INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY **DELHI**

Advanced Programming

Object Oriented Programming with Java

Srikanta Bedathur

CSE 201

Monsoon 2012

ArrayList <T>

- We used this earlier as a flexible replacement for an Array
- An example of a generic class
  - Parameterized by the type passed as argument
- Compiler can type-check and enforce strong type safety
  - No need for casting return values
  - Prior to Java 5.0, ArrayList just had Object references
    - Casting
    - Objects of mixed types

```
ArrayList mixtureList = new ArrayList ();
mixtureList.add (new String("Hello World"));
mixtureList.add (new File ("hello.txt"));
```


Implementing a Generic class

- Do we need to implement any generic class ever?
  - JDK has a large repertoire of generics
  - Almost all collection classes are generic
- It is a challenging task to design good generics
  - Users of your class can (and will) use any class
  - You can not assume any one specific class usage
- Let's try to implement some simple generic classes
  - **Kennel <T>**
  - KeyValue <K, V>

Kennel <T>

```
abstract class Mammal {
  abstract void eat (Food f);
}
class Dog extends Mammal {
  void eat (Food f);
}
class Cat extends Mammal {
  void eat (Food f);
}
class Kennel <T> {
  private List<T> animals;
  void printNames () {
    for (T animal : animals) {
      System.out.println(animal.toString());
    }
  }
  void add (T in) {...}
}
```

- Almost correct ...
- How do we know that animal.eat(f) is valid?

Kennel<T> with Type Bounding

```
abstract class Mammal {
  abstract void eat (Food f);
}
class Dog extends Mammal {
  void eat (Food f);
}
class Cat extends Mammal {
  void eat (Food f);
}
class Kennel <T> {
  private List<T> animals;
  void printNames () {
    for (T animal : animals) {
      System.out.println(animal.toString());
    }
  }
  void add (T in) {...}
}
```

- We use *extends* even when we require T to implement a specific interface
  - i.e., if Mammal was an interface
- There can be multiple bounds
  - E.g. **class Kennel <T extends Mammal & PetAnimal>**
  - **Naturally only one of them can be a class**

Type Erasure

- Internally, JVM **does not** have objects of generic type
  - Compiler adds additional code to ensure type-safety
- E.g. in the previous code,
  `(kennel1.getClassName () == kennel2.getClassName())`
  returns true!
- The *Raw type* of the generic erases its type
  - Replaces T with the best possible choice
    - If unbounded - replaces with Object
    - Otherwise, the bounding class
  - What if you have multiple type bounds
    - The first bounding type

Some Restrictions on Generics

- Type parameters cannot be instantiated within
  - You can use Class.newInstance() - but quite tricky
  - T.class - is invalid, you need an instance!
- Type parameters cannot be replaced by primitive types
  - Autoboxing helps here
- Generics cannot extend Throwable
  - But you can throw a type variable from a generic
  - **public void doWork (int x) throws T**
- Arrays of generics is not legal
  - As a result of type erasure, JVM cannot guarantee type safety

Type Erasure and Type bounding

```
public class Interval <T extends Comparable & Serializable> implements
Serializable  {
   public Interval (T first, T second) {
      ...
   }
   private T lower;
   private T upper;
}
```

```
public class Interval implements Serializable  {
   public Interval (Comparable first, Comparable second) {
      ...
   }
   private Comparable lower;
   private Comparable upper;
}
```

- Type erasure retains only the first type bound in the parameter list
  - Compiler still inserts the required casts whenever needed
    - You can still assign to a Serializable reference
- Casting costs a bit - tiny bit, but still...
  - So, keep the tagging interfaces at the end to minimize casting

Inner Generics

- Inner classes can be generics
  - Within another generic class or *normal* class

```
public class CircularLinkedList <T> {
   private static class Node <U> {
      U me;
      Node <U> next;
      Node () { me = null; next = null;}
      Node (U m, Node <U> nxt) { me = m; next = nxt;}
      ...
   }
   private Node <T> connector = new Node <T> ();
   ...
}
```

- Also can access the type parameter of the outer

```
public class CircularLinkedList <T> {
   private static class Node {
      T me;
      Node next;
      Node () { me = null; next = null;}
      Node (T m, Node nxt) { me = m; next = nxt;}...
   }
}
```

Generic Interfaces

```
public interface Generator <T> {
  T create ();
}
public StudentFactory implements Generator <Student>,
Iterable <Student> {
  public Student create()
  {Student.class.newInstance();...}
  class StudentIterator implements Iterator <Student>
  {...}
  public Iterator<Student> iterator() {
    return new StudentIterator ();
  }
  ...
}
```

- Generics work with Interfaces just like in classes
  - Notice the pattern of usage in the example above

Generic Methods too...!

```
class HolderOfGenericMethods {
   public <T> void foo (T x) {
      System.out.println ("I have the " + x.getClass().getName());
   }
   public static <U> U bar (U[] in) {
      return in[0];
   }
}

public class Test {
    public static void main (String[] args) {
        HolderOfGenericMethods hogm = new HolderOfGenericMethods ();
        hogm.<Person> foo (new Person());

        hogm.foo (new Person());
        Integer[] arr = {10, 12, 13};
        Integer first = HolderOfGenericMethods.<Integer>bar (arr);
    }
}
```


Inheritance with Generics

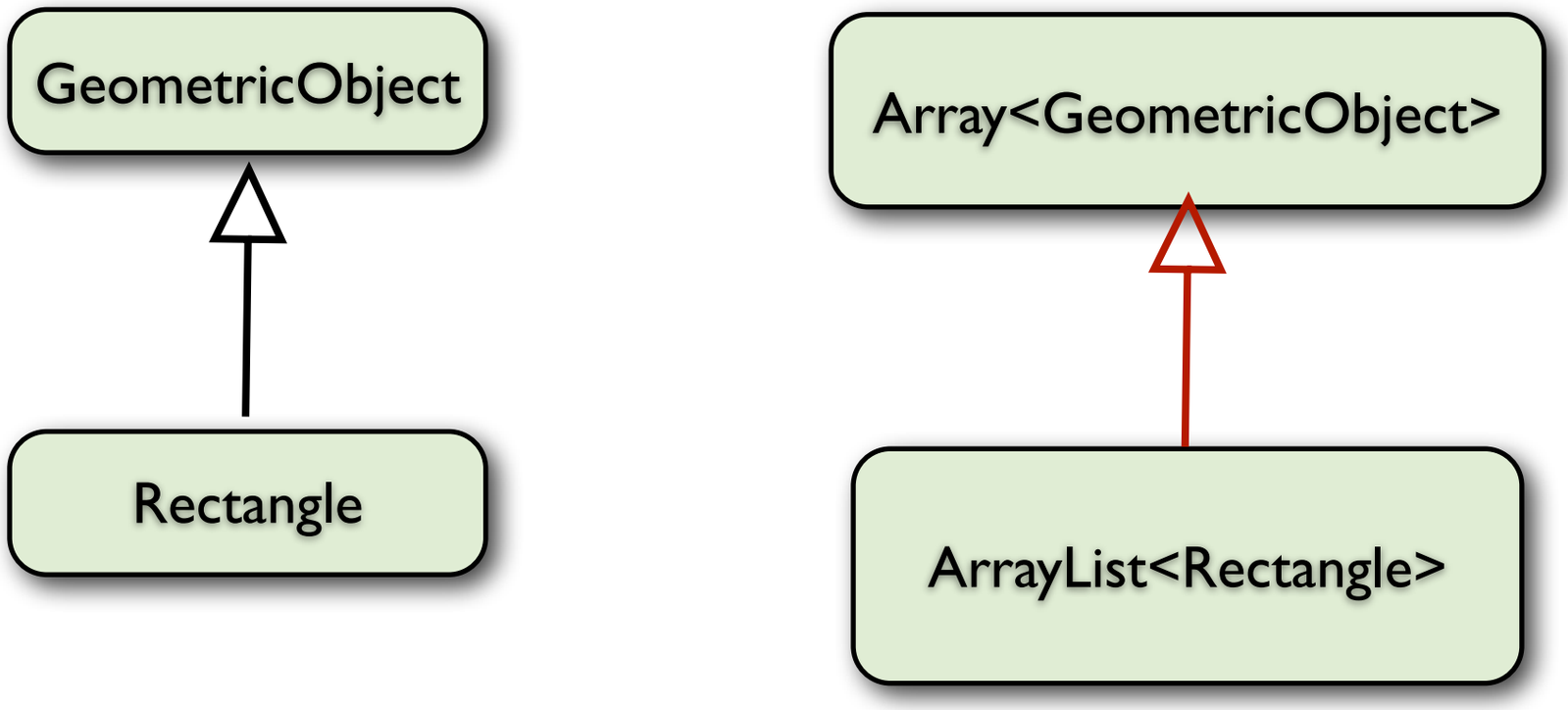
- Consider Pair <S, T> - Can be extended to Triple

```
public class Pair <S, T> {
  public final S first;
  public final T second;
  public Pair (S f, T s) {
    first = f;
    second = s;
  }
  public String toString () {...}
}
```

```
public class Triple <S, T, U> extends Pair <S, T> {
  public final U third;
  public Triple (S f, T s, U t) {
    super (f, s);
    third = t;
  }
  public String toString () {...}
}
```

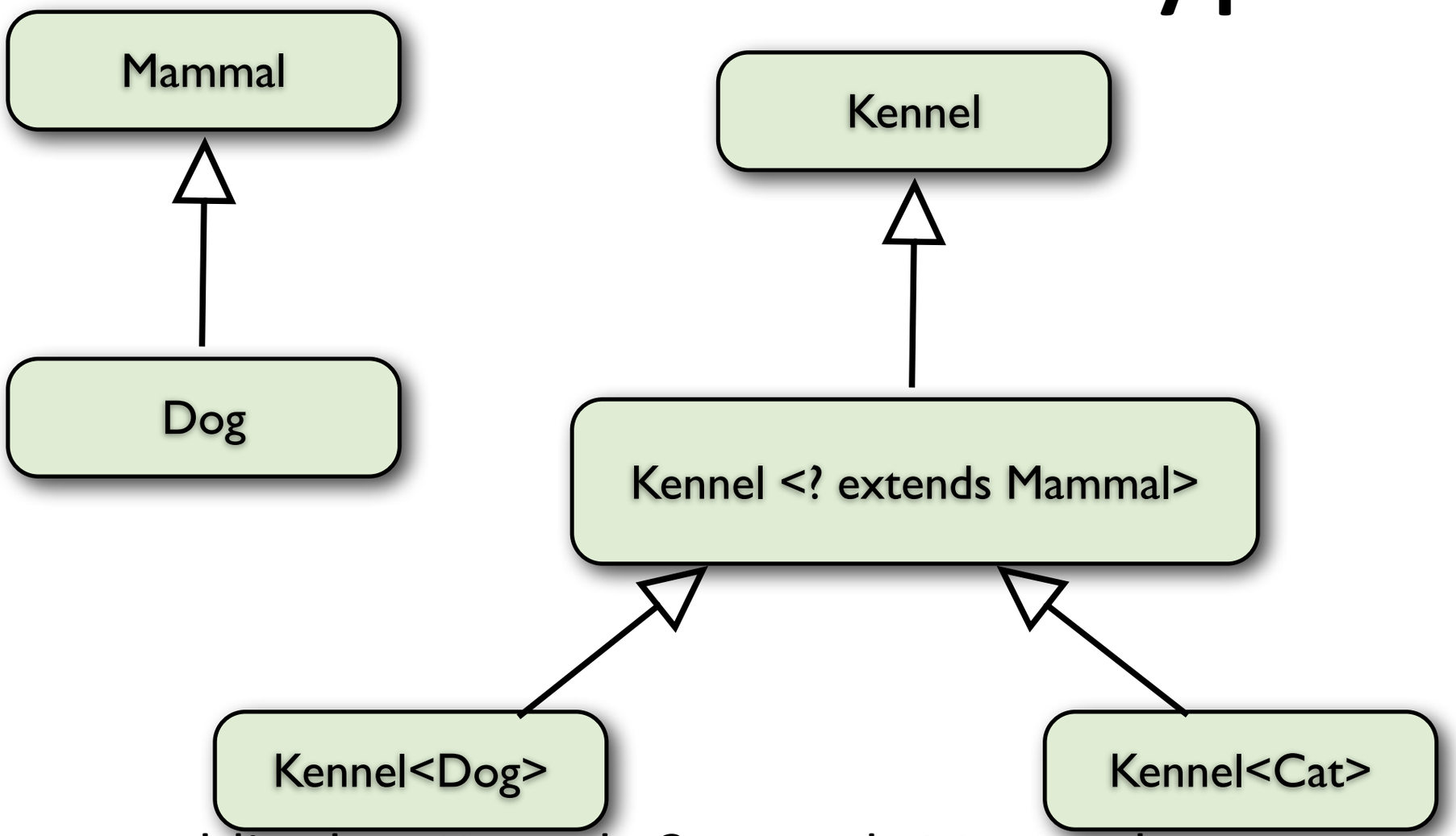
- Triple to Quadruple, Quituple, etc.

Inheritance Rules for Generics



- Generics do not automatically inherit the inheritance of parameter types
- But a Generic class can be inherited!
  - `CushyKennel <Mammal> extends Kennel <Mammal>`
- Can we get the advantage of inheritance of parameter types?

Wildcard Types



- public class Kennel <? extends Mammal>
  - This enables subtyping/inheritance
- **Just using "? extends" enables safe accessor, but locks-out mutators**
- Similarly, you can use "super" in place of "extends"
  - enables safe mutator, but locks-out accessor!

Wildcard Types - Workings

- Let's look at Kennel<? Extends Mammal>

```
class Kennel <T> {
   private List<T> animals;
   T getFirst () {
      return animals.get (0);
   }
   void printNames () {
      for (T animal : animals) {
         System.out.println(animal.toString());
      }
   }
   void add (T in) {animals.add (in);}
}
```

```
class Kennel <? extends Mammal> {
   private List<T> animals;
   ? extends Mammal getFirst () {
      return animals.get (0);
   }
   void printNames () {
      for (? extends Mammal animal : animals) {
         System.out.println(animal.toString());
      }
   }
   void add (? extends Mammal in) {animals.add (in);}
}
```

What about Kennel <?>

- It looks similar to Kennel without any type parameter
  - After all ? can match any type
- But,
  - This makes the generic immutable through that reference
    - getFirst () can be assigned an Object
    - add (?) can never be called!

What about Kennel <?>

- It looks similar to Kennel without any type parameter
  - After all ? can match any type
- But,
  - This makes the generic immutable through that reference
    - getFirst () can be assigned a
    - add (?) can never be called