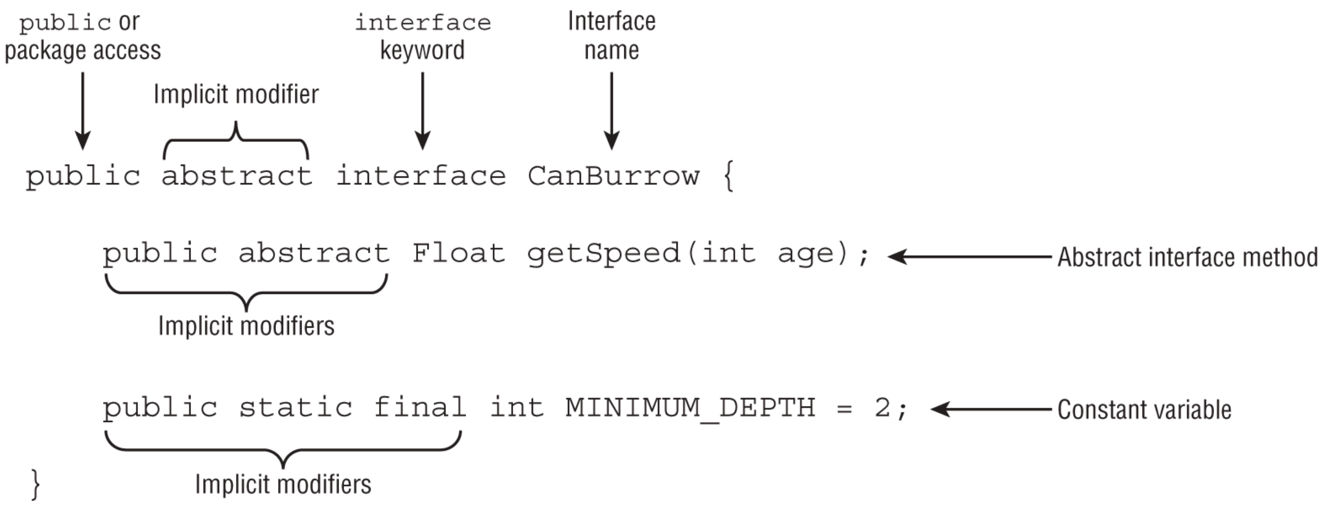
Capitolul 7 - Beyond classes

Interface

> Interface
>
> An abstract data type that declares a list of abstract methods that any class implementing the interface.
> A class can implement one or more interfaces.

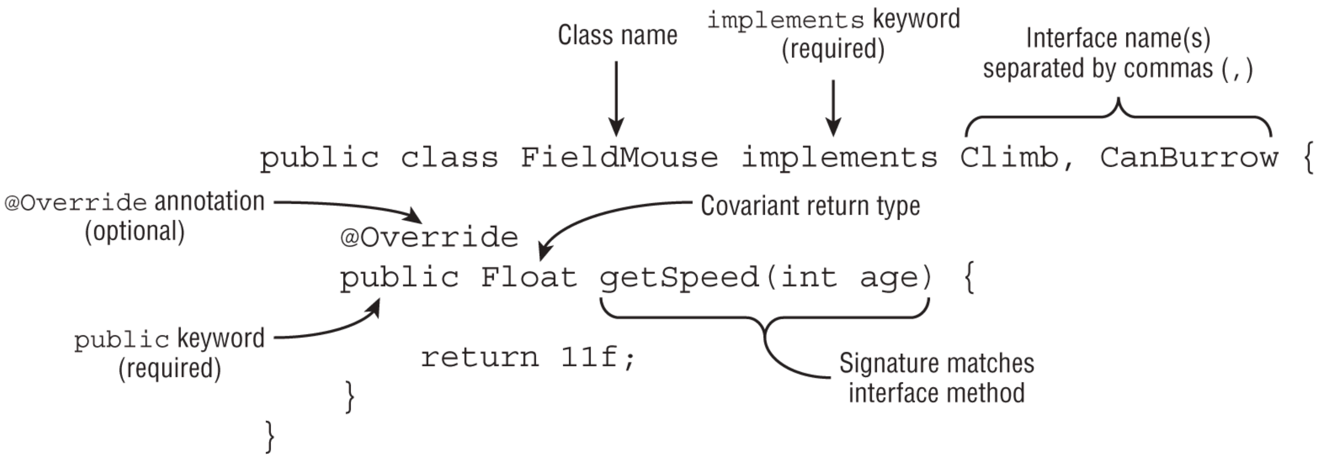


```
public abstract interface CanBurrow {

    public abstract Float getSpeed(int age);

    public static final int MINIMUM_DEPTH = 2;
}
```

- ***Implicit modifier*** ->modifier that the compiler automatically inserts into the code.
- Also the code doesn't compile if you marked the interface with `final`.



```
public class FieldMouse implements Climb, CanBurrow {
    @Override
    public Float getSpeed(int age) {
        return 11f;
    }
}
```

Class name
implements keyword (required)
Interface name(s) separated by commas (,)
@Override annotation (optional)
Covariant return type
public keyword (required)
Signature matches interface method

Also, an interface can extend another interface using the `extends` keyword.

```
public interface Nocturnal{}

public interface HasBigEyes extends Nocturnal{}
```

Also, an interface can extend multiple interfaces.

```java
public interface Nocturnal {
    public int hunt();
}

public interface CanFly {
    public void flap();
}

public interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements HasBigEyes {
    public int hunt(){ return 5; }
    public void flap(){ System.out.println("Flap!"); }
}
```

Inheriting an Interface

When a concrete class inherits an interface, all of the inherited abstract methods must be implemented

Mixing class and interface keywords

A class can implement an interface, but can't extend an interface.
An interface can extend other interfaces, but can't implements other interfaces.

Inheriting duplicate abstract methods

Java supports inheriting 2 abstract methods that have compatible method declaration.
Compatible -> a method can be written that properly overrides both inherited methods.

```java
public interface Herbivore { public int eatPlants(int plantsLeft); }

public interface Omnivore { public int eatPlants(int foodRemaining); }

public class Bear implements Herbivore, Omnivore {
   public int eatPlants(int plants) {
      System.out.print("Eating plants");
      return plants - 1;
   }
}
```

- Interfaces are *implicitly* `abstract`.
- Interface variables are *implicitly* `public`, `static`, and `final`.
- Interface methods without a body are *implicitly* `abstract`.
- Interface methods without the `private` modifier are *implicitly* `public`.

```
public interface Soar {
   int MAX_HEIGHT = 10;
   final static boolean UNDERWATER = true;
   void fly(int speed);
   abstract void takeoff();
   public abstract double dive();

}

public abstract interface Soar {
   public static final int MAX_HEIGHT = 10;
   public final static boolean UNDERWATER = true;
   public abstract void fly(int speed);
   public abstract void takeoff();
   public abstract double dive();
}
```

Differences between Interfaces and Abstract Classes

Only interfaces make use of implicit modifiers.

```
abstract class Husky {    // abstract modifier required
   abstract void play();  // abstract modifier required
}

interface Poodle {        // abstract modifier optional
   void play();           // abstract modifier optional
}

----------------------------------------------------------------------

public class Webby extends Husky {
   void play() {}      // play() is declared with package access in Husky
}

public class Georgette implements Poodle {
   void play() {}      // DOES NOT COMPILE - play() is public in Poodle and
here we change the access modifier
}
```

Default method

Is a method defined in an interface with the default keyword and includes a method body.
It may be optionally overridden by a class implementing the interface.
Usage: backward compatibility.

```
public interface IsColdBlooded {
   boolean hasScales();
   default double getTemperature() {
      return 10.0;
   }
}

-----------------------------------------

public class Snake implements IsColdBlooded {
   public boolean hasScales() {       // Required override
      return true;
   }

   public double getTemperature() {   // Optional override
      return 12.2;
   }
}
```

Default Interface method definition rules:

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is implicitly `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

A `default` method exists on any object inheriting the interface, not on the interface itself. In other words, you should treat it like an inherited method that can be optionally overridden, rather than as a `static` method.

```
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   }
}
```

This is an area where a `default` method `getSpeed()` exhibits properties of both an instance and `static` method. We use the interface name to indicate which method we want to call, but we use the `super` keyword to show that we are following instance inheritance, not class inheritance.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is implicitly `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name

```
public interface Hop{
    static int getjumpHeight(){
        return 8;
    }
}

public class Skip implements Hop{
    public int skip(){
        return Hop.getJumpHeight(); // si numai asa poti accesa metodele
statice din interfata
    }
}
```

Reusing Code with *private* Interface Methods

```
public interface Schedule {

   default void wakeUp()        { checkTime(7);  }
   private void haveBreakfast() { checkTime(9);  }
   static void workOut()        { checkTime(18); }
   private static void checkTime(int hour) {
      if (hour > 17) {
         System.out.println("You're late!");
      }
      else {
         System.out.println("You have "+(17-hour)+" hours left "
               + "to make the appointment");
      }
   }
}
```

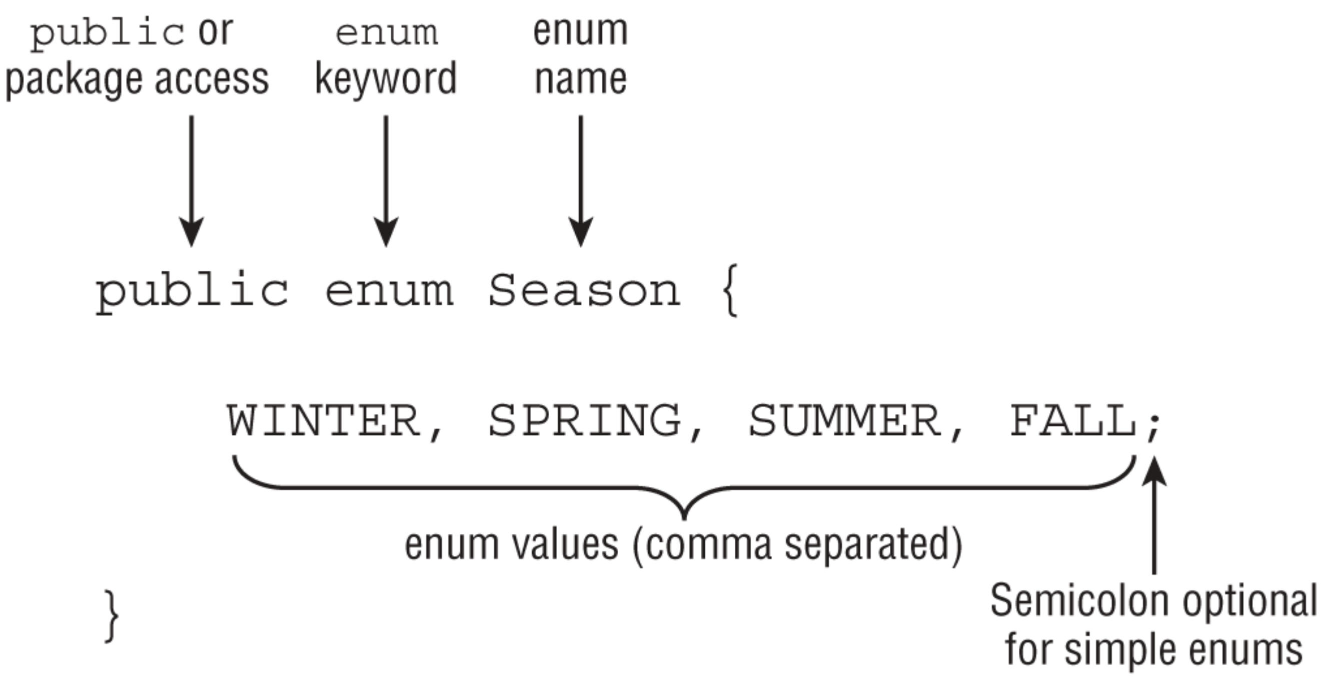
Private Interface Method Definition Rules

1. A `private` interface method must be marked with the `private` modifier and include a method body.
2. A `private static` interface method may be called by any method within the interface definition.
3. A `private` interface method may only be called by `default` and other `private` non-`static` methods within the interface definition.

Enums

Enum

List of values


public or package access
enum keyword
enum name
public enum Season {
WINTER, SPRING, SUMMER, FALL;
enum values (comma separated)
}
Semicolon optional for simple enums

The values in an enum are fixed. You cannot add more by extending the enum nor can you mark an enum `final`. On the other hand, an enum can implement an interface, which we will cover shortly.

```
for(var season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
-----------------------------------------------------------------
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

```
Season s = Season.valueOf("SUMMER"); // SUMMER
Season t = Season.valueOf("summer"); // IllegalArgumentException
```

Complex enums

```
21: interface Visitors { void printVisitors(); }
22: enum SeasonWithVisitors implements Visitors {
23:    WINTER("Low"), SPRING("Medium"), SUMMER("High"), FALL("Medium");
       //semicolon required for complex enums
       //always list of values FIRST
24:
25:    private final String visitors;
26:    public static final String DESCRIPTION = "Weather enum";
27:    private SeasonWithVisitors(String visitors) {
28:       System.out.print("constructing,");
29:       this.visitors = visitors;
30:    }
31:    @Override public void printVisitors() {
32:       System.out.println(visitors);
33:    } }
```

All enum constructor are implicitly private.
Parentheses from line 23 -> constructor calls. The first time we ask for any of the enum values, Java constructs all of the enum values. After that, Java just returns the already constructed enum values.

Also the enums can contains `static` and instance methods.

```
public enum SeasonWithTimes {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   FALL {
      public String getHours() { return "9am-5pm"; }
   };
   public abstract String getHours();
}
//here is an example of abstract class-ish -> with an abstract method -> so
all enum values
//must implement this
--------------------------------------------------------

SeasonWithVisitors.SUMMER.printVisitors(); // we access the method of enum
```

```
value like here
```

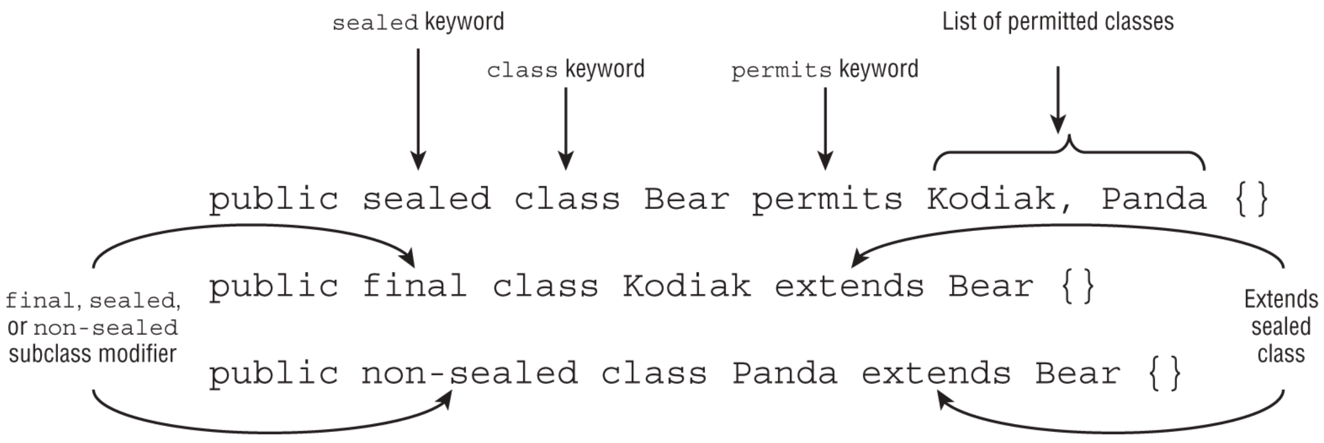
If we don't want each and every enum value to have a method -> can create an implementation for all values, and override for special cases.

```
public enum SeasonWithTimes {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   SPRING, FALL;
   public String getHours() { return "9am-5pm"; }
}
```

Sealed classes

Sealed class

Class that restricts which other classes may extend it.


sealed keyword
class keyword
permits keyword
List of permitted classes
public sealed class Bear permits Kodiak, Panda {}
public final class Kodiak extends Bear {}
public non-sealed class Panda extends Bear {}
final, sealed, or non-sealed subclass modifier
Extends sealed class

Sealed Class keyword

1. `sealed`: Indicates that a class or interface may only be extended/implemented by named classes or interfaces -> always before `class` keyword
2. `permits`: Used with the `sealed` keyword to list the classes and interfaces allowed
3. `non-sealed`: Applied to a class or interface that extends a sealed class, indicating that it can be extended by unspecified classes

Sealed classes are commonly declared with the `abstract` modifier, although this is certainly not required.

A sealed class needs to be declared (and compiled) in the same package as its direct subclasses. But what about the subclasses themselves? They must each extend the sealed class.

Every class that directly extends a sealed class must specify exactly one of the following three modifiers:

1. `final` -> A sealed class with only `final` subclasses has a fixed set of types, which is similar to an enum with a fixed set of values.
2. `sealed` -> The `sealed` modifier applied to a subclass means the same kind of rules that we applied to the parent class must be present.
3. `non-sealed` -> The `non-sealed` modifier is used to open a sealed parent class to potentially unknown subclasses.

You can omit to use `permit` keyword, only the child classes are in same file or the subclasses are nested.

Also, the interfaces can be sealed.

```
// Sealed interface

public sealed interface Swims permits Duck, Swan, Floats {}

// Classes permitted to implement sealed interface
public final class Duck implements Swims {}
public final class Swan implements Swims {}

// Interface permitted to extend sealed interface
public non-sealed interface Floats extends Swims {}
```

Also, you can use for `switch` pattern matching

```
abstract sealed class Fish permits Trout, Bass {}
final class Trout extends Fish {}
final class Bass extends Fish {}

---------------------------------------------------------


public String getType(Fish fish) {
   return switch (fish) {
      case Trout t -> "Trout!";
      case Bass b -> "Bass!";
   };
}
```

Sealed Class Rules

- Sealed classes are declared with the `sealed` and `permits` modifiers.

- Sealed classes must be declared in the same package or named module as their direct subclasses.
- Direct subclasses of sealed classes must be marked `final`, `sealed`, or `non-sealed`. For interfaces that extend a sealed interface, only `sealed` and `non-sealed` modifiers are permitted.
- The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
- Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them.

Encapsulating Data with Records

POJO - Plain Old Java Object

Class used to model and pass data around, often with few or no complex methods.

Encapsulation

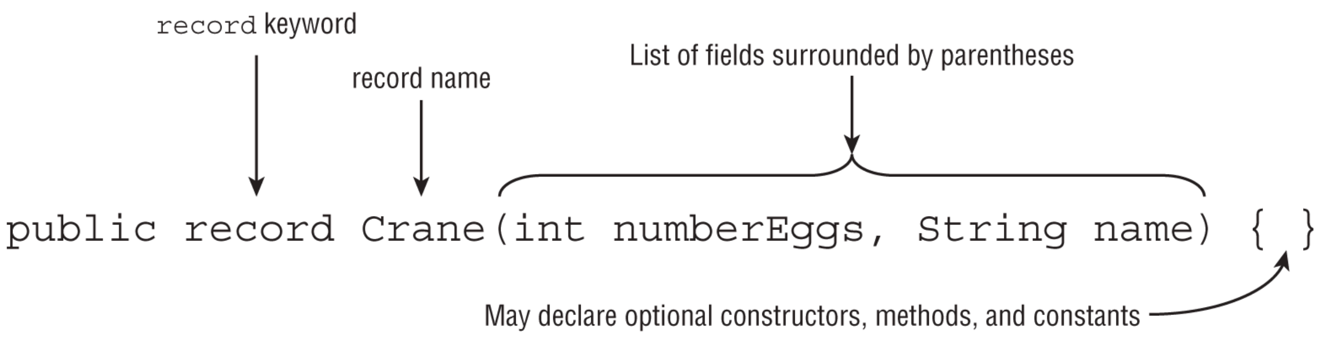
Is a way to protect class members by restricting access to them.
Encapsulation is about protecting a class from unexpected use. It also allows us to modify the methods and behavior of the class later without someone already having direct access to an instance variable within the class.

```java
public final class Crane {
   private final int numberEggs;
   private final String name;
   public Crane(int numberEggs, String name) {
      if (numberEggs >= 0) this.numberEggs = numberEggs;  // guard
      else throw new IllegalArgumentException();
      this.name = name;
   }
   public int getNumberEggs() {         // getter
      return numberEggs;
   }
   public String getName() {            // getter
      return name;
   }
}
```

Record

Is a special type of data-oriented class in which the compiler inserts boilerplate code for you.

The compiler inserts *useful* implementations of the `Object` methods `equals()`, `hashCode()`, and `toString()`.



record keyword
record name
List of fields surrounded by parentheses
public record Crane(int numberEggs, String name) { }
May declare optional constructors, methods, and constants

Members Automatically Added to Records

- **Constructor**: A constructor with the parameters in the same order as the record declaration
- **Accessor method**: One accessor for each field
- `equals()`: A method to compare two elements that returns `true` if each field is equal in terms of `equals()`
- `hashCode()`: A consistent `hashCode()` method using all of the fields
- `toString()`: A `toString()` implementation that prints each field of the record in a convenient, easy-to-read format

Long Constructor	Compact Constructor

public record Crane(int numberEggs, String name) {
  public Crane(int numberEggs, String name) {
    if (numberEggs < 0) throw new IllegalArgumentException();
    this.numberEggs = numberEggs;
    this.name = name;
  }
}
	public record Crane(int numberEggs, String name){
	  public Crane{
	    if(numberEggs < 0) throw new IllegalArgumentException();
	    name=name.toUpperCase();
	  }
	}

You should be aware of the following rules when working with pattern matching and records:

- If any field declared in the record is included, then all fields must be included.
- The order of fields must be the same as in the record.
- The names of the fields do not have to match.
- At compile time, the type of the field must be compatible with the type declared in the record.
- The pattern may not match at runtime if the record supports elements of various types.

Nested Classes

Nested class

A class that is defined within another class.

Types of nested class:

- *Inner class*: A non-`static` type defined at the member level of a class

Inner classes have the following properties:

- Can be declared `public`, `protected`, package, or `private`
- Can extend a class and implement interfaces
- Can be marked `abstract` or `final`
- Can access members of the outer class, including `private` members

```
public class Home {
   private String greeting = "Hi";  // Outer class instance variable

   protected class Room {           // Inner class declaration
      public int repeat = 3;
      public void enter() {
         for (int i = 0; i < repeat; i++) greet(greeting);
      }
      private static void greet(String message) {
         System.out.println(message);
      }
   }

   public void enterRoom() {        // Instance method in outer class
      var room = new Room();        // Create the inner class instance
      room.enter();
   }
   public static void main(String[] args) {
      var home = new Home();        // Create the outer class instance
      home.enterRoom();
} }
```

Inner classes can have the same variable names as outer classes, making scope a little tricky. There is a special way of calling `this` to say which variable you want to access.

- *Static nested class:* A `static` type defined at the member level of a class

A `static` nested class can be instantiated without an instance of the enclosing class. In other words, it is like a top-level class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it.
- It can additionally be marked `private` or `protected`.
- The enclosing class can refer to the fields and methods of the `static` nested class.

```java
public class Park {
   static class Ride {
      private int price = 6;
   }
   public static void main(String[] args) {
      var ride = new Ride();
      System.out.println(ride.price);
} }
```

- *Local class*: A class defined within a method body

Local classes have the following properties:

- Do not have an access modifier.
- Can be declared `final` or `abstract`.
- Can include instance and `static` members.
- Have access to all fields and methods of the enclosing class (when defined in an instance method).
- Can access `final` and effectively final local variables.

```java
public class PrintNumbers {
   private int length = 5;
   public void calculate() {
      final int width = 20;
      class Calculator {
         public void multiply() {
            System.out.print(length * width);
         }
      }
      var calculator = new Calculator();
      calculator.multiply();
   }
   public static void main(String[] args) {
      var printer = new PrintNumbers();
      printer.calculate();  // 100
   }
}
```

- *Anonymous class*: A special case of a local class that does not have a name

```
public class ZooGiftShop {
   abstract class SaleTodayOnly {
      abstract int dollarsOff();
   }
   public int admission(int basePrice) {
      SaleTodayOnly sale = new SaleTodayOnly() {
         int dollarsOff() { return 3; }
      };  // Don't forget the semicolon!
      return basePrice - sale.dollarsOff();
} }


-------------------------------------------


public class ZooGiftShop {
   interface SaleTodayOnly {
      int dollarsOff();
   }
   public int admission(int basePrice) {
      SaleTodayOnly sale = new SaleTodayOnly() {
         public int dollarsOff() { return 3; }
      };
      return basePrice - sale.dollarsOff();
} }
```

Prior to Java 8, anonymous classes were frequently used for asynchronous tasks and event handlers. For example, the following shows an anonymous class used as an event handler in a JavaFX application:

Polymorphism

Polymorphism

The property of an object to take on many different forms.
A Java object may be accessed using the following:

- A reference with the same type as the object
- A reference that is a superclass of the object
- A reference of an interface the object implements or inherits

```
public class Primate {
   public boolean hasHair() {
      return true;
   }
}
```

```
public interface HasTail {
   public abstract boolean isTailStriped();
}

public class Lemur extends Primate implements HasTail {
   public boolean isTailStriped() {
      return false;
   }
   public int age = 10;
   public static void main(String[] args) {

      Lemur lemur = new Lemur();
      System.out.println(lemur.age);

      HasTail hasTail = lemur;
      System.out.println(hasTail.isTailStriped());

      Primate primate = lemur;
      System.out.println(primate.hasHair());
   } }


------------------------------------------------------
10
false
true
```

Polymorphism enables an instance of `Lemur` to be reassigned or passed to a method using one of its supertypes, such as `Primate` or `HasTail`.
Once the object has been assigned to a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast.

Object vs reference

All objects are accessed by reference, so as a developer you never have direct access to the object itself.
We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

Casting Object

Once we changed the reference type, though, we lost access to more specific members defined in the subclass that still exist within the object.

```
Lemur lemur = new Lemur();
Primate primate = lemur;         // Implicit Cast to supertype
Lemur lemur2 = (Lemur)primate;  // Explicit Cast to subtype
Lemur lemur3 = primate;          // DOES NOT COMPILE (missing cast)
```

We summarize these concepts into a set of rules:

1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown.
4. The compiler disallows casts to unrelated types.

Casting Interfaces

While a given class may not implement an interface, it's possible that some subclass may implement the interface. When holding a reference to a particular class, the compiler doesn't know which specific subtype it is holding.

```
interface Canine {}
interface Dog {}
class Wolf implements Canine {}

public class BadCasts {
   public static void main(String[] args) {
      Wolf wolfy = new Wolf();
      Dog badWolf = (Dog)wolfy; // compileaza, dar arunca eroarea ClassCastException, pt ca nu are trb Wolf cu interfata Dog
   } }
```

The *instanceOf* operator

The `instanceof` operator can be used to check whether an object belongs to a particular class or interface and to prevent a `ClassCastException` at runtime.

```
class Rodent {}

public class Capybara extends Rodent {
   public static void main(String[] args) {
      Rodent rodent = new Rodent();
      var capybara = (Capybara)rodent;  // ClassCastException
   }
}

---------------use this instead--------------------

```

```java
      if(rodent instanceof Capybara c) {
         // Do stuff
      }
```

Just as the compiler does not allow casting an object to unrelated types, it also does not allow `instanceof` to be used with unrelated types.

Polymorphism and Method overriding

Polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class.
Polymorphism's ability to replace methods at runtime via overriding is one of the most important properties of Java. It allows you to create complex inheritance models with subclasses that have their own custom implementation of overridden methods. It also means the parent class does not need to be updated to use the custom or overridden method. If the method is properly overridden, then the overridden version will be used in all places that it is called.

Overriding vs. Hiding Members

While method overriding replaces the method everywhere it is called, `static` method and variable hiding do not.
Unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.