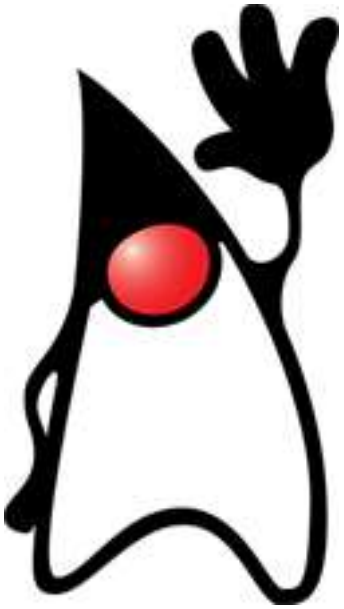


Generic Programming

Programmazione Concorrente e Distr.

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2015 – 2016

rcardin@math.unipd.it

SUMMARY

- Introduction
- Generic classes and methods
- Type variables bounds
- Type erasure
- Restrictions and limitations
- Generics and inheritance
- Wildcard types

INTRODUCTION

- Generic programming means writing code that can be reused for object of many different types
  - Added to the JDK from version 5.0
    - It tooks 5 years to be formalized (from 1999)
    - First added to support strongly typed collections (as in C++)
  - No generic code is obscure and not typesafe

```
// Before generic classes
ArrayList files = new ArrayList();
// You do not know which kind of object are store inside the list
String filename = (String) files.get(0);
```

    - Before generics there was no typesafe checking
    - Errors will be find only at *runtime*
    - Violation of FAIL FAST principle

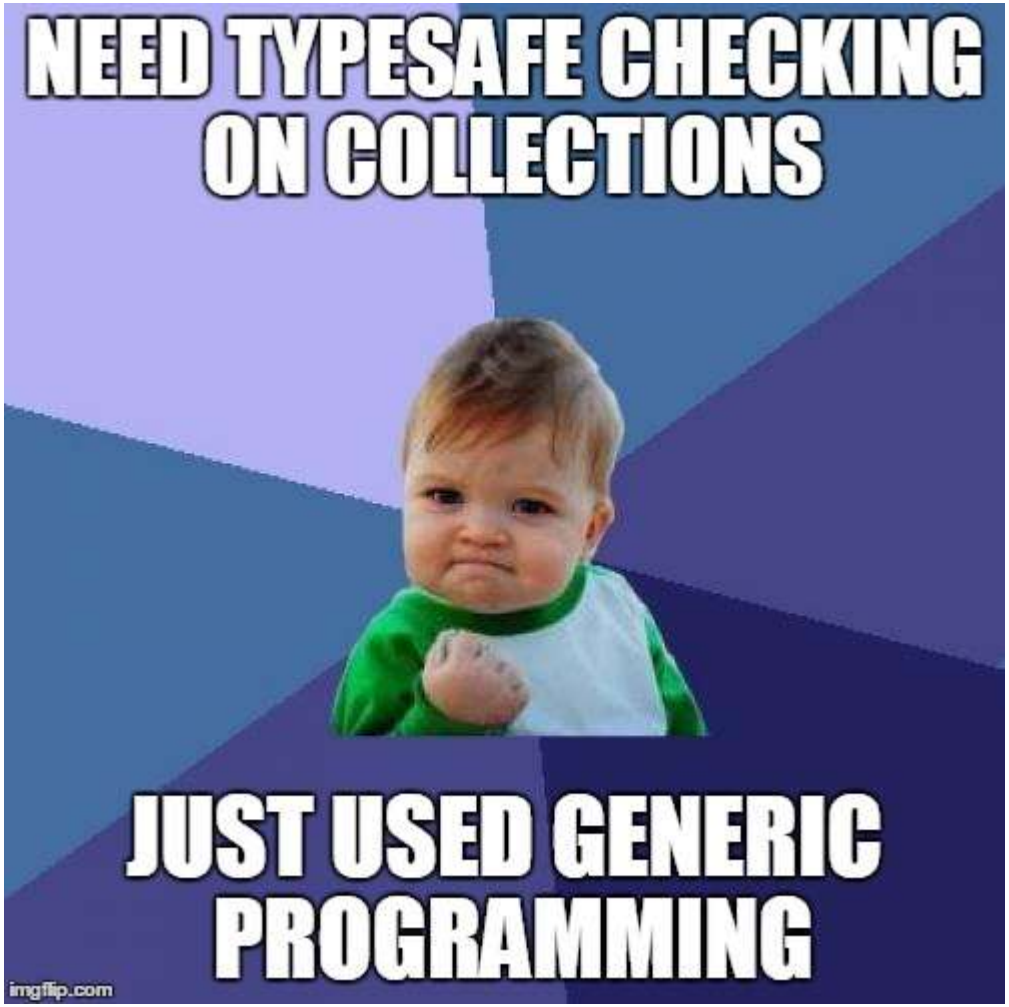
Introduction

- Generics solution: type parameters
  - The compiler can perform typesafe checks

```
// After generic classes
ArrayList<String> files = new ArrayList<>();
// You do not need the cast operation anymore
String filename = files.get(0);
```

  - Generics make the code safer and easier to read
- But how will you use generic programming?
  - Basic level: just use generic classes
  - Intermediate level: when you encounter your first enigmatic error using generic classes
  - Advanced level: implement your own generic classes

INTRODUCTION



GENERIC CLASSES

- A class with one or more type variable
  - Type variable are introduced after class name, enclosed by angle brackets < >
    - Type variables are used throughout the class definition

```
class Pair<T, U> {
    private T first;
    private U second;
    // ...
}
```

    - It is common to use upper case letters for type variables
  - A type variables is instantiated by substituting types
    - Generic classes act as a factory for ordinary classes

```
Pair<Integer, String> pair = new Pair<>();
Integer first = pair.getFirst();
String second = pair.getSecond();
```

GENERIC METHODS

- Also single methods can be defined as generics
  - Type variable definition stands behind method's return type

```
public static <T> T getMiddle(T... a) { /* ... */ }
String middle = getMiddle("Riccardo", "Cardin", "Professor");
```

    - Can be placed inside both generic an ordinary classes
  - You can omit the type variable instantiation
    - But sometimes the compiler gets it wrong...

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

    - Which type is inferred by the compiler? `Number`!!!
  - In C++, type parameters are after method name
    - This can lead to ambiguites

TYPE VARIABLES BOUNDS

- Rescriction of a type variable to a class that is a subtype of an another type

```
class ArrayAlg {
    public static <T extends Comparable> T min(T[] a) {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            // We know for sure that compareTo is available
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}
```

- Use `extends` keyword both for classes and interfaces
- It is possibile to use multiple bounds

```
T extends Comparable & Serializable
```

  - At most one bound can be a class

TYPE ERASURE

- The JVM does not have objects of generic types
  - Type variables are erased and replaced with bounding variables or `Object` type if it ain't any
    - For each generic type a raw type is provided automatically

```
// Remember generic class Pair<T, U> ?
class Pair {
    private Object first;
    private Object second;
    // ...
}
```

  - Casts are inserted if necessary to preserve type safety
  - Bridge methods are generated to preserve polymorphism
  - No new class are created for parametrized types

TYPE ERASURE

- Translating generic expression
  - Compiler inserts casts when the return the return type has been erased

```
Pair<Employee, Salary> buddies = /* ... */;
Employee buddy = buddies.getFirst();
// After type erasure will become
Pair buddies = /* ... */;
Employee buddy = (Employee) buddies.getFirst(); // Returns an Object
```

- Translating generic methods
  - A generic method is not a family of methods

```
public static <T extends Comparable> T min(T[] a)
// becomes after type erasure
public static Comparable min(Comparable[] a)
```

  - This fact has a series of implications...

TYPE ERASURE

- Translating generic methods
  - Type erasure interferes with polymorphism, then the compiler generates synthetic bridge methods

```
public class Node<T> {
    public T data;
    public Node(T data) { this.data = data; }
    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }
    @Override
    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

TYPE ERASURE

- Translating generic methods
  - After type erasure setData method has wrong type and cannot override parent method

```
public class Node {
    public Object data;
    public Node(Object data) { this.data = data; }
    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
public class MyNode extends Node {
    public MyNode(Integer data) { super(data); }
    // Not polymorphic anymore :O
    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

TYPE ERASURE

- Translating generic methods
  - Compiler insert a bridge method that overrides the parent method, resolving subtyping
    - Delegation design pattern

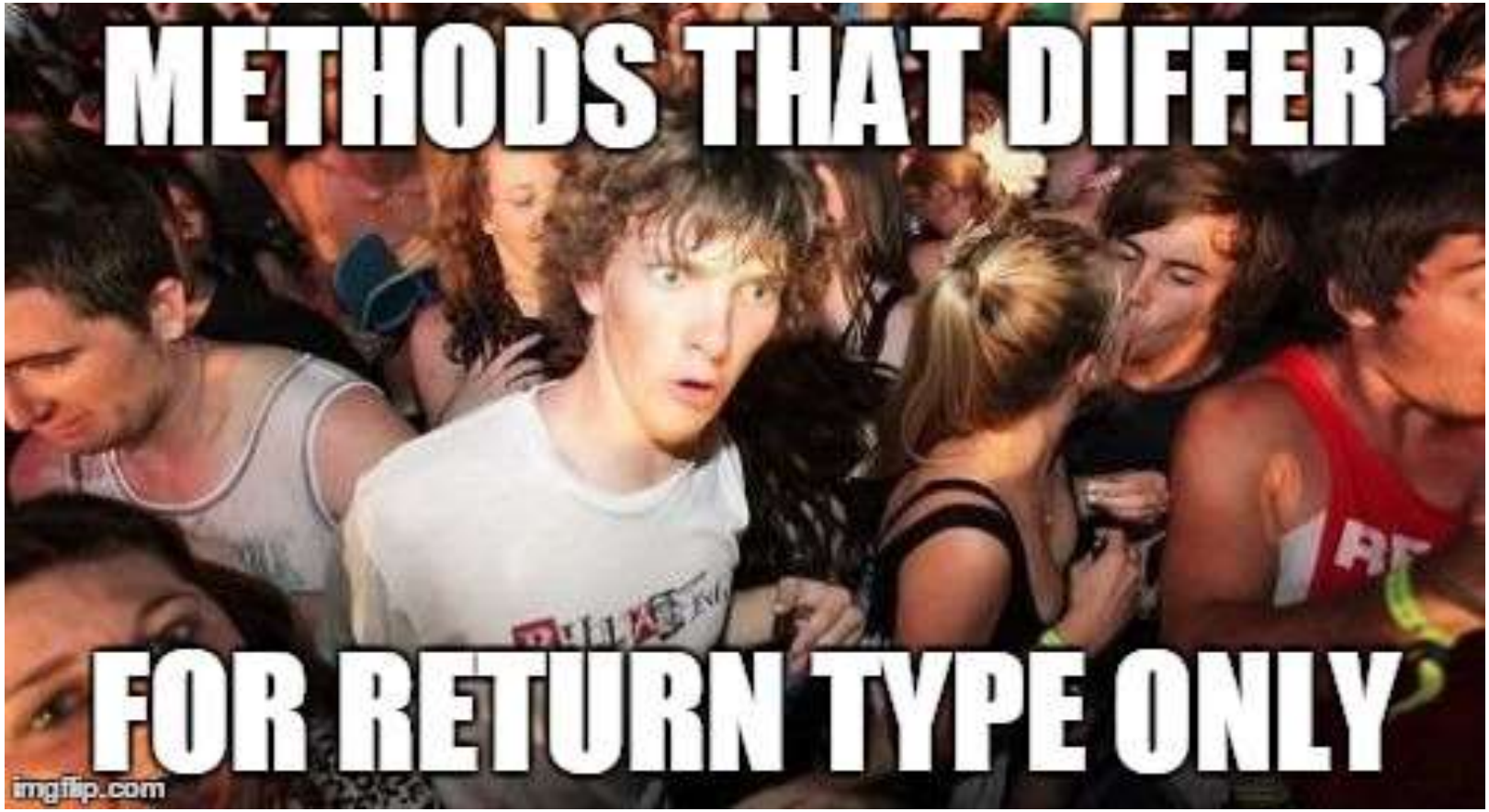
```
class MyNode extends Node {
    // Bridge method generated by the compiler
    public void setData(Object data) {
        setData((Integer) data);
    }
    // ...
```

  - There are cases that are stanger
    - The type parameters appears only in method return type

```
class DateInterval extends Pair<Date> {
    public Date getSecond() {
        return (Date) super.getSecond().clone(); }
}
```

Let's try it yourself

Type erasure



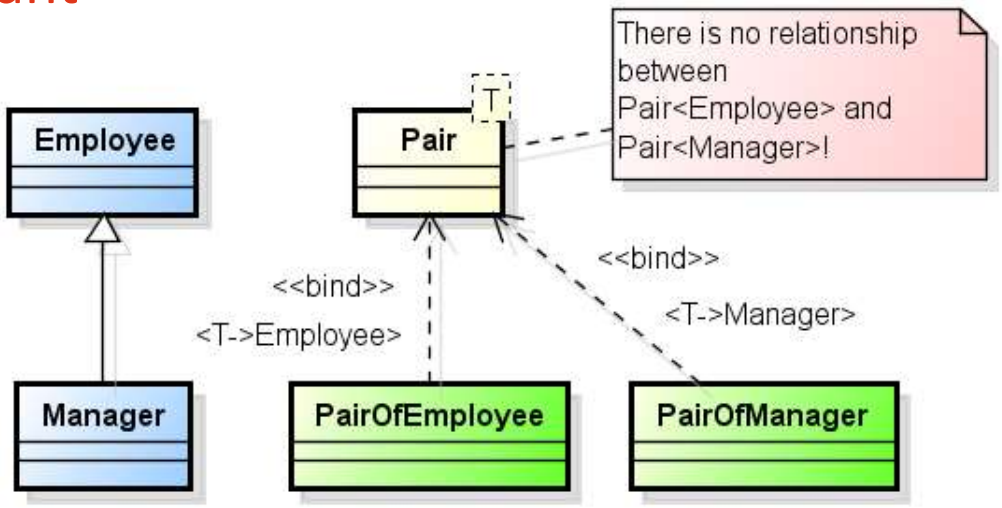
Restrictions and limitations

- Type Parameters Cannot Be Instantiated with Primitive Types
- Runtime Type Inquiry Only Works with Raw Types
- You Cannot Create Arrays of Parameterized Types
- Varargs Warnings...
- You Cannot Instantiate Type Variables
- Type Variables Are Not Valid in Static Contexts of Generic Classes
- You Cannot Throw or Catch Instances of a Generic Class
- Beware of Clashes after Erasure

GENERICS AND INHERITANCE

- Generics and inheritance works togheter in an unintuitive way
  - In general, there is *no* relationship between `Pair<S>` and `Pair<T>`, no matter how `S` and `T` are related
    - Necessary for type safety
  - Generics are said invariant

```
// Type safety restriction
Pair<Manager> managerBuddies =
    new Pair<>(ceo, cfo);
// Illegal, but suppose it
// wasn't
Pair<Employee> employeeBuddies =
    managerBuddies;
employeeBuddies.setFirst(
    lowlyEmployee);
```



GENERICS AND INHERITANCE

- Unfortunately there is a way to bypass this type safety controls

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
// Only a compile-time warning. A ClassCastException
// is thrown at runtime
rawBuddies.setFirst(new File(". . ."));
```

  - But, this is the same behaviour we obtain using older version of Java (≤ 1.5)
    - NEVER use raw type of generics if you can
- Generic classes can extend or implement other generic types
  - For example, `ArrayList<T>` implements `List<T>`
    - So, an `ArrayList<Manager>` is subtype of `List<Manager>`

WILDCARD TYPES

- The type system derived is too rigid: wildcard types help to safetly relax some constraint

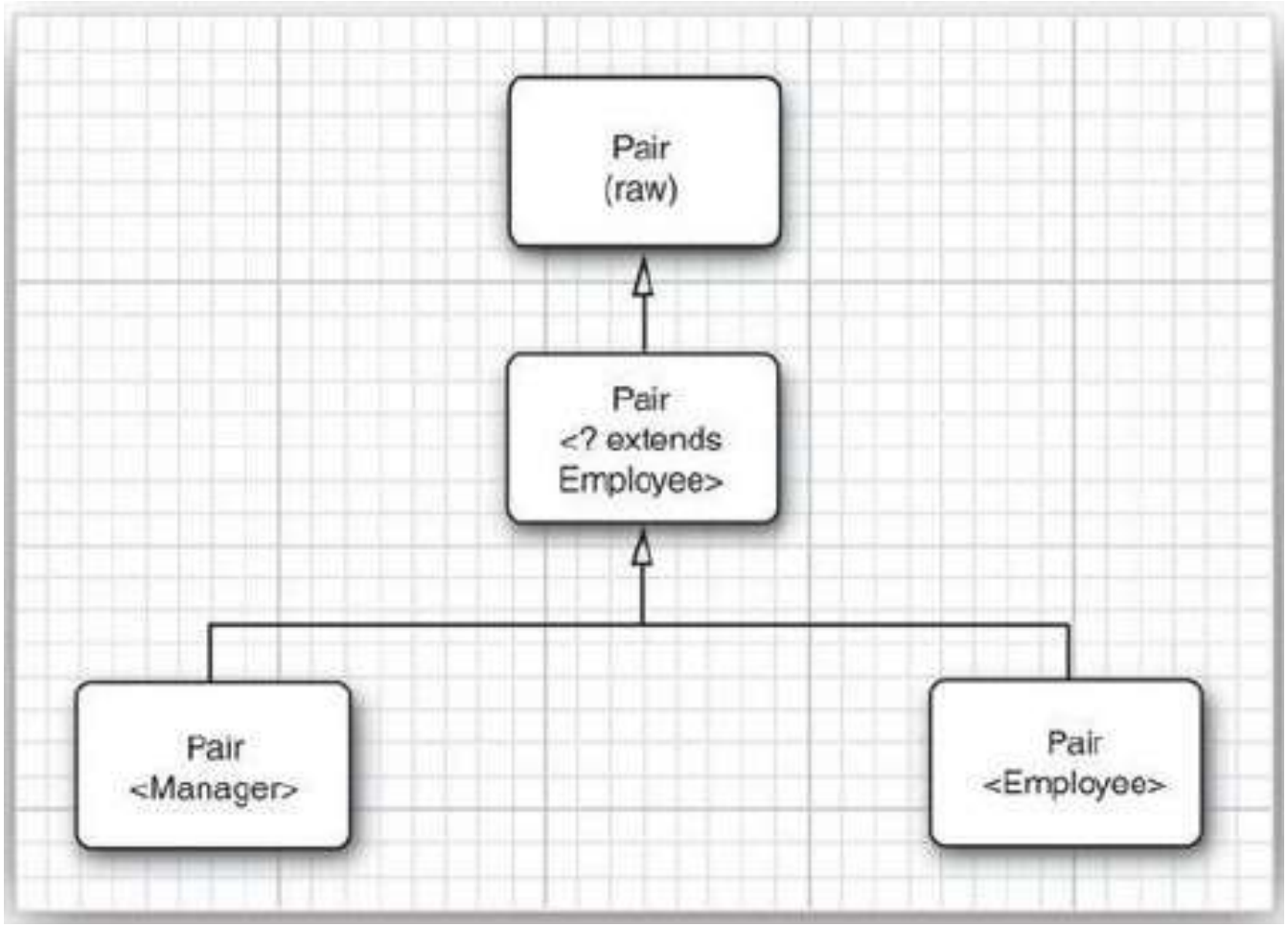
```
// Any generic Pair whose type parameter is a subclass of Employee
Pair<? extends Employee>
```

  - Why using wildcard types can we force type safety?
    - The compiler cannot guess the real type the object passed

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

  - Use wildcard type as return types in methods
    - In return types the compiler needs only to know which is the supertype, to allow assignments
    - Type `? extends T` is said covariant wrt type `T`

WILDCARD TYPES



WILDCARD TYPES

- How can we use wildcard type for parameters?
  - In Java are available supertype bounds

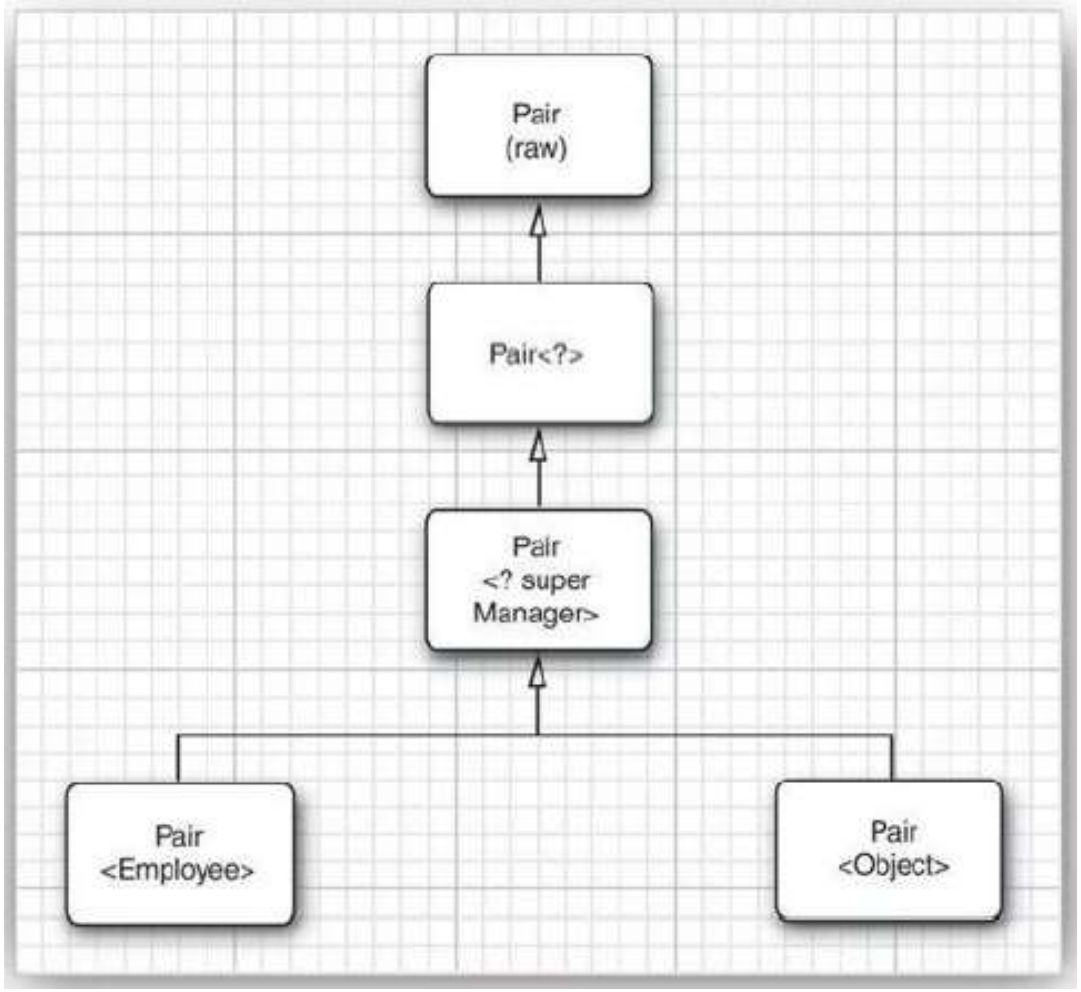
```
// Any generic Pair whose type parameter is restricted to all
// supertypes of Manager
Pair<? super Manager>
```

    - A wildcard with a *super type* bound gives you a behavior that is opposite to that of the wildcards
  - You can supply parameters to methods, but you can't use the return values

```
// Any generic Pair whose type parameter is restricted to all
// supertypes of Manager
Pair<? super Manager>
```

    - As a return values the only possible assignee type is `Object`
    - Type `? super T` is said contravariant wrt type `T`

WILDCARD TYPES



WILDCARD TYPES

- You can even use wildcards with no bounds

```
// It is different from the raw type Pair
Pair<?>
```

  - As a return value, can only be assigned to an `Object`
  - As a parameter, no type matches, not even Object
    - A method with a parameter with an unbounded wildcard type can never be called
      - Actually, you can call it passing a `null` reference...
  - Remember, an unbounded wildcard is not a type
    - So you can not use it when a type is requested

```
// This code is not valid, use type variables instead
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

WILDCARD TYPES

EXAMPLES

https://github.com/rcardin/pcd-snippets

REFERENCES

- Chap. 12 «Generic Programming», Core Java Volume I - Fundamentals, Cay Horstmann, Gary Cornell, 2012, Prentice Hall
- Chap. 6 «Generic Programming», Core Java for the Impatient, Cay Horstmann, 2015, Addison-Wesley
- Type Erasure https://docs.oracle.com/javase/tutorial/java/generics/erasure.html
- Effects of Type Erasure and Bridge Methods https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html
- Covariance and Contravariance In Java https://dzone.com/articles/covariance-and-contravariance
- Covariance, Invariance and Contravariance explained in plain English? http://stackoverflow.com/questions/8481301/covariance-invariance-and-contravariance-explained-in-plain-english