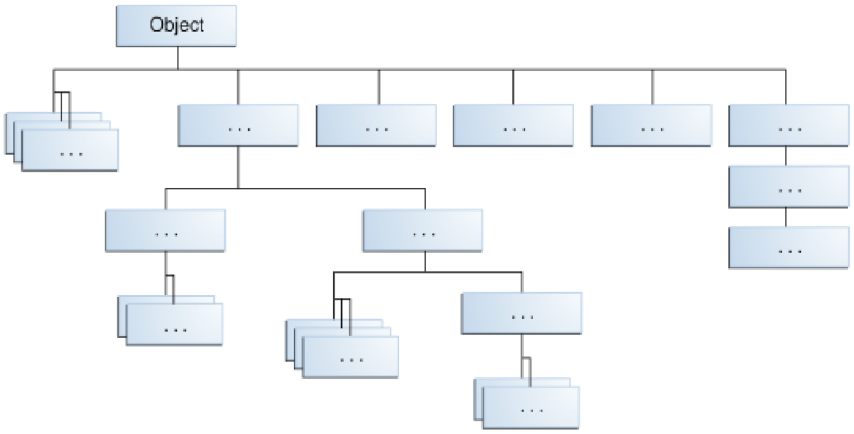
Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be descended from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.
A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The Java Platform Class Hierarchy

The Object class, defined in the java.lang package, defines and implements behaviour common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Decendencts of Object

Module java.base
Package java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass.

All objects, including arrays, implement the methods of this class.

Constructor Summary

Constructors

Constructor	Description
`Object()`	Constructs a new object.

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method	Description
`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`void`	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.
`void`	`wait()`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.
`void`	`wait(long timeoutMillis)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`void`	`wait(long timeoutMillis, int nanos)`	Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

@Komina Podolo

Example of Inheritance

Example

Sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.

@Romina Podolo

Inheritance

What You Can Do in a Subclass

A subclass inherits all of the public & protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Casting Objects

An object is of the data type of the class from which it was instantiated.

Example: `public MountainBike myBike = new MountainBike();` then myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true: a Bicycle may be a MountainBike, but it isn't necessarily. Similarly, an Object may be a Bicycle or a MountainBike, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance & implementations.

```
Object obj = new MountainBike();
```

***obj** is both an **Object** and a **MountainBike** (until such time as obj is assigned another object that is not a MountainBike). This is called **implicit casting**.*

But, writing `MountainBike myBike = obj;` will result in a compile-time error because obj is not known to the compiler to be a MountainBike.

However, we can tell the compiler that we promise to assign a MountainBike to obj by ***explicit casting***:

```
MountainBike myBike = (MountainBike)obj;
```

@Romina Podolo

Inheritance

Casting Objects Cont.

This cast example (from the previous page) inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a MountainBike at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

Multiple Inheritance of State, Implementation, and Type

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. You can instantiate a class to create an object, which you cannot do with interfaces. As explained in the section What Is an Object?, an object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of multiple inheritance of state, which is the ability to inherit fields from multiple classes. Eg, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence? Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple inheritance of implementation is the ability to inherit method definitions from multiple classes. Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass. Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.

The Java programming language supports multiple inheritance of type, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface. This is discussed in the section Using an Interface as a Type.

As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

@Romina Podolo

Inheritance

Overriding and Hiding Methods

Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overridesthe superclass's method. The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a covariant return type. When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on @Override, see Annotations.

Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

The distinction between hiding a static method & overriding an instance method has NB implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

2 classes: Animal, which contains one instance method and one static method:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of Animal, is called Cat:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Output

```
The static method in Animal
The instance method in Cat
```

The Cat class overrides the instance method in Animal and hides the static method in Animal. The main method in this class creates an instance of Cat and invokes testClassMethod() on the class and testInstanceMethod() on the instance.

The version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

@Komina Podolo

Inheritance

Overriding and Hiding Methods

Interface Methods

Default methods & abstract methods in interfaces are *inherited* like instance methods.
However, when the *supertypes* of a class or interface provide *multiple* default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict.

These rules are driven by the following 2 principles:

1. **Instance** methods are preferred over interface default methods.

```
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}
public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}
public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Pegasus.identifyMyself` returns the string `I am a horse.`

2. Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}
public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}
public interface FireBreather extends Animal { }
public class Dragon implements EggLayer, FireBreather {
    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Dragon.identifyMyself` returns the string `I am able to lay eggs.`

@Romina Podolo

Overriding & Hiding Methods

Interface Methods

If 2 / more *independently defined default methods* **conflict**, / a *default method conflicts* with an **abstract** method, then the Java compiler produces a ***compiler error***. You must **explicitly** override the supertype methods.

Example:

Computer-controlled cars that can now fly
2 interfaces; `OperateCar & FlyCar`
that provide default implementations for the same `startEngine` method,

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

A class that implements both OperateCar and FlyCar must override the method startEngine. You could invoke any of the of the default implementations with the super keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding super (in this example, FlyCar or OperateCar) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the super keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods.

```
public interface Mammal {
    String identifyMyself();
}
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself` returns the string `I am a horse.`

The class Mustang inherits the method identifyMyself from the class Horse, which overrides the abstract method of the same name in the interface Mammal.

Note: Static methods in interfaces are never inherited.

@Romina Podolo

Overriding & Hiding Methods

Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method.

Example: a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Summary

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	`Overrides`	`Generates a compile-time error`
Subclass Static Method	`Generates a compile-time error`	`Hides`

Table shows what happens when you define a method with the same signature as a method in a superclass.

NOTE: In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

@Romina Podolo

Inheritance

Polymorphism

Definition: A principle in biology in which an organism / species can have many different forms / stages. This principle can also be applied to object-oriented programming & languages like the Java language. Subclasses of a class can define their own unique behaviors & yet share some of the same functionality of the parent class. **Polymorphism can be demonstrated with a tiny modification to the Bicycle class.**

Example: `printDescription` method could be added to the class that displays all the data currently stored in an instance.

```
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, **extend** the Bicycle class with a MountainBike and a RoadBike class. For MountainBike, add a field for suspension, which is a String value that indicates if the bike has a front shock absorber, Front. Or, the bike has a front and back shock absorber, Dual.

updated class

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
               int startCadence,
               int startSpeed,
               int startGear,
               String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

...note the overridden printDescription method.
In addition to the information provided before, additional data about the suspension is included to the output.

@Romina Podolo

Polymorphism

Next, create the RoadBike class. Because road or racing bikes have skinny tires, add an attribute to track the tire width.

Here is the RoadBike class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Summary:

3 classes: Bicycle, MountainBike, and RoadBike.
The 2 subclasses override the printDescription method and print unique information.

Note: Once again, the printDescription method has been overridden. Info about the tire width is displayed.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

Output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It doesn't call the method that is defined by the variable's type. This behavior is referred to as virtual **method invocation** and demonstrates an aspect of the important polymorphism features in the Java language.

@Romina Podolo

Inheritance

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section.
NOT RECOMMEND: hiding fields (makes code difficult to read).

Using the Keyword super

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.

You can also use super to refer to a hidden field **(hiding fields is discouraged).**

Eg class: `Superclass`

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in
Superclass.");
    }
}
```

Eg subclass: `Subclass`

that overrides `printMethod()`

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Within Subclass, the simple name printMethod() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to printMethod() inherited from Superclass, Subclass must use a qualified name, using super as shown.

Compiling and executing Subclass prints the following:

```
Printed in Superclass.
Printed in Subclass
```

Subclass Constructors

Example: how to use the super keyword to invoke a superclass's constructor.

Remember Bicycle eg that MountainBike is a subclass of Bicycle. Here MountainBike(subclass) constructor that calls the superclass constructor then adds initialisation code of its own:

```
public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                    int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the first line in the subclass constructor.

Syntax

calling a superclass constructor is:

OR `super();` superclass no-argument constructor is called.

`super(parameter list);` superclass constructor with a matching parameter list is called.

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. `Object` *does* have such a constructor, so if `Object` is the only superclass, there is no problem.

Constructor Chaining

The process of calling a sequence of constructors. If a subclass constructor invokes a constructor of its superclass, either explicitly / implicitly, a chain of constructors is called, all the way back to the constructor of Object. Be aware of it when there is a **long line of class descent.**

@Romina Podolo

Inheritance

Object as a Superclass

The Object class, in the java.lang package, sits at the top of the class hierarchy tree.

Every class is a descendant, direct /indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class.

The methods inherited from Object that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException`
  Creates and returns a copy of this object.
- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class getClass()`
  Returns the runtime class of an object.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson.

There are 5 of these methods:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

The clone() Method

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object.

To create a clone: `aCloneableObject.clone();`

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedExceptionexception.

clone() is declared as: `protected Object clone() throws CloneNotSupportedException`

OR

`public Object clone() throws CloneNotSupportedException`

...if you are going to write a clone() method to override the one in Object..

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object and ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

@Romina Podolo

Object as a Superclass

The equals() Method

The equals() method compares 2 objects for equality & returns true if they are equal.

The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.

To test whether 2 objects are equal in the sense of equivalency (containing the same information), you must override the equals() method.

Example:

`Book` class that overrides `equals()`

```
public class Book {
    String ISBN;

    public String getISBN() {
        return ISBN;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

Consider this code that tests two instances of the Book class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the equals() method if the identity operator is not appropriate for your class.

Note: If you override equals(), you must override hashCode() as well.

The finalize() Method

The Object class provides a callback method, finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, don't rely on this method to do your cleanup for you.
Eg: if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors. Instead, use a try-with resources statement to automatically close your application's resources.

See The try-with-resources Statement and Finalization and Weak, Soft, and Phantom References in Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide.

@Romina Podolo

Object as a Superclass

The getClass() Method

You can't override `getClass`

The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()).

example:

The following method gets & displays the class name of an object

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
        obj.getClass().getSimpleName());
}
```

The Class class, in the java.lang package, has a large number of methods (more than 50).

Eg: you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

The hashCode() Method

The value returned by hashCode() is the object's hash code, which is an integer value generated by a hashing algorithm.

By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

The toString() Method

You should always consider overriding the toString() method in your classes.
The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

```
System.out.println(firstBook.toString());
```

...which would, for a properly overridden toString() method, print something useful, like this:

```
ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition
```

@Romina Podolo

Inheritance

Writing Final Classes and Methods

You can declare some or all of a class's methods final. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.
You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.

example:

you might want to make the `getFirstPlayer` method in this `ChessAlgorithm` class final

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...

}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you *can also declare an entire class final*. A class that is declared ***final can't be subclassed***. This is particularly useful, for example, when creating an immutable class like the String class.

Abstract Methods and Classes

Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You **can't instantiate** them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. Additionally you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Abstract classes vs interfaces:

- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.

An example of an abstract class in the JDK is AbstractMap, which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.

An example of a class in the JDK that implements several interfaces is HashMap, which implements the interfaces Serializable, Cloneable, and Map<K, V>. By reading this list of interfaces, you can infer that an instance of HashMap (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map. In addition, the Map<K, V> interface has been enhanced with many default methods such as merge and forEach that older classes that have implemented this interface do not have to define.

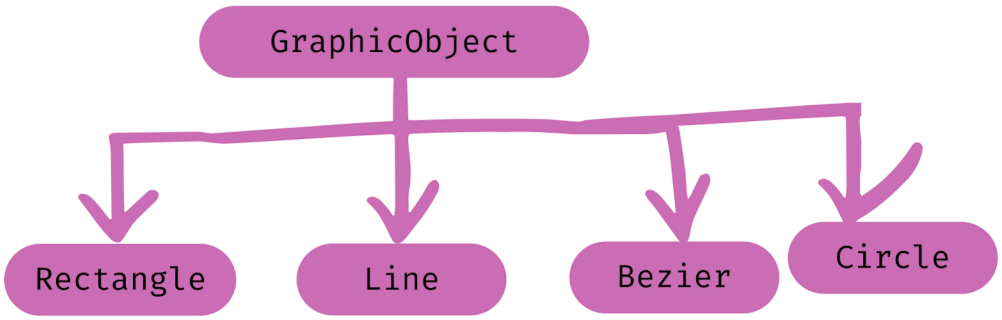
Note that many software libraries use both abstract classes and interfaces; the HashMap class implements several interfaces and also extends the abstract class AbstractMap.

@Romina Podolo

Abstract Methods & Classes

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (eg: position, orientation, line color, fill color) and behaviors (eg: moveTo, rotate, resize, draw) in common. Some of these states & behaviors are the same for all graphic objects (eg: position, fill color, and moveTo). Others require different implementations (eg, resize or draw). All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (eg, GraphicObject) as shown in the following figure.



1. Declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method.

2. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways.

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each nonabstract subclass of GraphicObject, such as Circle & Rectangle, must provide implementations for the draw & resize methods:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

@Romina Podolo

Abstract Methods & Classes

When an Abstract Class Implements an Interface

In the section on Interfaces, it was noted that a class that implements an interface must implement all of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract.

Example:

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

class X has to be abstract because it **doesn't** fully implement Y, but class XX implements Y

Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

Summary of Inheritance

- Except for the Object class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide fields or methods that it inherits. (Note that hiding fields is generally bad programming practice.)
- The table in Overriding and Hiding Methods section shows the effect of declaring a method with the same signature as a method in the superclass.
- The Object class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from Object include toString(), equals(), clone(), and getClass().
- You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.

@Romina Podolo