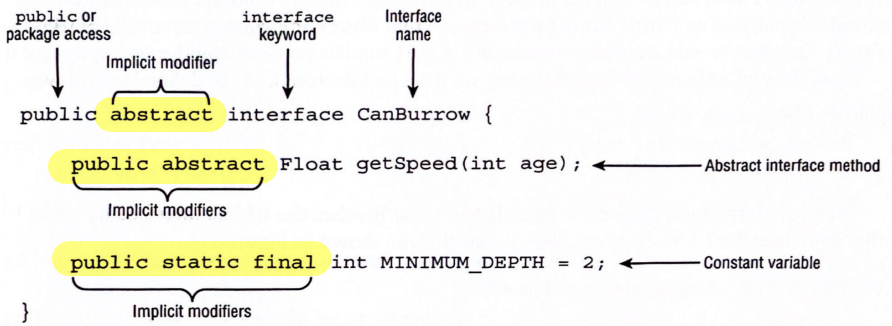
Chapter 7

Beyond Classes

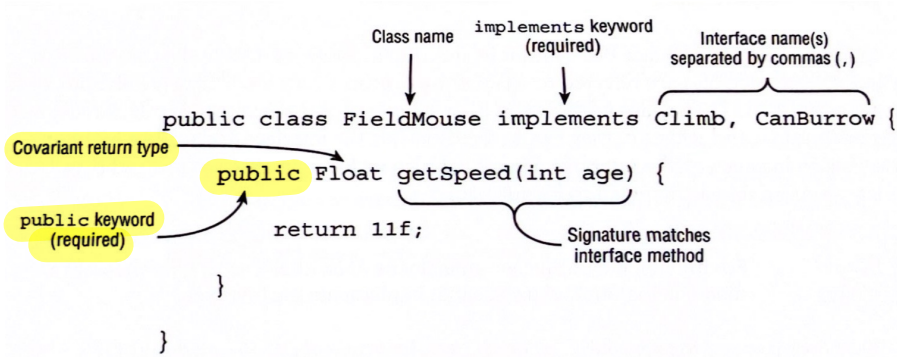
Implementing Interface

Declaring & Using an Interface



Cannot be marked as ~~final~~

```
public interface Climb {
    Number getSpeed(int age);
}
```



Extending an Interface

An interface can extend multiple interfaces

```
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() { }
}
```

Inheriting an Interface

Mixing Class & Interface Keywords

```
public interface CanRun { }
public class Cheetah extends CanRun { }//Error
public class Hyena { }
public interface HasFur extends Hyena { }//Error
```

Inheriting Duplicate Abstract Methods

```
public interface Herbivore { public void eatPlants(); }
public interface Omnivore { public int eatPlants(); }
public class Tiger implements Herbivore, Omnivore {//Error
   ...
}
```

Can use covariant return types

Inserting Implicit Modifiers

- 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`.

Conflicting Modifiers

```
public interface Dance {
   private int count = 4; //Error
   protected void step(); //Error
}
```

Declaring Concrete Interface Methods

	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	Class	—	`public static final`	Yes
`abstract` method	Instance	—	`public abstract`	No
`default` method	Instance	`default`	`public`	Yes
`static` method	Class	`static`	`public`	Yes
`private` method	Instance	`private`	—	Yes
`private static` method	Class	`private static`	—	Yes

Writing a default Interface Method

```
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;
  }
}
```

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.

```
public interface Walk {
  public default int getSpeed() {
    return 5;
  }
}
public interface Run {
  public default int getSpeed() {
    return 10;
  }
}
public class Cat implements Walk, Run { }//Rule #6
```

```
public class Cat implements Walk, Run {
  public int getSpeed() {
    return 1;
  }
  public int getWalkSpeed() {
    //Calling a Hidden default Method
    return Walk.super.getSpeed();
  }}
```

Declaring static Interface Methods

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 Bunny implements Hop {
   public void printDetails() {
      //Error:
      //System.out.print(getJumpHeight());
      //Correct:
      System.out.print(Hop.getJumpHeight());
   }}
```

Reusing Code with private Interface Methods

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.

```
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");
      } } }
```

Calling Abstract Methods

```
public interface ZooRenovation {
   public String projectName();
   abstract String status();
   default void printStatus() {
      System.out.print("The " + projectName()
         + "project " + status());
   }}
```

Reviewing Interface Members

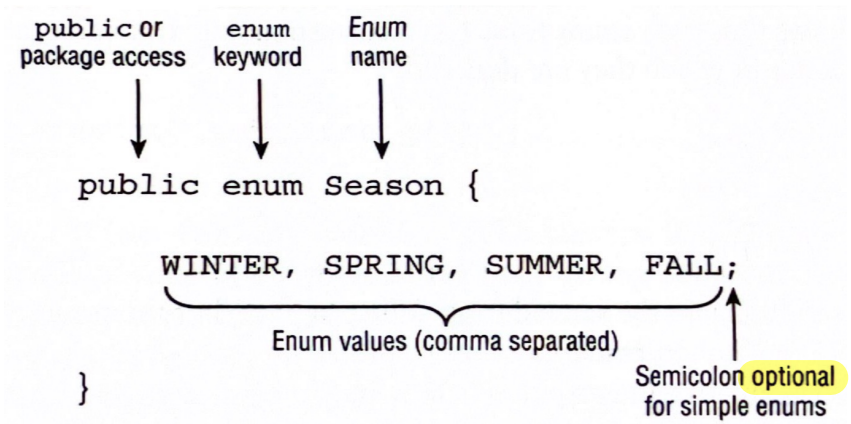
	Accessible from `default` and `private` methods within the interface?	Accessible from `static` methods within the interface?	Accessible from methods in classes inheriting the interface?	Accessible without an instance of the interface?
Constant variable	Yes	Yes	Yes	Yes
`abstract` method	Yes	No	Yes	No
`default` method	Yes	No	Yes	No
`static` method	Yes	Yes	Yes (interface name required)	Yes (interface name required)
`private` method	Yes	No	No	No
`private static` method	Yes	Yes	No	No

- Treat `abstract`, `default`, and non-`static` `private` methods as belonging to an instance of the interface.
- Treat `static` methods and variables as belonging to the interface class object.
- All `private` interface method types are only accessible within the interface declaration.

Non-static interface methods with a body must be explicitly marked private or default

Working with Enums

Creating Simple Enums




```
var s = Season.SUMMER;
System.out.println(Season.SUMMER);//SUMMER
System.out.println(s == Season.SUMMER);//true
```

Cannot extend an enum

```
public enum ExtendedSeason extends Season {}
```

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

```
Season summer = Season.SUMMER;
switch(summer) {
   case WINTER:
      System.out.print("Get out the sled!");
      break;
   case SUMMER:
      System.out.print("Time for the pool!");
      break;
   default:
      System.out.print("Is it summer yet?");
}
```

the code will not compile if we use Season.__

Adding Constructors, Fields, and Methods

```
1: public enum Season {
2:    WINTER("Low"), SPRING("Medium"), SUMMER("High"), FALL("Medium");
3:    private final String expectedVisitors;
4:    private Season(String expectedVisitors) {
5:       this.expectedVisitors = expectedVisitors;
6:    }
7:    public void printExpectedVisitors() {
8:       System.out.println(expectedVisitors);
9:    } }
```

constructor calls

It is required if there is anything in the enum besides the values

All enum constructors are implicitly private. An enum constructor will not compile if it contains a public or protected modifier.

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.

```
public enum OnlyOne {
   ONCE(true);
   private OnlyOne(boolean b) {
      System.out.print("constructing,");
   }
}

public class PrintTheOne {
   public static void main(String[] args) {
      System.out.print("begin,");
      OnlyOne firstCall = OnlyOne.ONCE;   // Prints constructing,
      OnlyOne secondCall = OnlyOne.ONCE;  // Doesn't print anything
      System.out.print("end");
   }
}
```

// begin, constructing, end

Call an enum method:

Season.SUMMER.printExpectedVisitors();

Create an implementation for all values & override it only for the special cases:

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

```
public enum Season {
  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();
}
```

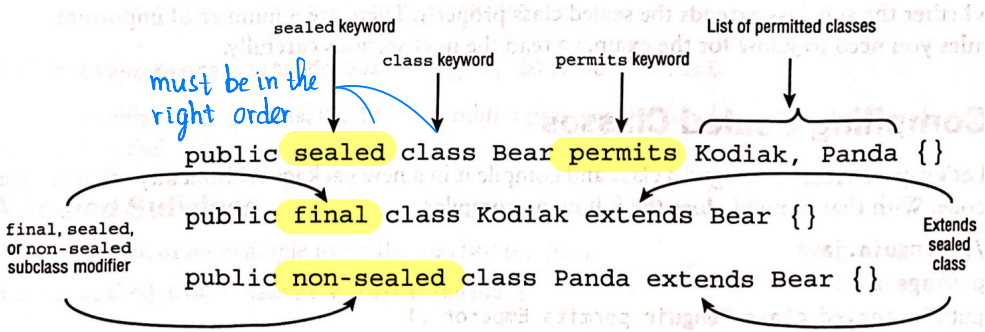
each & every enum value is required to implement it

An enum can even implement an interface

```
public interface Weather {
  int getAverageTemperature();
}
public enum Season implements Weather {
  WINTER, SPRING, SUMMER, FALL;
  public int getAverageTemperature(){
    return 30;
  }
}
```

Sealing Classes

Declaring a Sealed Class



Compiling Sealed Classes

A sealed class needs to be declared (and compiled) in the same package as its direct subclasses. The subclasses must each extend the sealed class.

Specifying the Subclass Modifiers

Every class that directly extends a sealed class must specify exactly one of the following 3 modifiers:

final, sealed, or non-sealed

A final Subclass

```
public sealed class Antelope permits Gazelle { }
public final class Gazelle extends Antelope { }
public class George extends Gazelle { } //Error
```

A sealed Subclass

```
public sealed class Mammal permits Equine { }
public sealed class Equine extends Mammal
                        permits Zebra { }
public final class Zebra extends Equine { }
```

A non-sealed Subclass

```
public sealed class Wolf permits Timber { }
public non-sealed class Timber extends Wolf { }
public class MyWolf extends Timber { }
```

Omitting the permits Clause

Location of direct subclasses	`permits` clause
In a different file from the sealed class	Required
In the same file as the sealed class	Permitted, but not required
Nested inside of the sealed class	Permitted, but not required

```
// Snake.java
public sealed class Snake { }
final class Cobra extends Snake { }

// Snake.java
public sealed class Snake {
  final class Cobra extends Snake { }
}
```

if permit, must be: permits **Snake.Cobra**

Sealing Interfaces

```
// 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 Swim { }
```

cannot be marked **final**

My Notes

```
5  sealed class Friendly extends Mandrill permits Silly {}
6
7  non-sealed class Silly {}
8
```

2 lines contain error

must extend

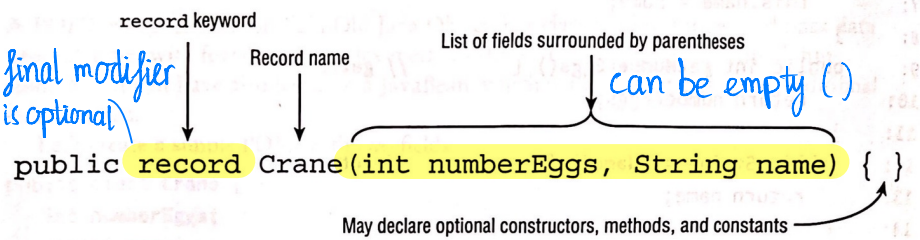
```
5  sealed class Friendly extends Mandrill permits Silly {}
6
7  final class Silly {}
8
```

1 line contains error

still compiles if no extends

Encapsulating Data with Records

Applying Records



Members automatically added to records:

- Constructor
- Accessor methods (getters)
- equals()
- hashCode()
- toString()

```
var mommy = new Crane(4, "Cammy");
System.out.println(mommy.numberEggs()); // 4
System.out.println(mommy.name());       // Cammy
```

no ~~get~~ prefix

```
var father = new Crane(0, "Craig");
System.out.println(father);                    // Crane[numberEggs=0, name=Craig]

var copy = new Crane(0, "Craig");
System.out.println(copy);                      // Crane[numberEggs=0, name=Craig]
System.out.println(father.equals(copy));       // true
System.out.println(father.hashCode() + ", " + copy.hashCode()); // 1007, 1007
```

Understanding Record Immutability

- Records don't have setters
- Every field is inherently final & cannot be modified after it has been written in the constructor
- Records are implicitly final → cannot extend or inherit a record
- Can implement a regular or sealed interface

```
public interface Bird { }
public record Crane(int numberEggs, String name)
                                    implements Bird { }
```

Declaring Constructors

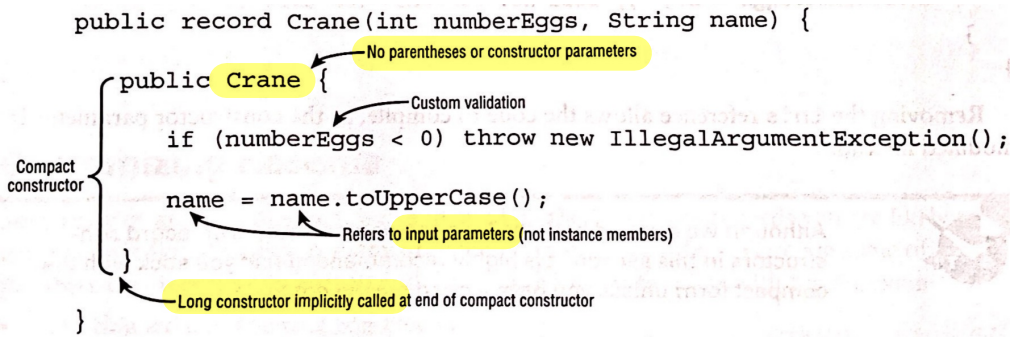
The Long Constructor

```
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;
    }
}
```

- The compiler will not insert a constructor if we define one with the same list of parameters in the same order
- Since each field is final, the constructor must set every field

```
public record Crane(int numberEggs, String name) {
    public Crane(int numberEggs, String name) { }
}
```

Compact Constructors



Transforming Parameters

While compact constructors can modify the constructor parameters, they cannot modify the fields of the record

```
public record Crane (int numberEggs, String name){
  public Crane {
    this.numberEggs = 10; // Error
  }
}
```

Overloaded Constructors

The first line of an overloaded constructor must be an explicit call to another constructor via this()

If there are no other constructors, the long constructor must be called

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
      this(numberEggs + 1, firstName + " " + lastName);
      numberEggs = 10; // NO EFFECT (applies to parameter, not instance field)
      this.numberEggs = 20; // DOES NOT COMPILE
   }
}
```

```
public record Crane(int numberEggs, String name) {
   public Crane(String name) {
      this(1);  // DOES NOT COMPILE
   }
   public Crane(int numberEggs) {
      this("");  // DOES NOT COMPILE
   }
}
```

Customizing Records

Some of the members that records can include

- Overloaded and compact constructors
- Instance methods including overriding any provided methods (accessors, `equals()`, `hashCode()`, `toString()`)
- Nested classes, interfaces, annotations, enum, and records

```
public record Crane(int numberEggs, String name){
   @Override public int numberEggs()
   { return 10; }
   @Override public String toString()
   { return name; }
}
```

```
public record Crane(int numberEggs, String name){
   private static int type = 10;
   public int size; //Error
   private boolean friendly; //Error
} //also do not support instance initializers
```

Creating Nested Classes

Declaring an Inner Class

A non-static type defined at the member level of a class

Properties:

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

```
1:  public class Home {
2:     private String greeting = "Hi";  // Outer class instance variable
3:
4:     protected class Room {           // Inner class declaration
5:        public int repeat = 3;
6:        public void enter() {
7:           for (int i = 0; i < repeat; i++) greet(greeting);
8:        }
9:        private static void greet(String message) {
10:          System.out.println(message);
11:       }
12:    }
13:
14:    public void enterRoom() {        // Instance method in outer class
15:       var room = new Room();        // Create the inner class instance
16:       room.enter();
17:    }
18:    public static void main(String[] args) {
19:       var home = new Home();        // Create the outer class instance
20:       home.enterRoom();
21: } }
```

Instantiating an Instance of an Inner Class

```
public static void main(String[] args) {
   var home = new Home();
   Room room = home.new Room();
   room.enter();
}
```

Referencing Members of an Inner Class

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);          // 30
9:              System.out.println(this.x);     // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

can have the same name

can write B, because it is available at the member level of A

A.B.C is necessary, because C is too deep

Creating a static Nested Class

A static type defined at the member level. Properties:

- 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 & methods of the static nested class

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

Writing a Local Class

A nested class defined within a method. Properties:

- Do not have an access modifier
- Can be declared final or abstract
- Have access to all fields & methods of the enclosing class (when defined in an instance method)
- Can access final & effectively final local variable

```
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
  }
}
```

```
public void processData() {
  final int length = 5;
  int width = 10;
  int height = 2;
  class VolumeCalculator {
    public int multiply() {
      return length * width * height; //Error
    }
  }
  width = 2;
}
```

Defining an Anonymous Class

A specialized form of a local class that doesn't have a name

Must extend an existing class or implement an existing interface

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

Can define anonymous classes outside a method body

```
public class Gorilla {
  interface Climb { }
  Climb climbing = new Climb() { };
}
```

Reviewing Nested Classes

Permitted modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`final`	Yes	Yes	Yes	No

	Inner class	`static` nested class	Local class	Anonymous class
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method?	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Understanding Polymorphism

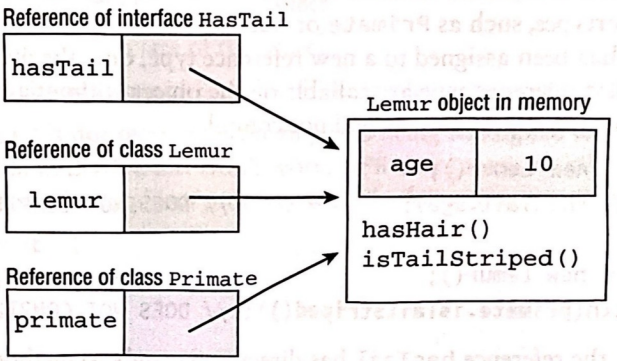
Object vs. Reference

- The type of the object determines which properties exists within the object in memory
- The type of the reference to the object determines which methods & variables are accessible to the Java program

```
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());
   } }
```

HasTail hasTail
= new Lemur();
hasTail.age; //Error

Primate primate
= new Lemur();
primate.isTailStriped();
//Error



Casting Objects

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.

```
Lemur lemur = new Lemur();
Primate primate = lemur; // Implicit cast
Lemur lemur2 = (Lemur)primate; // Explicit cast
Lemur lemur3 = primate; // Error, missing cast
```

Disallowed Casts

```
public class Bird { }
public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    Bird bird = (Bird) fish; //Error
  }}
```

Casting Interfaces

```
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;
  }} //compiles, but throws ClassCastExc. at runtime
```

if final

Error

The instanceof Operator

```
class Rodent { }
public class Capybara extends Rodent {
  public static void main(String[] args) {
    Rodent rodent = new Rodent();
    if (rodent instanceof Capybara c) {...}
  } }
```

explicit cast to Capybara will cause ClassCastException at runtime

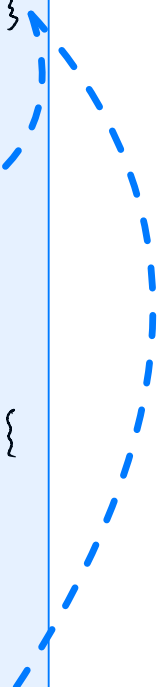
```
public class Bird { }
public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    if (fish instanceof Bird b) { //unrelated types
      // Do stuff
  } } }
```

Polymorphism and Method Overriding

```
class Penguin {
  public int getHeight() { return 3; }
  public void printInfo() {
    System.out.print(this.getHeight());
  }
}
public class EmperorPenguin extends Penguin {
  public int getHeight() { return 8; }
  public static void main(String [] fish) {
    new EmperorPenguin().printInfo();//8
  }
}
```

Overriding vs. Hiding Members

```
class Penguin {
  public static int getHeight() { return 3; }
  public void printInfo() {
    System.out.println(this.getHeight());
  }
}

public class CrestedPenguin extends Penguin {
  public static int getHeight() { return 8; }
  public static void main(String... fish) {
    new CrestedPenguin().printInfo(); //3
  }
}
```



```
class Marsupial {
   protected int age = 2;
   public static boolean isBiped() {
      return false;
   } }

public class Kangaroo extends Marsupial {
   protected int age = 6;
   public static boolean isBiped() {
      return true;
   }

   public static void main(String[] args) {
      Kangaroo joey = new Kangaroo();
      Marsupial moey = joey;
      System.out.println(joey.isBiped());
      System.out.println(moey.isBiped());
      System.out.println(joey.age);
      System.out.println(moey.age);
   } }
```

true

false

6

2

Since static methods/variables can only be hidden, Java uses the reference type to determine:

- which version of isBiped() should be called
- which value of age to output

Review Questions

1/ ~~A~~ B D (same name as instance field)

2/ A B D E ✓

3/ C ✓

4/ C ✓

5/ ~~F~~ **E** (error at class declaration)

6/ D E ~~F~~ (no invalid method override)

7/ E ✓

8/ A **B C** (not require method→public)

9/ A E F ✓

10/ A B C E ✓

11/ ~~C D E~~ **B** (private constr. can be used within class)

12/ A B E ✓

13/ F ✓ (null in switch → NullPointerEx.)

14/ A C D E ✓

15/ ~~C D F~~ **G** (anonymous class can't inherit final class)

16/ ~~C~~ **E** (static nested class can't access instance mem)

17/ G **E**

18/ E ✓

19/ G ✓

20/ D ~~E~~ F **A**

21/ ~~A C E~~ **F** (1st line must call to another constr.)

22/ C D ~~F~~ **G**

23/ ~~C~~ **D** → focus on syntax

24/ B E ~~F~~ (can have ; after method body)

25/ ~~C D~~ **B** (focus on syntax)

26/ ~~A~~ F **C** (enum constr. is implicitly private)

27/ B C D G ✓

28/ A B D ✓

29/ F ✓

30/ C ~~F~~ **E**

46,7%