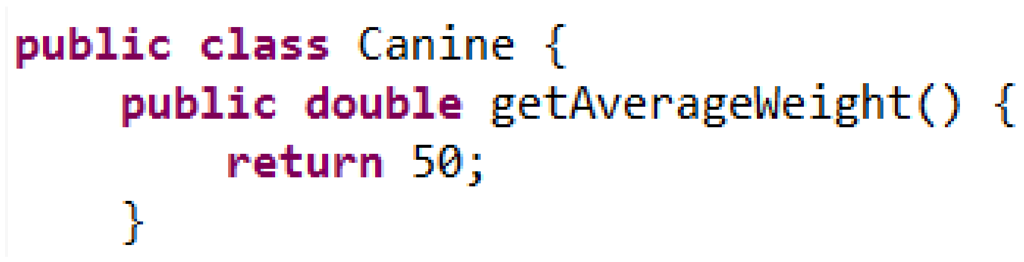
METHOD OVERRIDING AND POLYMORPHISM REITERATION

OUTLINE

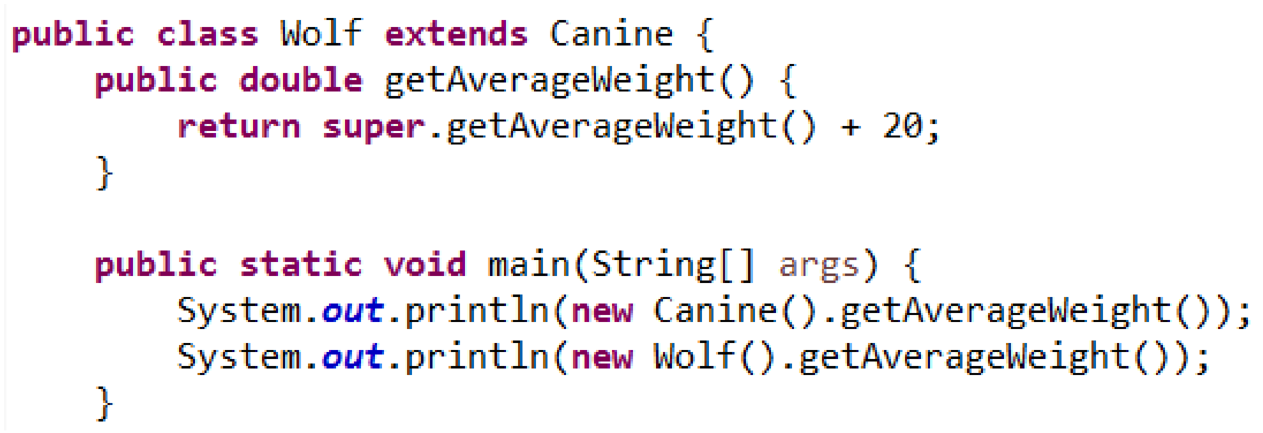
1. METHOD OVERRIDING
2. POLYMORPHISM

1) OVERRIDING METHODS

- Inheriting a class grants us access to the public and protected members of the parent class, but also sets the stage for collisions between methods defined in both the parent class and the subclass.
- What if there is a method defined in both the parent and child class?
- For example, we may want to define a new version of an existing method in a child class that makes use of the definition in the parent class.
- In this case, we can override a method by declaring a new method with the signature and return type as the method in the parent class.
- When we override a method, we may reference the parent version of the method using the `super` keyword. In this manner, the keywords `this` and `super` allow us to select between the current and parent version of a method, respectively



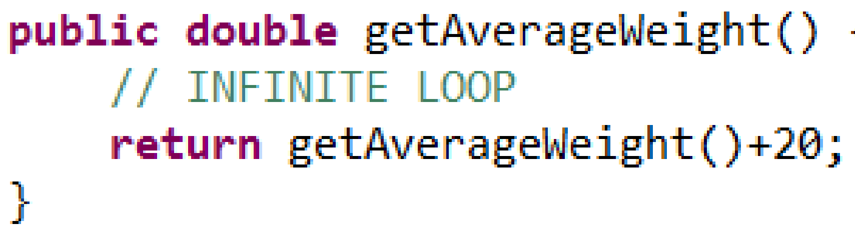
```java
public class Canine {
    public double getAverageWeight() {
        return 50;
    }
```



```java
public class Wolf extends Canine {
    public double getAverageWeight() {
        return super.getAverageWeight() + 20;
    }

    public static void main(String[] args) {
        System.out.println(new Canine().getAverageWeight());
        System.out.println(new Wolf().getAverageWeight());
    }
```

1) OVERRIDING METHODS

- In this example, in which the child class `Wolf` overrides the parent class `Canine`, the method `getAverageWeight()` runs without issue and outputs the following:
  - 50.00
  - 70.00
- We might be wondering, if the use of **`super`** in the child's method was required?
- For example, what would the following code output if we removed the **`super`** keyword in the `getAverageWeight()` method of the `Wolf` class?



```java
public double getAverageWeight() {
    // INFINITE LOOP
    return getAverageWeight()+20;
}
```

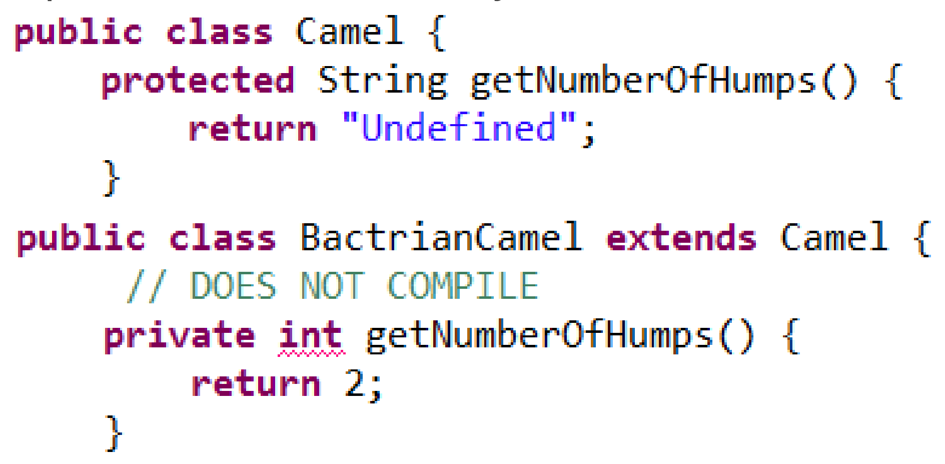
- In this example, the compiler would not call the parent `Canine` method.
- It would call the current `Wolf` method since it would think we were executing a recursive call.
- A recursive method is one that calls itself as part of execution.
- **A recursive function must have a termination condition.**
- In this example, there is no termination condition, therefore, the application will attempt to call itself infinitely and produce a stack overflow error at runtime.

1) OVERRIDING METHODS

- Overriding a method is not without limitations, though.
- The compiler performs the following checks when you override a non-private method:
    1. The method in the child class must have the same signature as the method in the parent class.
    2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
    3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
    4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as covariant return types.
- The first rule of overriding a method is somewhat self-explanatory.
- If two methods have the same name but different signatures, the methods are overloaded, not overridden.
- As we may recall from our discussion of overloaded methods, the methods are unrelated to each other and do not share any properties.

1) OVERRIDING METHODS

- Let’s review some examples of the last three rules of overriding methods so we can learn to spot the issues when they arise:

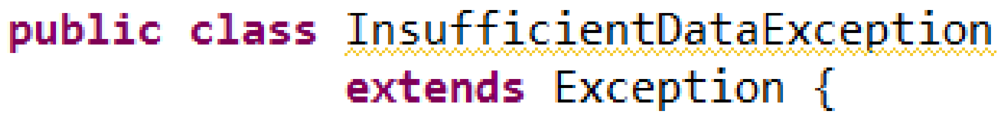


```
public class Camel {
    protected String getNumberOfHumps() {
        return "Undefined";
    }
public class BactrianCamel extends Camel {
     // DOES NOT COMPILE
    private int getNumberOfHumps() {
        return 2;
    }
```

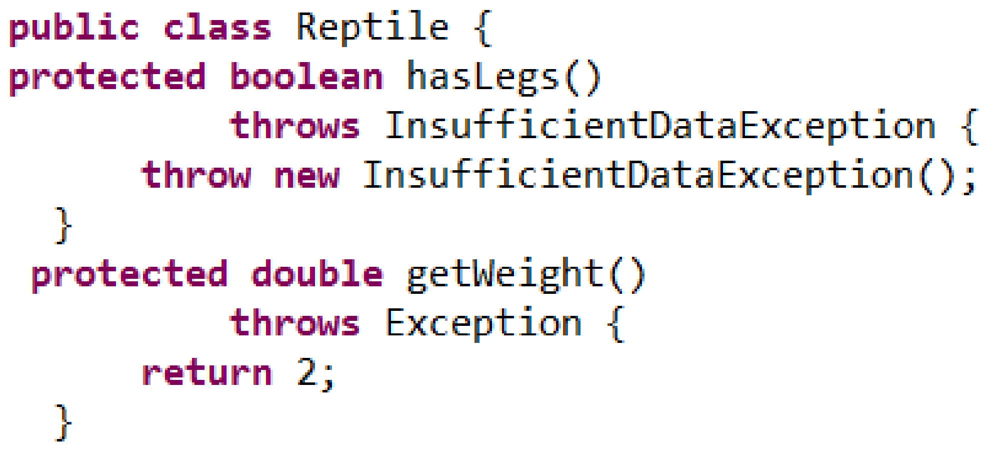
- In this example, the method in the child class doesn’t compile for two reasons.
- First, it violates the second rule of overriding methods: the child method must be at least as accessible as the parent.
- In the example, the parent method uses the **`protected`** modifier, but the child method uses the **`private`** modifier, making it less accessible in the child method than in the parent method.
- It also violates the fourth rule of overriding methods: the return type of the parent method and child method must be covariant.
- In this example, the return type of the parent method is **`String`**, whereas the return type of the child method is **`int`**, neither of which is covariant with each other.

1) OVERRIDING METHODS

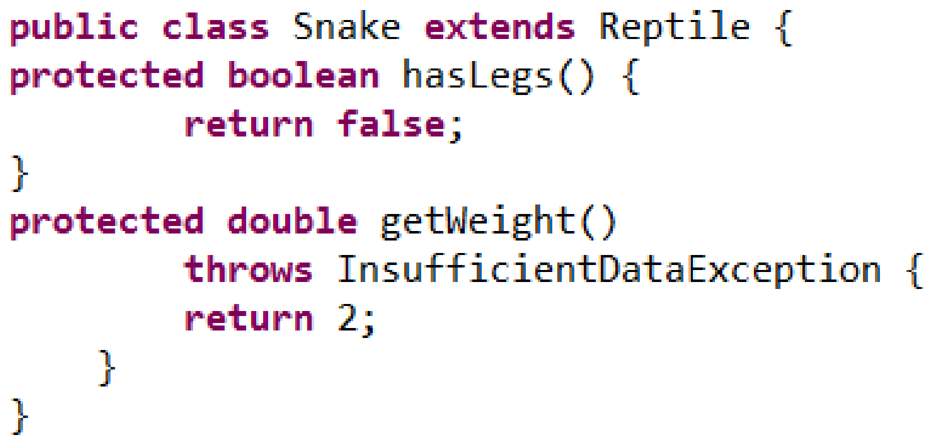
- Let’s take a look at some example methods that use exceptions:



```
public class InsufficientDataException
                extends Exception {
```



```
public class Reptile {
protected boolean hasLegs()
          throws InsufficientDataException {
      throw new InsufficientDataException();
  }
 protected double getWeight()
          throws Exception {
      return 2;
  }
```



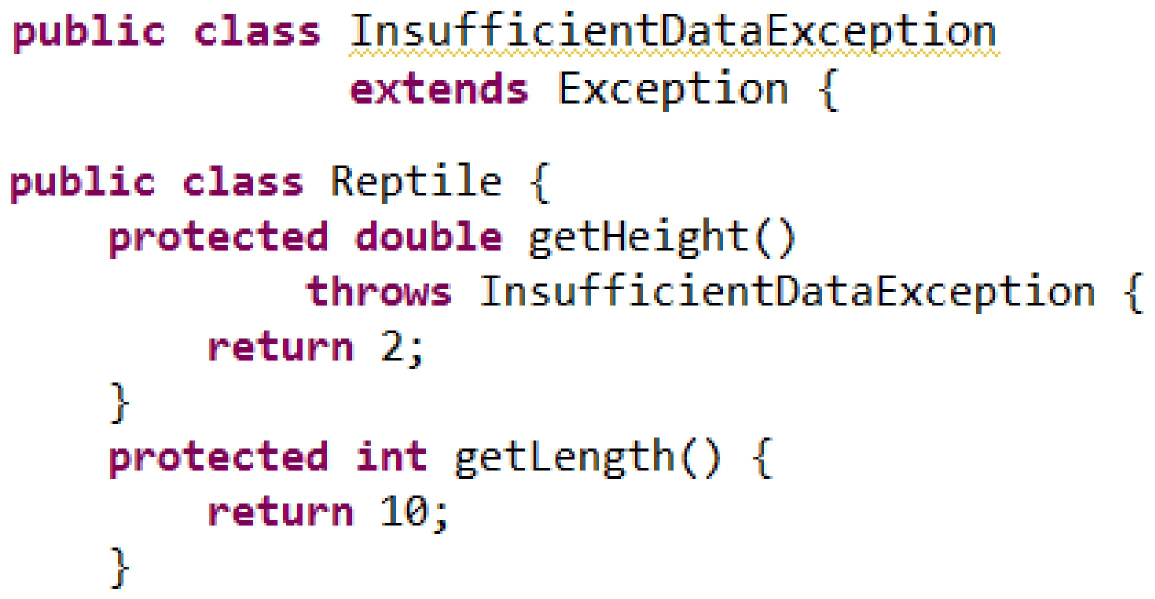
```
public class Snake extends Reptile {
protected boolean hasLegs() {
        return false;
}
protected double getWeight()
        throws InsufficientDataException {
        return 2;
    }
}
```

- In this example, both parent and child classes define two methods, `hasLegs()` and `getWeight()`.
- The first method, `hasLegs()`, **`throws`** an exception `InsufficientDataException`

1) OVERRIDING METHODS

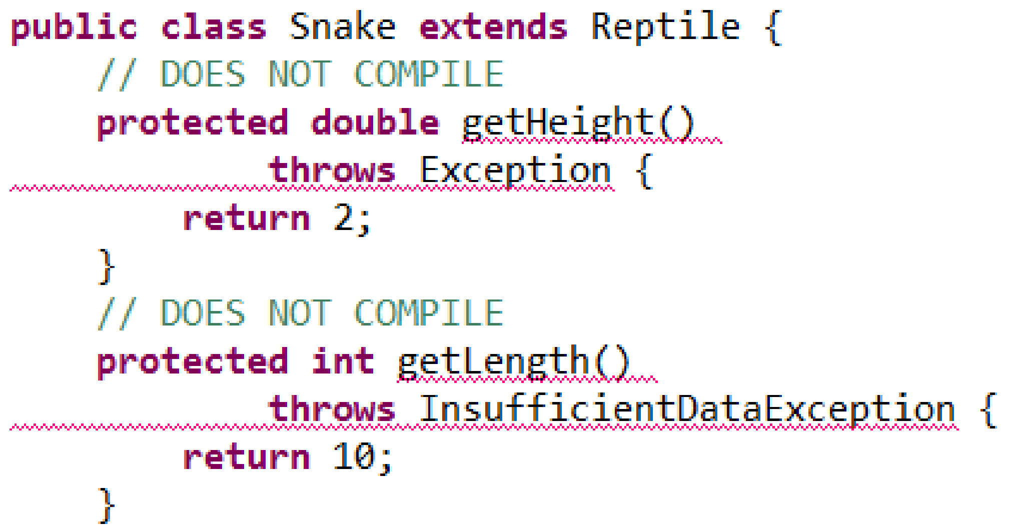
- This does not violate the third rule of overriding methods, though, as no new exception is defined.
- In other words, a child method may hide or eliminate a parent method's exception without issue.
- The second method, `getWeight()`, **`throws Exception`** in the parent class and `InsufficientDataException` in the child class.
- This is also permitted, as `InsufficientDataException` is a subclass of `Exception` by construction.
- Neither of the methods in the previous example violates the third rule of overriding methods, so the code compiles and runs without issue.

- Let's review some examples that do violate the third rule of overriding methods:



```java
public class InsufficientDataException
            extends Exception {

public class Reptile {
    protected double getHeight()
            throws InsufficientDataException {
        return 2;
    }
    protected int getLength() {
        return 10;
    }
```

1) OVERRIDING METHODS

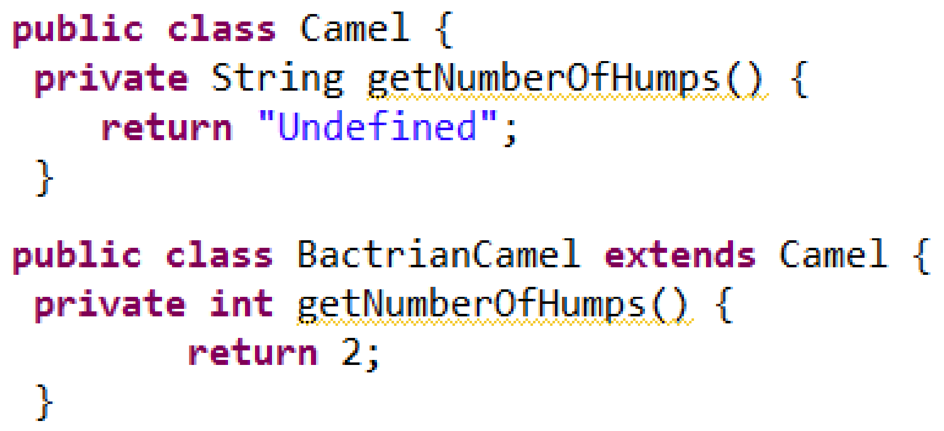


```
public class Snake extends Reptile {
    // DOES NOT COMPILE
    protected double getHeight()
            throws Exception {
        return 2;
    }
    // DOES NOT COMPILE
    protected int getLength()
            throws InsufficientDataException {
        return 10;
    }
```

- Unlike the earlier example, neither of the methods in the child class of this code will compile.
- The `getHeight()` method in the parent class throws an `InsufficientDataException`, whereas the method in the child class throws an `Exception`.
- Since `Exception` is not a subclass of `InsufficientDataException`, the third rule of overriding methods is violated and the code will not compile.
- Coincidentally, `Exception` is a superclass of `InsufficientDataException`.
- Next, the `getLength()` method doesn't throw an exception in the parent class, but it does throw an exception, `InsufficientDataException`, in the child class.
- In this manner, the child class defines a new exception that the parent class did not, which is a violation of the third rule of overriding methods.

1) OVERRIDING METHODS
1.1) REDECLARING PRIVATE METHODS

- In Java, it is not possible to override a `private` method in a parent class since the parent method is not accessible from the child class.
- Just because a child class doesn’t have access to the parent method, doesn’t mean the child class can’t define its own version of the method.
- It just means, strictly speaking, that the new method is not an overridden version of the parent class’s method.
- Java let’s us to redeclare a new method in the child class with the same or modified signature as the method in the parent class.
- This method in the child class is a separate and independent method, unrelated to the parent version’s method, so none of the rules for overriding methods are invoked.



```
public class Camel {
 private String getNumberOfHumps() {
    return "Undefined";
 }

public class BactrianCamel extends Camel {
 private int getNumberOfHumps() {
        return 2;
 }
```

- This code compiles without issue. Notice that the return type differs in the child method from `String` to `int`.
- In this example, the method getNumberOfHumps() in the parent class is hidden, so the method in the child class is a new method and not an override of the method in the parent class.

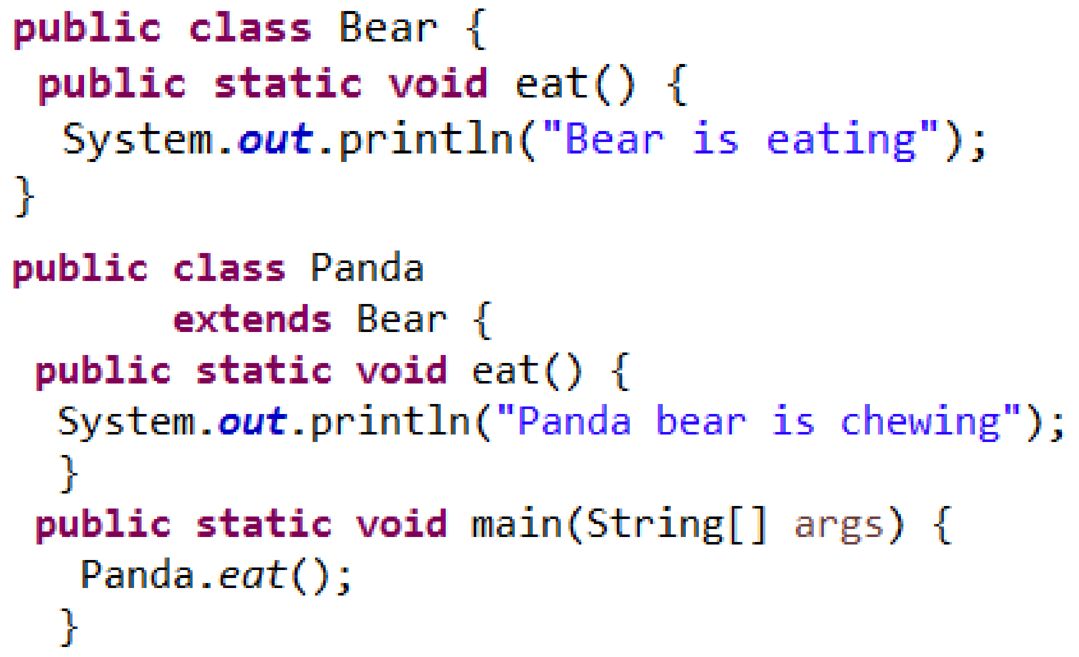
1) OVERRIDING METHODS
1.2) HIDDING STATIC METHODS

- A hidden method occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class.
- Method hiding is similar but not exactly the same as method overriding.
- First, the four previous rules for overriding a method must be followed when a method is hidden.
- In addition, a new rule is added for hiding a method, namely that the usage of the static keyword must be the same between parent and child classes. The following list summarizes the five rules for hiding a method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.
5. The method defined in the child class must be marked as static if it is marked as static in the parent class (method hiding). Likewise, the method must not be marked as static in the child class if it is not marked as static in the parent class (method overriding).

1) OVERRIDING METHODS
1.3) HIDDING STATIC METHODS

- Note that the first four are the same as the rules for overriding a method.
- Let’s review some examples of the new rule:

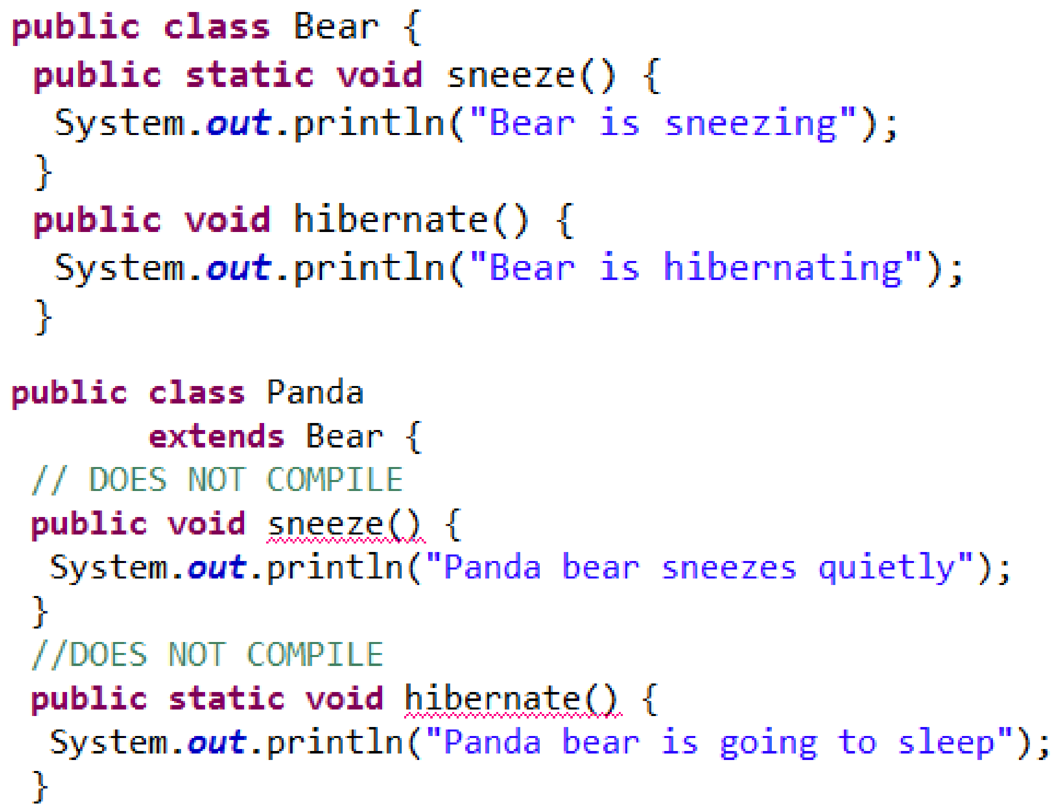


```
public class Bear {
 public static void eat() {
  System.out.println("Bear is eating");
}

public class Panda
       extends Bear {
 public static void eat() {
  System.out.println("Panda bear is chewing");
  }
 public static void main(String[] args) {
   Panda.eat();
  }
```

- In this example, the code compiles and runs without issue.
- The `eat()` method in the child class hides the `eat()` method in the parent class.
- Because they are both marked as static, this is not considered an overridden method.

1) OVERRIDING METHODS
1.4) HIDDING STATIC METHODS

- Let’s contrast this with examples that violate the fifth rule:



```
public class Bear {
 public static void sneeze() {
  System.out.println("Bear is sneezing");
 }
 public void hibernate() {
  System.out.println("Bear is hibernating");
 }

public class Panda
       extends Bear {
 // DOES NOT COMPILE
 public void sneeze() {
  System.out.println("Panda bear sneezes quietly");
 }
 //DOES NOT COMPILE
 public static void hibernate() {
  System.out.println("Panda bear is going to sleep");
 }
```

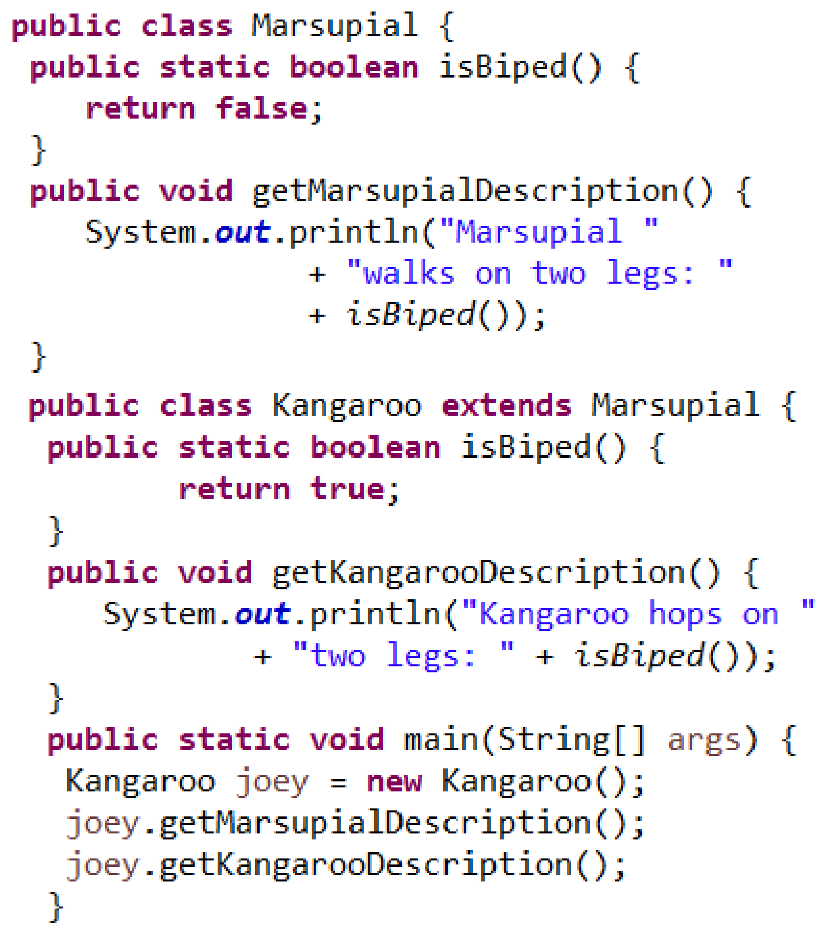
- In this example, `sneeze()` is marked as `static` in the parent class but not in the child class.
- The compiler detects that we’re trying to override a method that should be hidden and generates a compiler error.
- In the second method, `hibernate()` is an instance member in the parent class but a `static` method in the child class.
- In this scenario, the compiler thinks that we’re trying to hide a method that should be overridden and also generates a compiler error.

1) OVERRIDING METHODS
1.5) OVERRING VS HIDDING METHODS

- In our description of hiding of static methods, we indicated there was a distinction between overriding and hiding methods.
- Unlike overriding a method, in which a child method replaces the parent method in calls defined in both the parent and child, **hidden methods only replace parent methods in the calls defined in the child class**.
- At runtime the child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method.
- In this manner, the parent method is never used unless an explicit call to the parent method is referenced, using the syntax `ParentClassName.method()`.
- Alternatively, at runtime the parent version of a hidden method is always executed if the call to the method is defined in the parent class.

1) OVERRIDING METHODS
1.5) OVERRING VS HIDDING METHODS

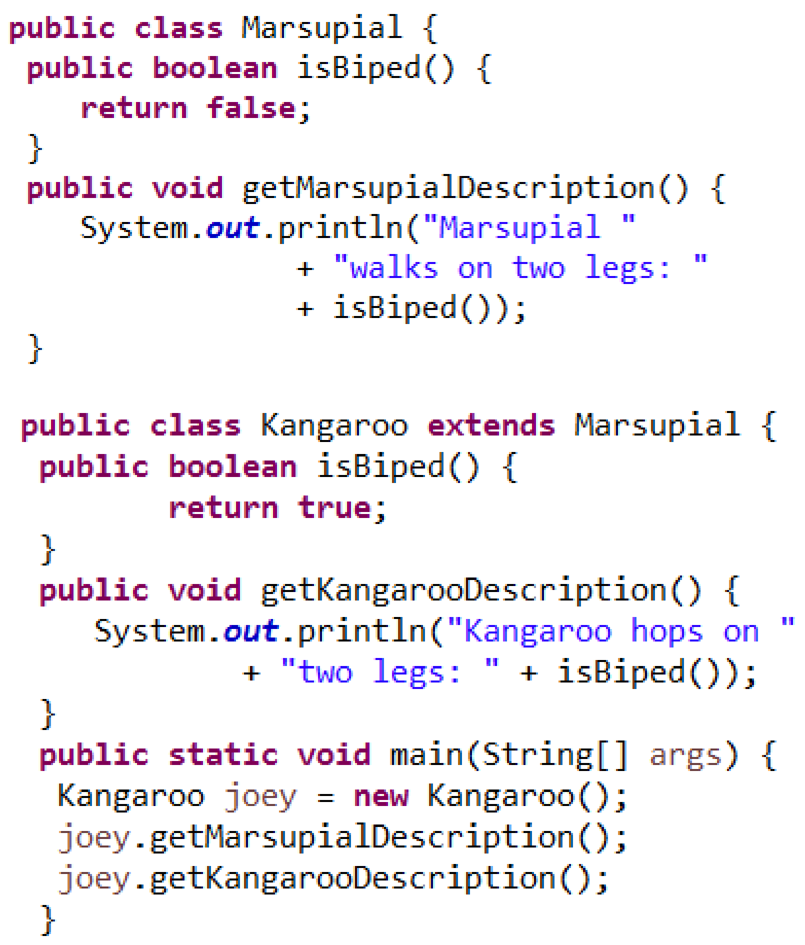
- Let’s take a look at an example:
- The code compiles and runs without issue, outputting the following:
  - `Marsupial walks on two legs: false`
  - `Kangaroo hops on two legs: true`
- Notice that `isBiped()` returns **`false`** in the parent class and **`true`** in the child class.
- In the first method call, the parent method `getMarsupialDescription()` is used.
- The `Marsupial` class only knows about `isBiped()` from its own class definition, so it outputs **`false`**.
- In the second method call, the child executes a method of `isBiped()`, which hides the parent method’s version and returns **`true`**.



```
public class Marsupial {
 public static boolean isBiped() {
    return false;
 }
 public void getMarsupialDescription() {
    System.out.println("Marsupial "
                + "walks on two legs: "
                + isBiped());
 }
public class Kangaroo extends Marsupial {
 public static boolean isBiped() {
        return true;
 }
 public void getKangarooDescription() {
    System.out.println("Kangaroo hops on "
             + "two legs: " + isBiped());
 }
 public static void main(String[] args) {
  Kangaroo joey = new Kangaroo();
  joey.getMarsupialDescription();
  joey.getKangarooDescription();
 }
```

1) OVERRIDING METHODS
1.5) OVERRING VS HIDDING METHODS

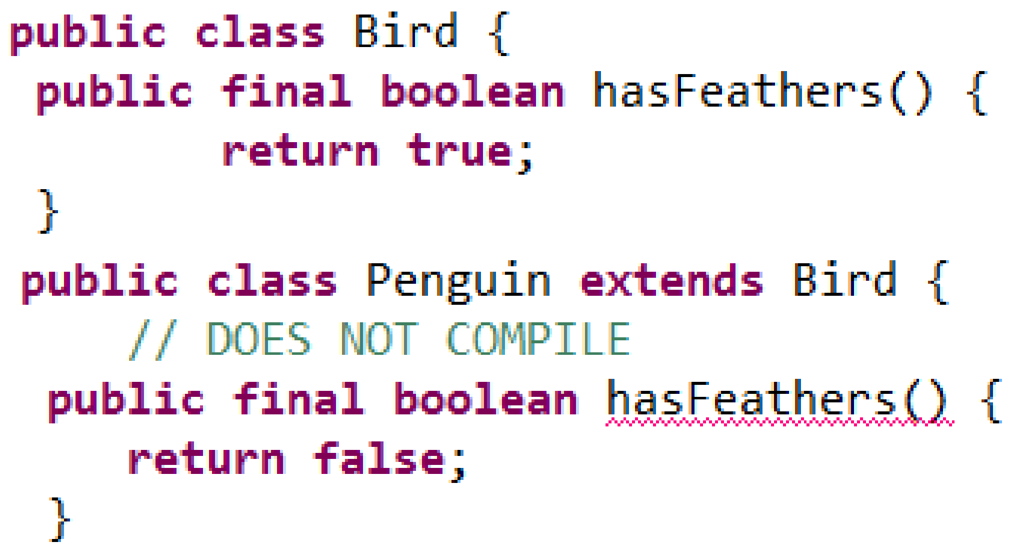
- Contrast this first example with the following example, which uses an overridden version of `isBiped()` instead of a hidden version.
- This code also compiles and runs without issue, but it outputs slightly different text:
  - `Marsupial walks on two legs: true`
  - `Kangaroo hops on two legs: true`
- In this example, the `isBiped()` method is overridden, not hidden, in the child class.
- Therefore, it is replaced at runtime in the parent class with the call to the child class's method.



```
public class Marsupial {
 public boolean isBiped() {
    return false;
 }
 public void getMarsupialDescription() {
    System.out.println("Marsupial "
                + "walks on two legs: "
                + isBiped());
 }

public class Kangaroo extends Marsupial {
 public boolean isBiped() {
        return true;
 }
 public void getKangarooDescription() {
    System.out.println("Kangaroo hops on "
            + "two legs: " + isBiped());
 }
 public static void main(String[] args) {
  Kangaroo joey = new Kangaroo();
  joey.getMarsupialDescription();
  joey.getKangarooDescription();
 }
```

1) OVERRIDING METHODS
1.6) CREATING FINAL METHODS

- `final` methods cannot be overridden.
- If we recall our discussion of modifiers, we can create a method with the `final` keyword.
- By doing so, though, we forbid a child class from overriding this method.
- This rule is in place both when we override a method and when we hide a method.
- In other words, we cannot hide a static method in a parent class if it is marked as final.
- In this example, the method `hasFeathers()` is marked as `final` in the parent class `Bird`, so the child class `Penguin` cannot override the parent method, resulting in a compiler error.
- Note that whether or not the child method used the `final` keyword is irrelevant - the code will not compile either way.



```
public class Bird {
 public final boolean hasFeathers() {
        return true;
 }
public class Penguin extends Bird {
    // DOES NOT COMPILE
 public final boolean hasFeathers() {
    return false;
 }
```

1) OVERRIDING METHODS
1.6) CREATING FINAL METHODS

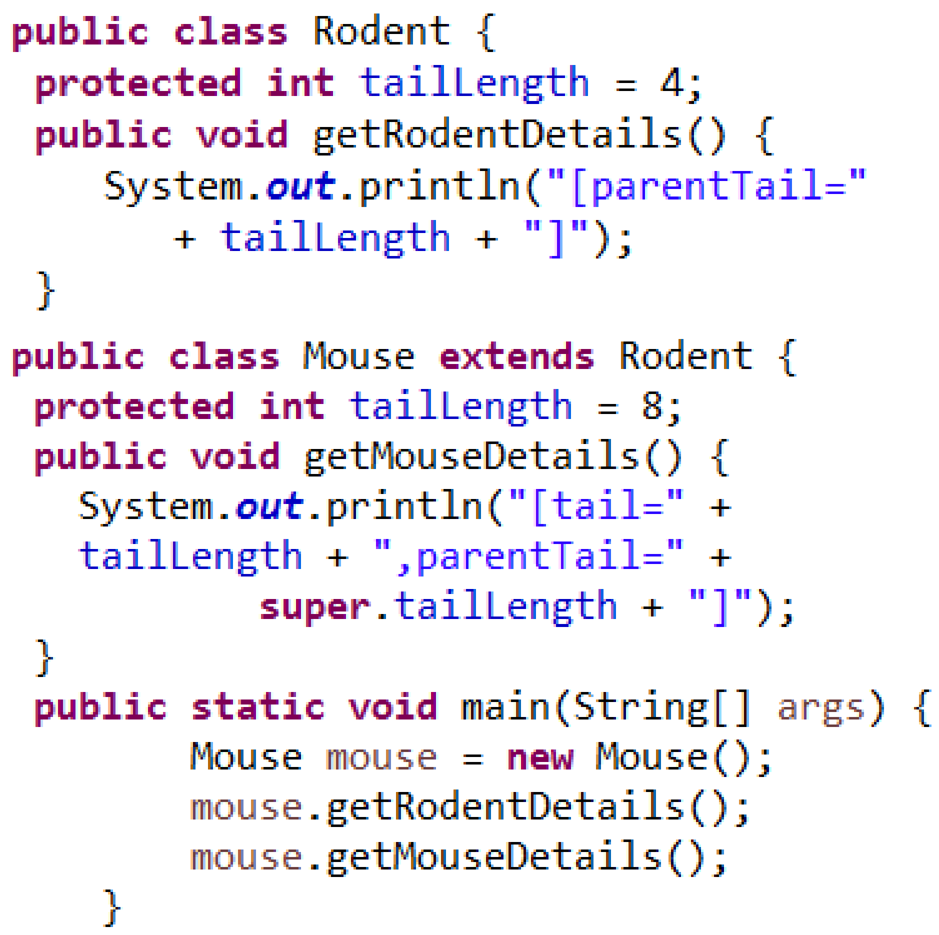
- Although marking methods as final prevents them from being overridden, it does have advantages in practice.
- For example, you'd mark a method as final when you're defining a parent class and want to guarantee certain behavior of a method in the parent class, regardless of which child is invoking the method.
- For example, in the previous example with Birds, the author of the parent class may want to ensure the method `hasFeathers()` always returns **`true`**, regardless of the child class instance on which it is invoked.
- The author is confident that there is no example of a Bird in which feathers are not present.
- The reason methods are not commonly marked as final in practice, though, is that it may be difficult for the author of a parent class method to consider all of the possible ways her child class may be used.
- For example, although all adult birds have feathers, a baby chick doesn't; therefore, if you have an instance of a Bird that is a chick, it would not have feathers.
- **In short, the final modifier is only used on methods when the author of the parent method wants to guarantee very precise behavior.**

1) OVERRIDING METHODS
1.7) HIDDING VARIABLES

- **When we hide a variable, we define a variable with the same name as a variable in a parent class.**
- This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference.
- As when hiding a static method, **we can’t override a variable; we can only hide it**.
- Also similar to hiding a static method, the rules for accessing the parent and child variables are quite similar.
- **If we’re referencing the variable from within the parent class, the variable defined in the parent class is used.**
- **Alternatively, if we’re referencing the variable from within a child class, the variable defined in the child class is used.**
- Likewise, we can reference the parent value of the variable with an explicit use of the `super` keyword.

1) OVERRIDING METHODS
1.7) HIDDING VARIABLES

- Consider the following example.
- This code compiles without issue and outputs the following when executed:
  - `[parentTail=4]`
  - `[tail=8,parentTail=4]`
- Notice that the instance of `Mouse` contains two copies of the `tailLength` variables: one defined in the parent and one defined in the child.
- These instances are kept separate from each other, allowing our instance of `Mouse` to reference both `tailLength` values independently.
- In the first method call, `getRodentDetails()`, the parent method outputs the parent value of the `tailLength` variable.
- In the second method call, `getMouseDetails()`, the child method outputs both the child and parent version of the `tailLength` variables, using the **`super`** keyword to access the parent variable's value.



```
public class Rodent {
 protected int tailLength = 4;
 public void getRodentDetails() {
    System.out.println("[parentTail="
       + tailLength + "]");
 }
public class Mouse extends Rodent {
 protected int tailLength = 8;
 public void getMouseDetails() {
   System.out.println("[tail=" +
   tailLength + ",parentTail=" +
           super.tailLength + "]");
 }
 public static void main(String[] args) {
       Mouse mouse = new Mouse();
       mouse.getRodentDetails();
       mouse.getMouseDetails();
    }
```

2) POLYMORPHISM

- Java supports polymorphism, **the property of an object to take on many different forms.**
- To put this more precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass.
- Furthermore, a cast is not required if the object is being reassigned to a super type or interface of the object.

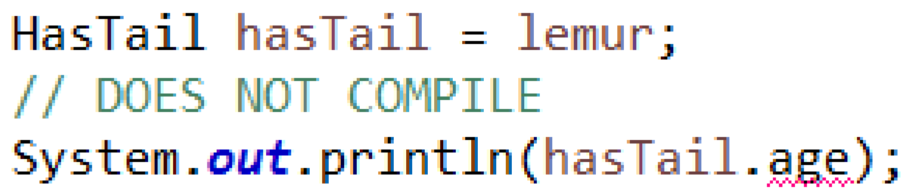
```java
public class Primate {
    public boolean hasHair() {
        return true;
    }
```

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

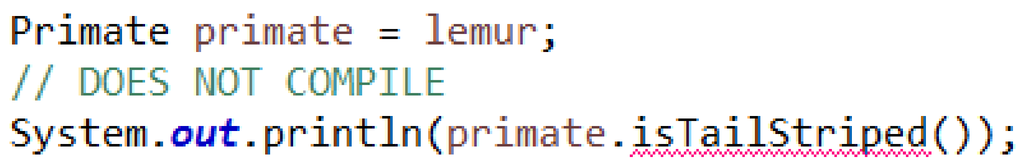
```java
public class Lemur extends Primate implements HasTail {
    public boolean isTailStriped() {
        return false;
    }
    public int age = 10;

    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());
```

2) POLYMORPHISM

- The most important thing to note about this example is that only one object, `Lemur`, is created and referenced.
- The ability of an instance of `Lemur` to be passed as an instance of an interface it implements, `HasTail`, as well as an instance of one of its superclasses, `Primate`, is the nature of polymorphism.
- Once the object has been assigned a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast.
- For example, the following snippets of code will not compile:



```
HasTail hasTail = lemur;
// DOES NOT COMPILE
System.out.println(hasTail.age);
```

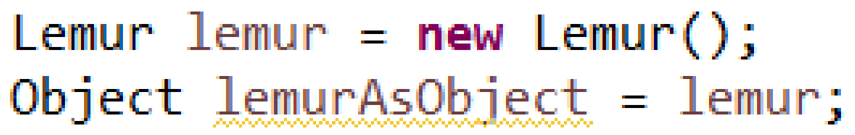


```
Primate primate = lemur;
// DOES NOT COMPILE
System.out.println(primate.isTailStriped());
```

- In this example, the reference `hasTail` has direct access only to methods defined in the `HasTail` interface; therefore, it doesn't know the variable `age` is part of the object.
- Likewise, the reference `primate` has access only to methods defined in the `Primate` class, and it doesn't have direct access to the `isTailStriped()` method.

2) POLYMORPHISM
2.1) OBJECT VS. REFERENCE

- In Java, all objects are accessed by reference, so as developers we never have direct access to the object itself.
- Conceptually, though, we should consider the object as the entity that exists in memory, allocated by the Java runtime environment.
- Regardless of the type of the reference we have for the object in memory, the object itself doesn't change.
- For example, since all objects inherit `java.lang.Object`, they can all be reassigned to `java.lang.Object`, as shown in the following example:

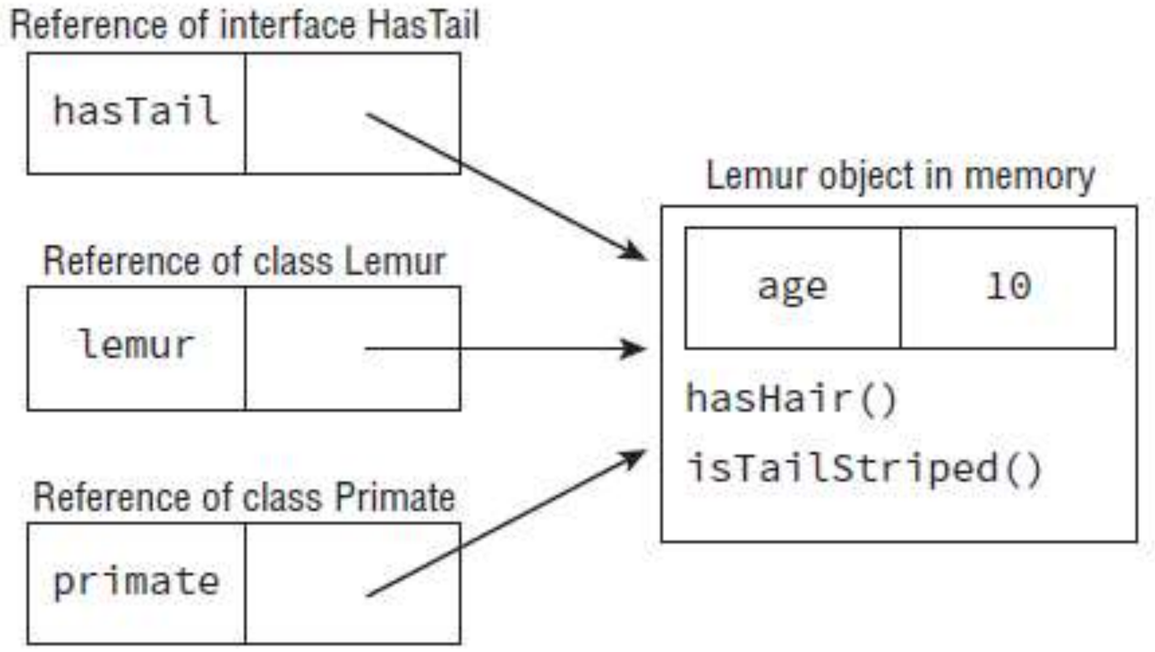


```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

- Even though the `Lemur` object has been assigned a reference with a different type, the object itself has not changed and still exists as a `Lemur` object in memory.
- What has changed, is our ability to access methods within the `Lemur` class with the `lemurAsObject` reference.
- Without an explicit cast back to `Lemur`, as we'll see, we no longer have access to the `Lemur` properties of the object.
- 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.

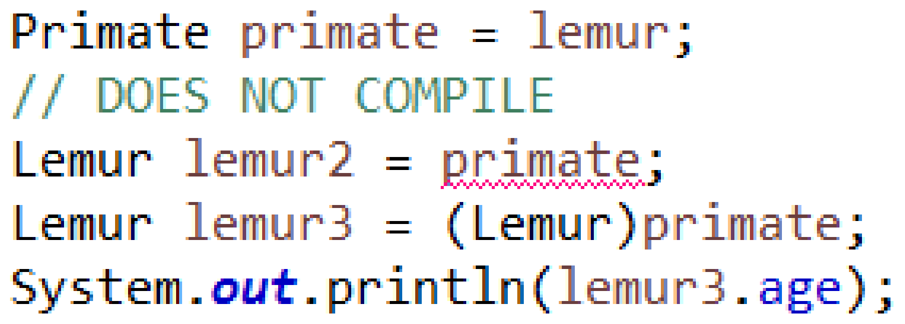
2) POLYMORPHISM
2.1) OBJECT VS. REFERENCE

- It therefore follows that successfully changing a reference of an object to a new reference type may give us access to new properties of the object, but those properties existed before the reference change occurred.
- As we can see in the figure, the same object exists in memory regardless of which reference is pointing to it.
- Depending on the type of the reference, we may only have access to certain methods.
- For example, the `hasTail` reference has access to the method `isTailStriped()` but doesn't have access to the variable `age` defined in the Lemur class.



2) POLYMORPHISM
2.2) CASTING OBJECTS

- We created a single instance of a `Lemur` object and accessed it via superclass and interface references.
- Once we changed the reference type, though, we lost access to more specific methods defined in the subclass that still exist within the object.
- We can reclaim those references by casting the object back to the specific subclass it came from:



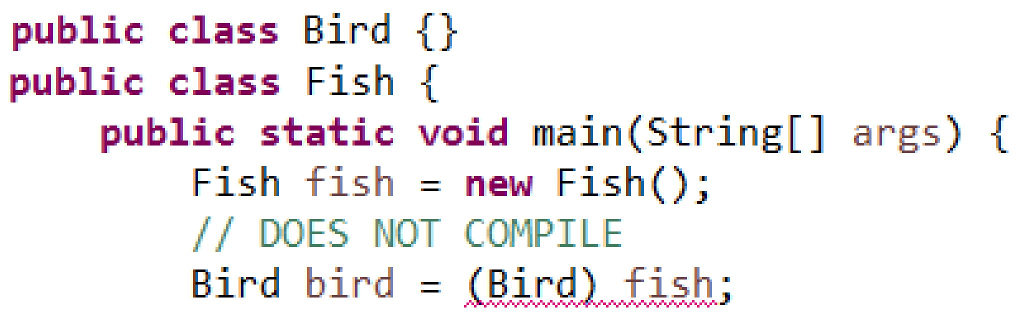
```
Primate primate = lemur;
// DOES NOT COMPILE
Lemur lemur2 = primate;
Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

- In this example, we first try to convert the `primate` reference back to a `lemur2` reference, without an explicit cast.
- The result is that the code will not compile.
- In the second example, though, we explicitly cast the object to a subclass of the object `Primate` and we gain access to all the methods available to the `Lemur` class.

2) POLYMORPHISM
2.2) CASTING OBJECTS

- Here are some basic rules to keep in mind when casting variables:
  1. Casting an object from a subclass to a superclass doesn't require an explicit cast.
  2. Casting an object from a superclass to a subclass requires an explicit cast.
  3. The compiler will not allow casts to unrelated types.
  4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

- For example, we were able to cast a `Primate` reference to a `Lemur` reference, because `Lemur` is a subclass of `Primate` and therefore related.
- Consider this example:

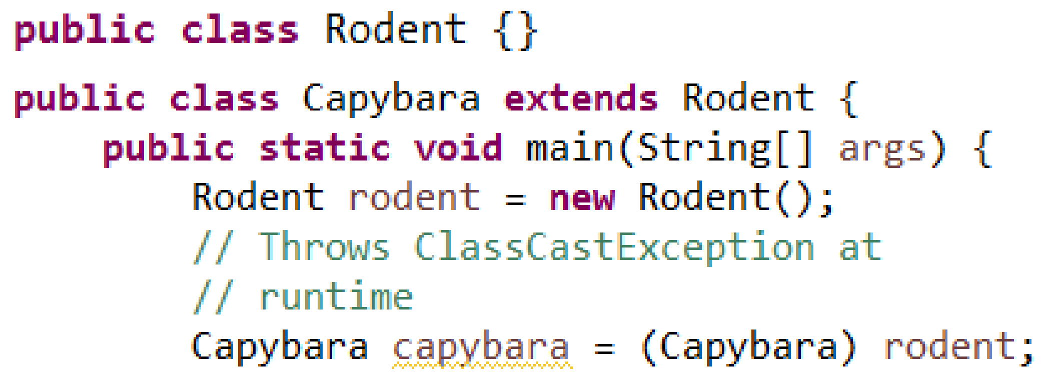


```
public class Bird {}
public class Fish {
    public static void main(String[] args) {
        Fish fish = new Fish();
        // DOES NOT COMPILE
        Bird bird = (Bird) fish;
```

- In this example, the classes Fish and Bird are not related through any class hierarchy that the compiler is aware of; therefore, the code will not compile.

2) POLYMORPHISM
2.2) CASTING OBJECTS

- Casting is not without its limitations.
- Even though two classes share a related hierarchy, that doesn’t mean an instance of one can automatically be cast to another.
- Here’s an example:

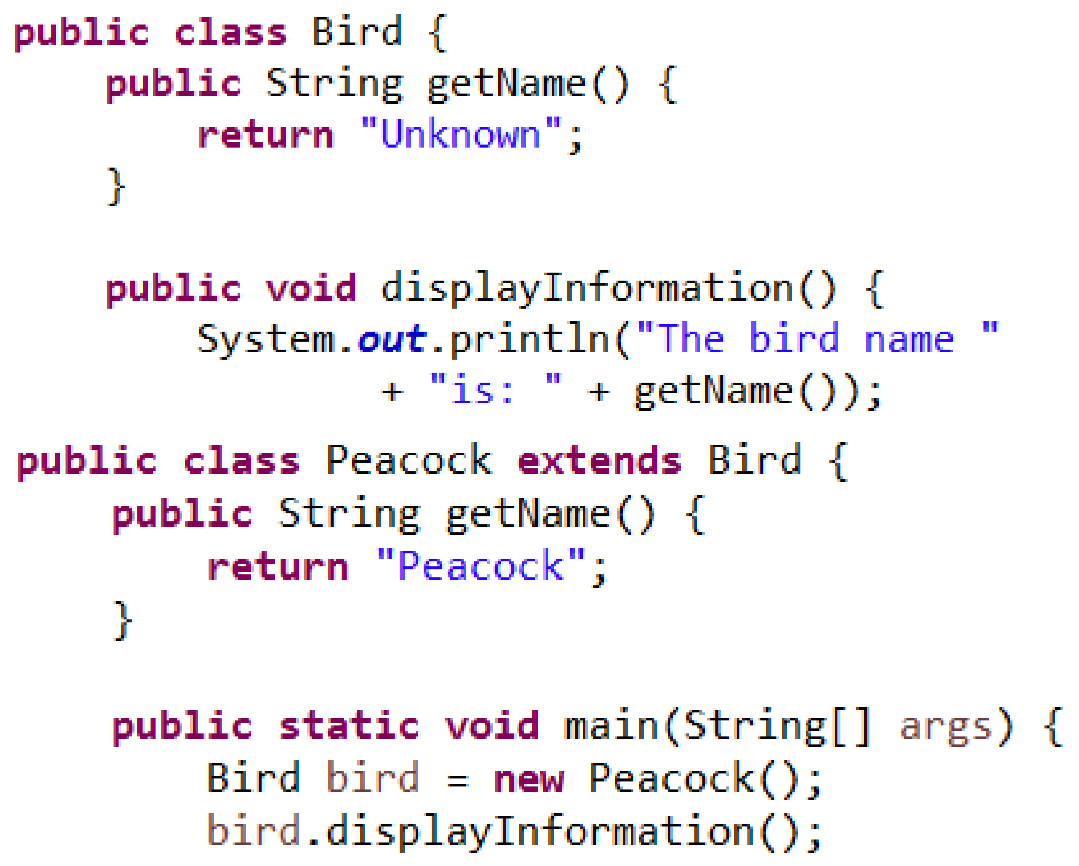


```
public class Rodent {}

public class Capybara extends Rodent {
    public static void main(String[] args) {
        Rodent rodent = new Rodent();
        // Throws ClassCastException at
        // runtime
        Capybara capybara = (Capybara) rodent;
```

- This code creates an instance of `Rodent` and then tries to cast it to a subclass of `Rodent`, `Capybara`.
- Although this code will compile without issue, it will throw a `ClassCastException` at runtime since the object being referenced is not an instance of the `Capybara` class.
- The thing to keep in mind in this example is the object that was created is not related to the Capybara class in any way.

2) POLYMORPHISM
2.3) VIRTUAL METHODS

- The most important feature of polymorphism, and one of the primary reasons we have class structure at all, is to support virtual methods.
- A virtual method is a method in which the specific implementation is not determined until runtime.
- In fact, all non-final, non-static, and non-private Java methods are considered virtual methods, since any of them can be overridden at runtime.
- What makes a virtual method special in Java is that if we call a method on an object that overrides a method, we get the overridden method, even if the call to the method is on a parent reference or within the parent class.
- We'll illustrate this principle with the following example:
- This code compiles and executes without issue and outputs the following:
  - `The bird name is: Peacock`



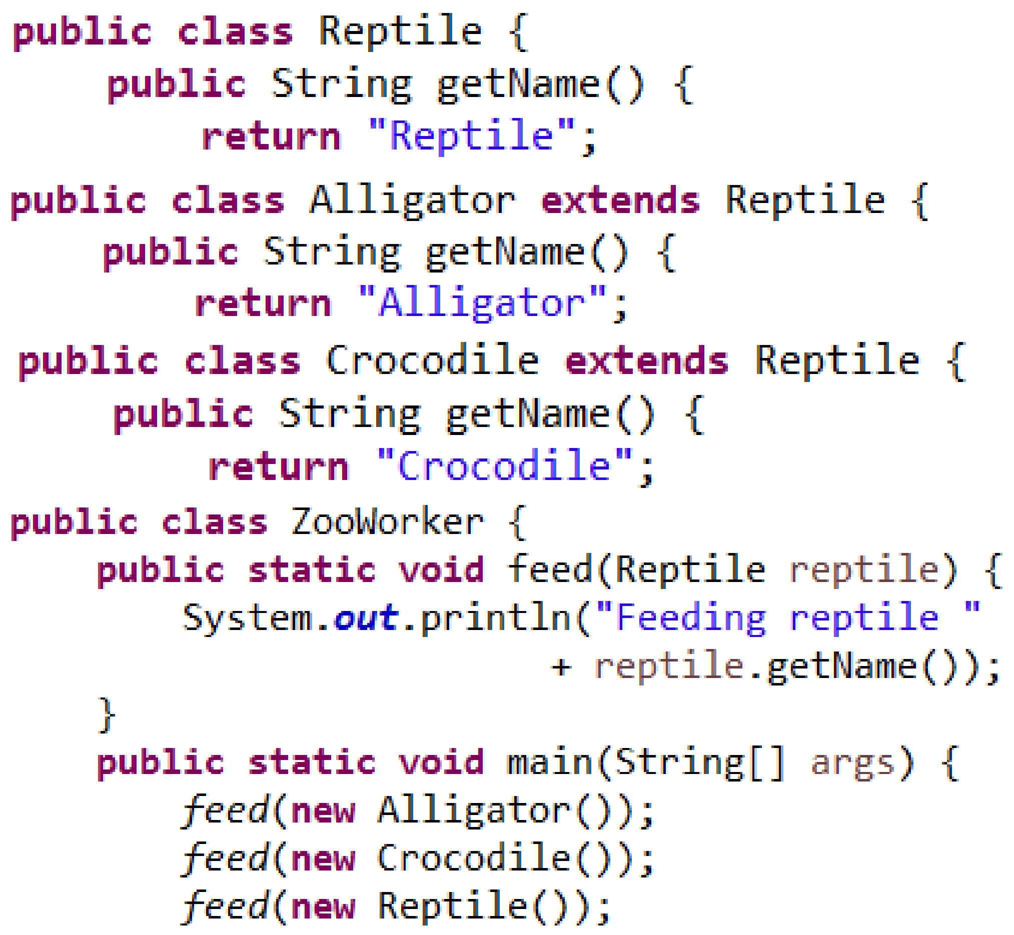
```java
public class Bird {
    public String getName() {
        return "Unknown";
    }

    public void displayInformation() {
        System.out.println("The bird name "
                + "is: " + getName());
public class Peacock extends Bird {
    public String getName() {
        return "Peacock";
    }

    public static void main(String[] args) {
        Bird bird = new Peacock();
        bird.displayInformation();
```

2) POLYMORPHISM
2.3) VIRTUAL METHODS

- As we saw in similar examples in the section “Overriding a Method,” the method `getName()` is overridden in the child class `Peacock`.
- More importantly, though, the value of the `getName()` method at runtime in the `displayInformation()` method is replaced with the value of the implementation in the subclass `Peacock`.
- In other words, even though the parent class `Bird` defines its own version of `getName()` and doesn’t know anything about the `Peacock` class during compile-time, at runtime the instance uses the overridden version of the method, as defined on the instance of the object.
- We emphasize this point by using a reference to the `Bird` class in the `main()` method, although the result would have been the same if a reference to `Peacock` was used.
- We now know the true purpose of overriding a method and how it relates to polymorphism.
- **The nature of the polymorphism is that an object can take on many different forms.**
- By combining your understanding of polymorphism with method overriding, we see that objects may be interpreted in vastly different ways at runtime, especially in methods defined in the superclasses of the objects.

2) POLYMORPHISM
2.4) POLYMORPHIC PARAMETERS

- One of the most useful applications of polymorphism is the ability to pass instances of a subclass or interface to a method.
- For example, we can define a method that takes an instance of an interface as a parameter.
- In this manner, any class that implements the interface can be passed to the method.
- Since we're casting from a subtype to a supertype, an explicit cast is not required.
- This property is referred to as polymorphic parameters of a method, and we demonstrate it in the following example:
- This code compiles and executes without issue, yielding the following output:
  - `Feeding: Alligator`
  - `Feeding: Crocodile`
  - `Feeding: Reptile`



```java
public class Reptile {
    public String getName() {
        return "Reptile";
public class Alligator extends Reptile {
    public String getName() {
        return "Alligator";
public class Crocodile extends Reptile {
    public String getName() {
        return "Crocodile";
public class ZooWorker {
    public static void feed(Reptile reptile) {
        System.out.println("Feeding reptile "
                + reptile.getName());
    }
    public static void main(String[] args) {
        feed(new Alligator());
        feed(new Crocodile());
        feed(new Reptile());
```

2) POLYMORPHISM
2.4) POLYMORPHIC PARAMETERS

- Let’s focus on the **`feed(Reptile reptile)`** method in this example.
- As we can see, that method was able to handle instances of `Alligator` and `Crocodile` without issue, because both are subclasses of the `Reptile` class.
- It was also able to accept a matching type `Reptile` class.
- If we had tried to pass an unrelated class, such as the previously defined `Rodent` or `Capybara` classes, or a superclass such as `java.lang.Object`, to the `feed()` method, the code would not have compiled.