Polymorphism in Java

Polymorphism

What is Polymorphism?

- the ability of an object to take on many forms - Tutorialspoint
- "The provision of a single interface to entities of different types" (Wikipedia
- A popular topic in technical interviews

Inheritance is important to polymorphism

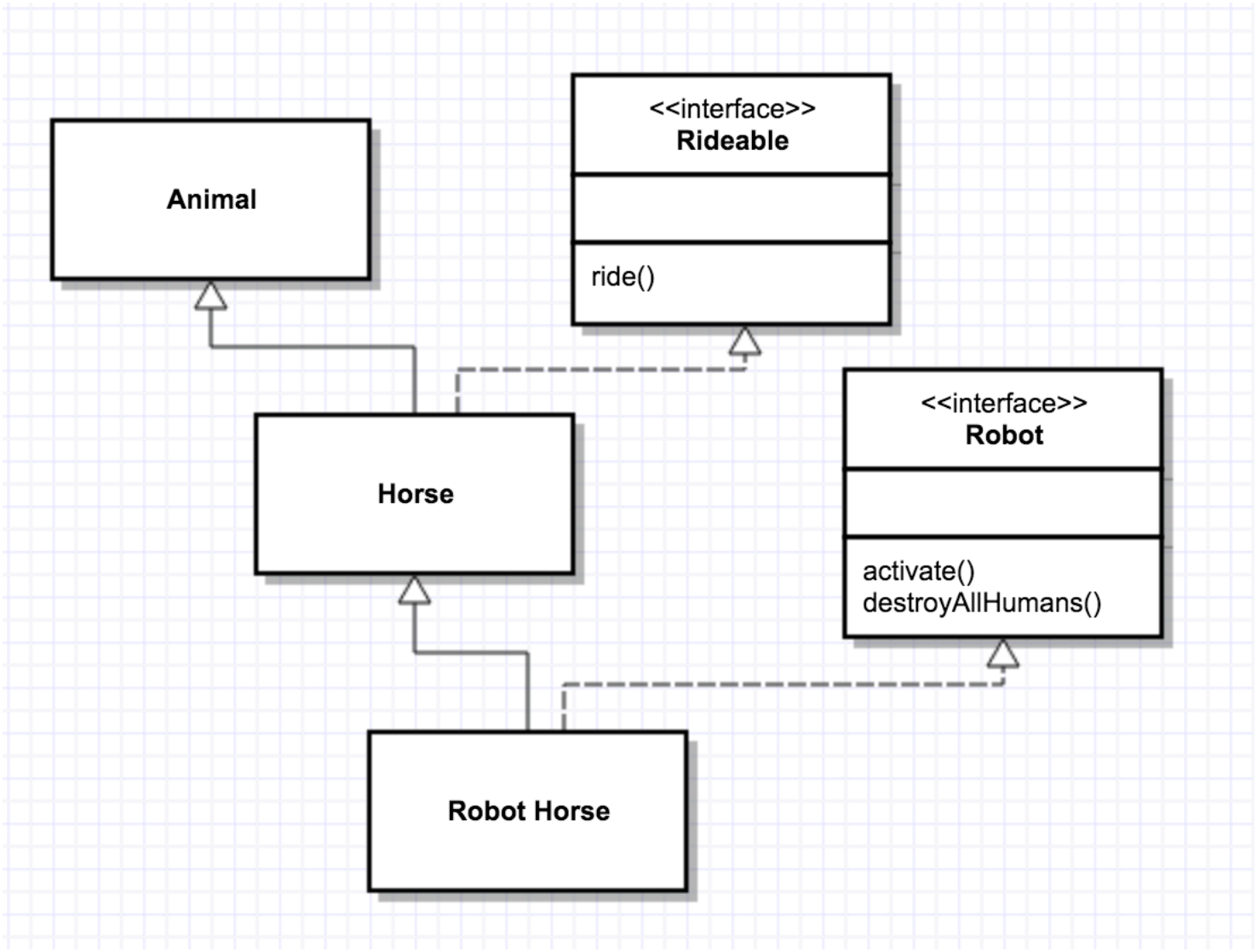
...and cash-strapped relatives and European royalty

Keyword:

- `extends` : Creates a subclass of the specified class
- `implements` : Implements the methods of the specified interface(s)

Polymorphic code examples

Example 1: Livin' on a Mare



Example 1 continued

```
class Animal {}
interface Rideable{ int getRange(); }
class Horse extends Animal implements Rideable{
  public int getRange(){ return 120; }
}
interface Robot {
  default void activate(){ System.out.println("Bzzt!"); }
  void destroyAllHumans();
}
class RobotHorse extends Horse implements Robot{
  public void destroyAllHumans(){
    System.out.println("Humans Wanted Dead or Alive");
  }
}
```

Example 1 continued

```
public class App{
  public void ride(Rideable mount){
    System.out.println("I can ride " + mount.getRange() + " miles today.");
  }

  public static void main(String[] args){
    RobotHorse cavallo = new RobotHorse();
    ride(cavallo);
    cavallo.destroyAllHumans();
  }

}
```

Polymorphic Program design

- Most methods rely on provided interfaces, rather than underlying implementations
- Object fields are accessed through getters/setters

Things that don't behave polimorphically

- field accesses (eg: `Parent p = new Child(); p.x;`)
- Static methods

```
//: polymorphism/FieldAccess.java
// Direct field access is determined at compile time.
class Super {
  public int field = 0;
  public int getField() { return field; }
}
class Sub extends Super {
  public int field = 1;
  public int getField() { return field; }
  public int getSuperField() { return super.field; }
}
```

```
public class FieldAccess {
  public static void main(String[] args) {
    Super sup = new Sub(); // Upcast
    System.out.println("sup.field = " + sup.field +
      ", sup.getField() = " + sup.getField());
    Sub sub = new Sub();
    System.out.println("sub.field = " +
      sub.field + ", sub.getField() = " +
      sub.getField() +
      ", sub.getSuperField() = " +
      sub.getSuperField());
  }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

Benefits of Polymorphism

Instances of a class can be treated as any superclass, interface, or superinterface in that class's type hierarchy.

Implementation Hiding

More modular code

Separate interface from implementation

Downcasting

- Downcast back to subclass with `(Subclass)`
- Downcasting is checked at runtime
- Incorrect downcasting causes `ClassCastException`

Polymorphic Refactoring Examples

Person class example:

```
public class Person{
  private String language;
  public Person(){ this.language = "English"; }
  public Person(String lang){
      switch(lang){
        case "German":
        case "Spanish":
        case "French":
            this.language = lang;
            break;
        case "English":
        default:
            this.language = "English"
      }
  }
  public String getLanguage(){ return language;}
  public String sayHello(){
    switch(language){
      case "German": return "Guten tag!";
      case "Spanish": return "¡Buenos días!";
      case "French": return "Bonjour!";
      case "English":
      default: return "Hello!"
  }
}
```

Person class refactored:

```
class Person{
  protected String language = "English";
  public String getLanguage(){ return language; }
  public String sayHello(){ return "Hello!"; }
}

class Germanophone extends Person{
  public Germanophone(){ language = "German"; }
  public String sayHello(){ return "Guten Tag!"; }
}
// repeat for speakers of Spanish, French, English

public class App{
  public static void main(String[] args){
    Person germanSpeaker = new Germanophone();
    Person frenchSpeaker = new Francophone();
    System.out.println("German speaker says: " + germanspeaker.sayHello());
    System.out.println("French speaker says: " + frenchSpeaker.sayHello());
  }
}
```

Person class refactored as abstract class:

```
abstract class Person{
  protected String language;
  public String getLanguage(){ return language;}
  public abstract String sayHello();
}

class Germanophone extends Person{
  public Germanophone(){ language = "German"; }
  public String sayHello(){ return "Guten Tag!"; }
}
// repeat for speakers of Spanish, French, English
class Francophone extends Person{
  public Francophone(){ language = "French"; }
  public String sayHello(){ return "Bonjour!"; }
}

public class App{
  public static void main(String[] args){
    Person germanSpeaker = new Germanophone();
    Person frenchSpeaker = new Francophone();
    System.out.println("German speaker says: " + germanSpeaker.sayHello());
    System.out.println("French speaker says: " + frenchSpeaker.sayHello());
  }
}
```