

Software Engineering Fundamentals

Advanced Implementation Mechanisms

Dr. Petru Florin Mihancea

Based on:
Bruce Eckel, Thinking in Java 4th Ed., Ch. Type information
Bruce Eckel, Thinking in Java 4th Ed., Ch. Generics

V20140411

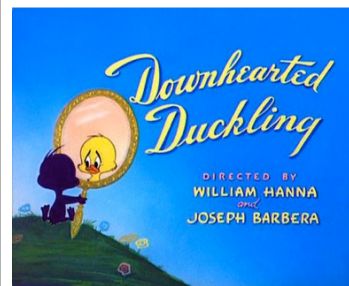
1

Reflection

in Java

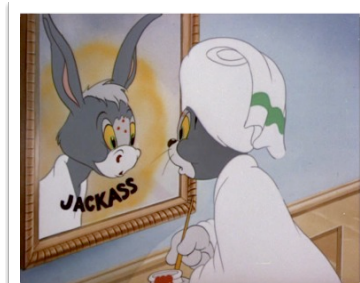
Dr. Petru Florin Mihancea

2



Wikipedia

Reflection



<http://www.tomandjerryonline.com>

... capability of a program to observe and manage **its own structure** at runtime

Dr. Petru Florin Mihancea

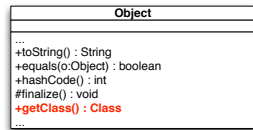
3

A

The **Class** class

Dr. Petru Florin Mihancea

4

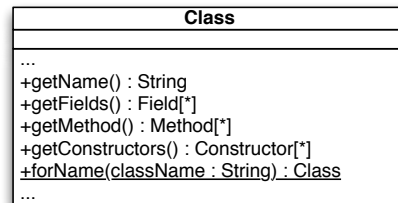


Meta-class **Class**

for example, the class `Object` is represented during a program execution by an instance of this class:

```

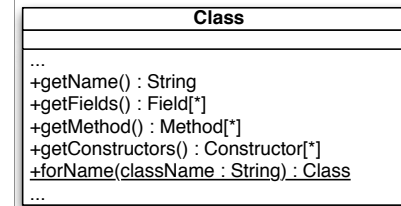
Class theObjectClass =
    Class.forName("java.lang.Object");
System.out.println(theObjectClass.getName());
  
```



an object of class **Class** represents a class from our program

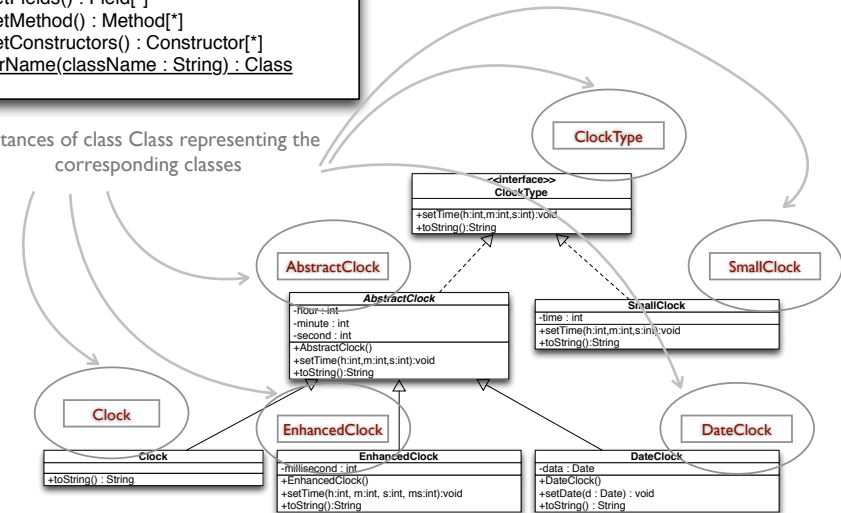
Dr. Petru Florin Mihances

5



Meta-class **Class (II)**

Instances of class `Class` representing the corresponding classes



Dr. Petru Florin Mihances

6

Be careful ...

in object-oriented programming the class notion and the object notion are (obviously) totally different

the previous notions do not change in reflection; it is only a matter of representing the entities from a program:

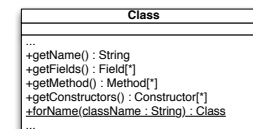
a class from a program is represented by an object and, like any other object, it is defined by a class (meta-class) named **Class**

Dr. Petru Florin Mihances

7

Accessing the **Class** objects

- 1) `getClass():Class` defined in `Object`
- 2) `ClassName.class` e.g., `Integer.class`, `int.class`, `Object.class`, etc.
- 3) for wrapper classes there is a special field **TYPE** e.g., `Integer.TYPE`
- 4) `forName(name:String):Class`



The fully qualified name

the class bytecode must be found at runtime in some folder specified in the classpath; otherwise the `ClassNotFoundException` is thrown

Dr. Petru Florin Mihances

8

References to **Class** objects

Class c; //can refer any Class object
 c = Integer.class;
 c = Object.class;

using another object, some errors might be observed very late (e.g., only at runtime); could the compiler help us ?

Class<Integer> i; //only the object representing the **Integer** class

i = Integer.class;
 i = int.class;
 i = Object.class; //Compile error

Class<Number> n; //only the object representing the **Number** class from the Java //library (superclass for all wrapper classes)

n = Number.class;
 n = Integer.class; //Compile error
 n = Object.class; //Compile error

Class<? extends Number> nb; //the Class object representing the **Number** class //or one of its subclasses

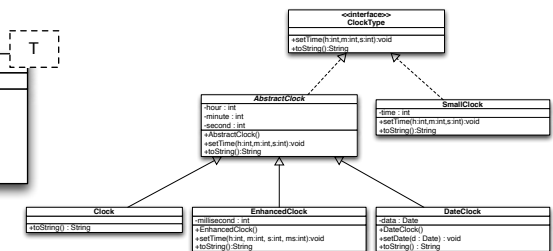
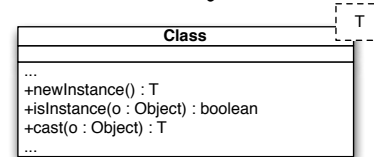
nb = Number.class;
 nb = Integer.class;
 nb = Object.class; //Compile error

Dr. Petru Florin Măhăceanu

9

Interesting methods in **Class**

The Class class is generic



```

Class clockClass = Clock.class;
Clock aClock = (Clock) clockClass.newInstance();

Class<Clock> clockClassGeneric = Clock.class;
aClock = clockClassGeneric.newInstance();
  
```

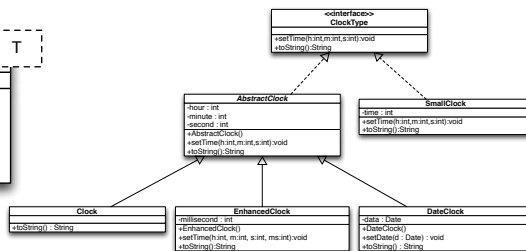
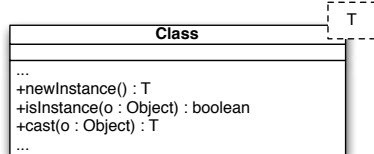
creates an instance of that class ;)
 the class must have a no-arg constructor
 otherwise an exception is raised
 (there are other reflection ways to create objects when there isn't a no-arg constructor)

Dr. Petru Florin Măhăceanu

10

Interesting methods in **Class**

The Class class is generic



```

Class clockTypeClass = ClockType.class;
if(clockTypeClass.isInstance(someObjectRef)) {
    ...
}
  
```

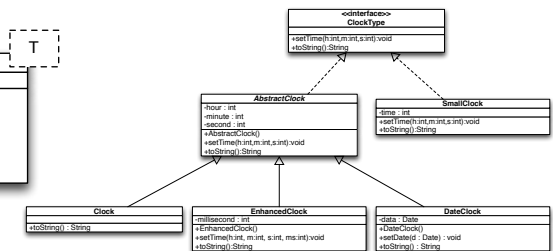
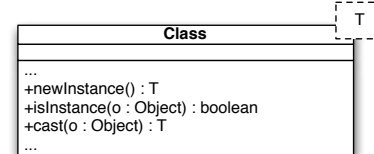
true if the object referred by someObjectRef is of the type represented by the object referred by clockTypeClass (like an instanceof)

Dr. Petru Florin Măhăceanu

11

Interesting methods in **Class**

The Class class is generic



```

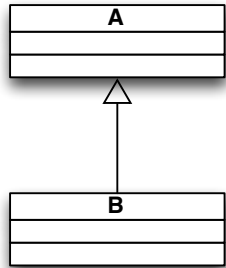
Class clockTypeClass = ClockType.class;
if(clockTypeClass.isInstance(someObject)) {
    ClockType aClock = (ClockType)clockTypeClass.cast(someObject);
    ...
}
or using generics
Class<ClockType> clockTypeClass = ClockType.class;
if(clockTypeClass.isInstance(someObject)) {
    ClockType aClock = clockTypeClass.cast(someObject);
    ...
}
  
```

We can change at runtime the instantiated class or the class "used in an instanceof" i.e., isInstance (this is not possible when using the new or instanceof operators)

Dr. Petru Florin Măhăceanu

12

Quiz



Object x = new B();

x instanceof B

TRUE

x instanceof A

TRUE

x.getClass().equals(A.class);

FALSE

x.getClass() == A.class;

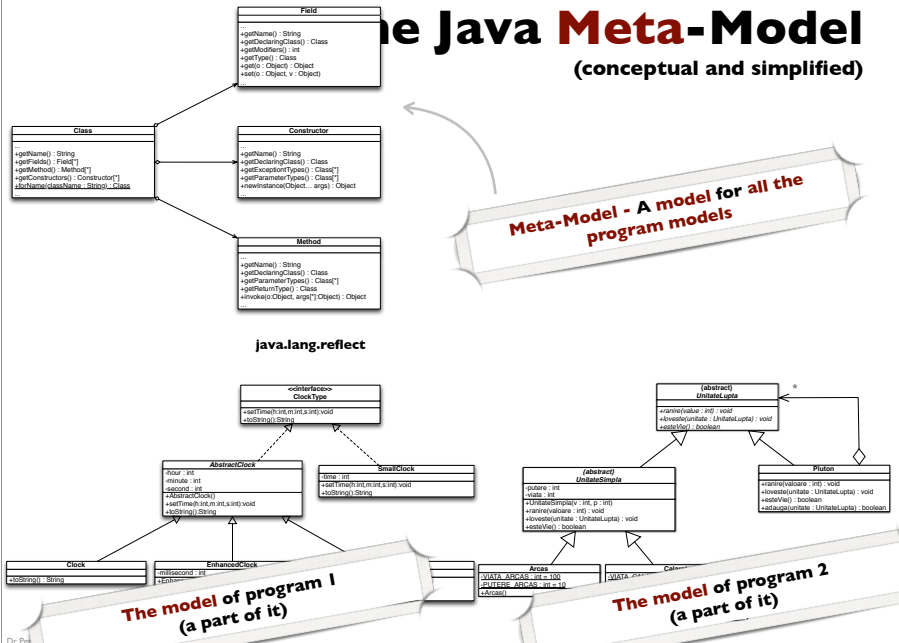
FALSE

x.getClass().equals(B.class);

TRUE

The **Class** class is only the beginning :)

The Java Meta-Model (conceptual and simplified)



Why ?

In these examples, the folders referred by a relative path are in the current working folder

Example 1 - A Mirror :)

```
package exemple.oglinda;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
public class Mirror {
    public static void main(String[] args) {
        for(String aClassName : args) {
            try {
                Class aClass = Class.forName(aClassName);
                System.out.println(aClass.getName());
                System.out.println("Is an interface? : " + aClass.isInterface());
                for(Field aField : aClass.getFields()) {
                    System.out.println("Field name: " + aField.getName());
                    System.out.println("Field type: " + aField.getType().getName());
                }
                for(Method aMethod : aClass.getMethods()) {
                    System.out.println("Method name: " + aMethod.getName());
                    System.out.println("Method return type: " + aMethod.getReturnType().getName());
                    for(int i = 0; i < aMethod.getParameterTypes().length; i++) {
                        System.out.println("Arg " + i + " : " + aMethod.getParameterTypes()[i].getName());
                    }
                }
                System.out.println();
            } catch(ClassNotFoundException e) {
                System.err.println("Class " + aClassName + " not found!");
            }
        }
    }
}
```

Dr. Pedro Florin Mhancas

17

Example 1 - A Mirror :)

```
package exemple.oglinda;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
public class Mirror {
    public static void main(String[] args) {
        for(String aClassName : args) {
            try {
                Class aClass = Class.forName(aClassName);
                System.out.println(aClass.getName());
                System.out.println("Is an interface? : " + aClass.isInterface());
                for(Field aField : aClass.getFields()) {
                    System.out.println("Field name: " + aField.getName());
                    System.out.println("Field type: " + aField.getType().getName());
                }
                for(Method aMethod : aClass.getMethods()) {
                    System.out.println("Method name: " + aMethod.getName());
                    System.out.println("Method return type: " + aMethod.getReturnType().getName());
                    for(int i = 0; i < aMethod.getParameterTypes().length; i++) {
                        System.out.println("Arg " + i + " : " + aMethod.getParameterTypes()[i].getName());
                    }
                }
                System.out.println();
            } catch(ClassNotFoundException e) {
                System.err.println("Class " + aClassName + " not found!");
            }
        }
    }
}
```

```
Pepis-MacBook-Pro:Desktop Pepi$ javac -d bin/ Mirror.java
Pepis-MacBook-Pro:Desktop Pepi$ java -cp bin exemple.oglinda.Mirror
Pepis-MacBook-Pro:Desktop Pepi$ java -cp bin exemple.oglinda.Mirror exemple.oglinda.Mirror
exemple.oglinda.Mirror
Is an interface? false
Method name:main
Method return type:void
Arg 0:[Ljava.lang.String;
Method name:wait
Method return type:void
Arg 0:long
Arg 1:int
Method name:wait
Method return type:void
Arg 0:long
Method name:wait
Method return type:void
Method name:equals
Method return type:boolean
Arg 0:java.lang.Object
Method name:toString
Method return type:java.lang.String
Method name:hashCode
Method return type:int
Method name:getClass
Method return type:java.lang.Class
Method name:notify
Method return type:void
Method name:notifyAll
Method return type:void
```

And we can do this for any class by specifying its fully qualified name; its bytecode must be found at runtime in the classpath folders

Dr. Pedro Florin Mhancas

17

Example 2 - Plugins

```
package exemple.plugins;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        String aPluginClassName;
        while((aPluginClassName = bf.readLine()) != null) {
            try {
                Class loadedClass = Class.forName(aPluginClassName);
                Class pluginClass = Plugin.class;
                Object anObject = loadedClass.newInstance();
                if(pluginClass.isInstance(anObject)) {
                    Plugin aPlugin = (Plugin)anObject;
                    aPlugin.execute();
                }
            } catch(ClassNotFoundException e) {
                System.out.println("Plugin class not found!");
            } catch(InstantiationException | IllegalAccessException e) {
                System.out.println("Plugin class cannot be instantiated - concrete/accessible/no-arg constructors!");
            }
        }
    }
}
```

```
package exemple.plugins;
public interface Plugin {
    public void execute();
}
```

Dr. Pedro Florin Mhancas

18

Example 2 - Plugins

```
package exemple.plugins;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        String aPluginClassName;
        while((aPluginClassName = bf.readLine()) != null) {
            try {
                Class loadedClass = Class.forName(aPluginClassName);
                Class pluginClass = Plugin.class;
                Object anObject = loadedClass.newInstance();
                if(pluginClass.isInstance(anObject)) {
                    Plugin aPlugin = (Plugin)anObject;
                    aPlugin.execute();
                }
            } catch(ClassNotFoundException e) {
                System.out.println("Plugin class not found!");
            } catch(InstantiationException | IllegalAccessException e) {
                System.out.println("Plugin class cannot be instantiated - concrete/accessible/no-arg constructors!");
            }
        }
    }
}
```

```
package exemple.plugins;
public interface Plugin {
    public void execute();
}
```

```
Pepis-MacBook-Pro:Desktop Pepi$ javac -d bin Plugin.java Main.java
Pepis-MacBook-Pro:Desktop Pepi$ java -cp bin:/Users/Pepi/Desktop/dropplugins/
exemple.plugins.Main
test
Plugin class not found!
exemple.plugins.Plugin
Plugin class cannot be instantiated - concrete/accessible/no-arg constructors?
personal.MyPlugin
Plugin class not found!
```

Dr. Pedro Florin Mhancas

18

Example 2 - Plugins (II)

```
package personal;

public class MyPlugin implements exemple.plugins.Plugin {
    public void execute() {
        System.out.println("My plugin is executing!");
    }
}
```

```
javac -cp bin/ -d dropplugins/ MyPlugin.java
```

```
Pepis-MacBook-Pro:Desktop Pepi$ javac -d bin Plugin.java Main.java
Pepis-MacBook-Pro:Desktop Pepi$ java -cp bin:/Users/Pepi/Desktop/dropplugins/
exemple.plugins.Main
test
Plugin class not found!
exemple.plugins.Plugin
Plugin class cannot be instantiated - concrete/accessible/no-arg constructors?
personal.MyPlugin
Plugin class not found!
personal.MyPlugin
My plugin is executing!
```

Reloading a modified class
is not that simple :)

Dr. Petru Florin Mihai

19

Example 3 - Plugin even more flexible

```
package exemple.maiflexibil;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        String aPluginClassName;
        while((aPluginClassName = bf.readLine()) != null) {
            try {
                Class loadedClass = Class.forName(aPluginClassName);
                Object anObject = loadedClass.newInstance();
                String methodName = bf.readLine();
                Method theMethod = loadedClass.getMethod(methodName, new Class[] {}); //We search a method with this name and no args
                theMethod.invoke(anObject, new Object[] {}); //We can specify actual arguments when they exist :)
            } catch (ClassNotFoundException e) {
                System.out.println("Plugin class not found!");
            } catch (InstantiationException | IllegalAccessException e) {
                System.out.println("Plugin class cannot be instantiated - concrete/accessible/no-arg constructors?");
            } catch (NoSuchMethodException e) {
                System.out.println("Method not found!");
            } catch (SecurityException e) {
                System.out.println("Security exception: " + e);
            } catch (IllegalArgumentException e) {
                System.out.println("Illegal arguments exception: " + e);
            } catch (InvocationTargetException e) {
                System.out.println("Invocation target exception: " + e);
            }
        }
    }
}
```

Dr. Petru Florin Mihai

20

Example 3 - Plugin even more flexible

```
package exemple.maiflexibil;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        String aPluginClassName;
        while((aPluginClassName = bf.readLine()) != null) {
            try {
                Class loadedClass = Class.forName(aPluginClassName);
                Object anObject = loadedClass.newInstance();
                String methodName = bf.readLine();
                Method theMethod = loadedClass.getMethod(methodName, new Class[] {}); //We search a method with this name and no args
                theMethod.invoke(anObject, new Object[] {}); //We can specify actual arguments when they exist :)
            } catch (ClassNotFoundException e) {
                System.out.println("Plugin class not found!");
            } catch (InstantiationException | IllegalAccessException e) {
                System.out.println("Plugin class cannot be instantiated - concrete/accessible/no-arg constructors?");
            } catch (NoSuchMethodException e) {
                System.out.println("Method not found!");
            } catch (SecurityException e) {
                System.out.println("Security exception: " + e);
            } catch (IllegalArgumentException e) {
                System.out.println("Illegal arguments exception: " + e);
            } catch (InvocationTargetException e) {
                System.out.println("Invocation target exception: " + e);
            }
        }
    }
}
```

```
javac -d bin Main.java
```

```
java -cp bin:/Users/Pepi/Desktop/drop/ exemple.maiflexibil.Main
test
Plugin class not found!
```

Dr. Petru Florin Mihai

20

Example 3 - Plugin even more flexible (II)

```
package personal;
public class SomeClass {
    public void exec() {
        System.out.println("exec method is executed now :");
    }
}
```

```
javac -d drop SomeClass.java
```

```
java -cp bin:/Users/Pepi/Desktop/drop/ exemple.maiflexibil.Main
test
Plugin class not found!
personal.SomeClass
metoda
Method not found!
personal.SomeClass
exec
exec method is executed now :)
```

Dr. Petru Florin Mihai

21

2

Generics (parametric polymorphism) in Java

Dr. Petru Florin Mihalcea

22

A

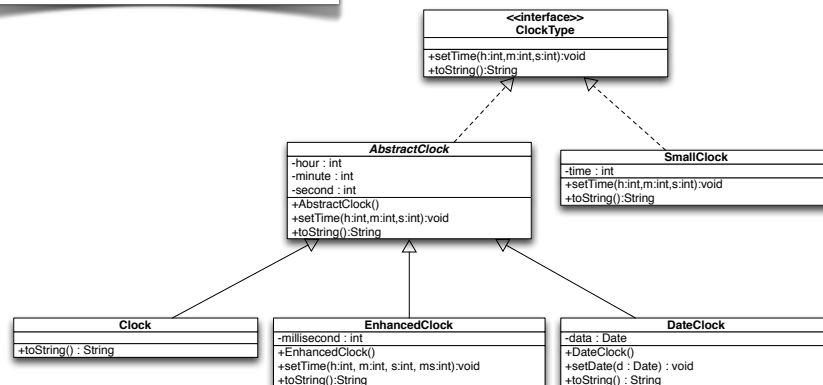
The Basics

Dr. Petru Florin Mihalcea

23

Working Example

```
class ClockMaker {
    public void fixTime(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```



Dr. Petru Florin Mihalcea

24

We need ...

... a class representing a 2-tuple

- a pair of two **elements of any type**
- constructor/methods to initialize/set the pair elements
- one method to access each element

“Solution I” : using
(subtype) polymorphism,
because an **Object**
reference can refer every
type of objects

```
public class Pair {
    private Object p1, p2;
    public Pair(Object p1, Object p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(Object p1) {
        this.p1 = p1;
    }
    public Object getFirst() {
        return p1;
    }
    public void setSecond(Object p2) {
        this.p2 = p2;
    }
    public Object getSecond() {
        return p2;
    }
}
```

Dr. Petru Florin Mihalcea

25


```
public class Utilities {
    private static Pair doSet(Pair p) {
        ClockMaker man = p.getFirst(); //Compile error
        ClockType device = p.getSecond(); //Compile error
        man.fixTime(device);
        return p;
    }
    public static void createSetAndDisplay(Clock device) {
        ClockMaker man = new ClockMaker();
        Pair p = new Pair(man,device);
        Pair res = doSet(p);
        ClockType c = res.getSecond(); //Compile error
        System.out.println(c);
    }
}
```

BUT ...

Dr. Petru Florin Măhănea

26

```
public class Utilities {
    private static Pair doSet(Pair p) {
        ClockMaker man = (ClockMaker)p.getFirst();
        ClockType device = (ClockType)p.getSecond();
        man.fixTime(device);
        return p;
    }
    public static void createSetAndDisplay(Clock device) {
        ClockMaker man = new ClockMaker();
        Pair p = new Pair(man,device);
        Pair res = doSet(p);
        ClockType c = (ClockType)res.getSecond();
        System.out.println(c);
    }
}
```

BUT ...

Dr. Petru Florin Măhănea

26

... many errors may appear
because of incompatible types and
they will be observed only at
runtime

```
public class Utilities {
    private static Pair doSet(Pair p) {
        ClockMaker man = (ClockMaker)p.getFirst();
        ClockType device = (ClockType)p.getSecond();
        man.fixTime(device);
        return p;
    }
    public static void createSetAndDisplay(Clock device) {
        ClockMaker man = new ClockMaker();
        Pair p = new Pair(man, device);
        Pair res = doSet(p);
        ClockType c = (ClockType)res.getSecond();
        System.out.println(c);
    }
    public static void iWouldLikeToKnowAtCompileTime() {
        Pair p1 = new Pair(new Integer(5),new Integer(6));
        doSet(p1); //Runtime error
        Pair p2 = new Pair(new ClockMaker(),new Clock());
        doSet(p2);
        Pair p3;
        p3 = p2; //Risky, what if the elements of the pair
        doSet(p3); //are not of the corresponding type ?
        p3 = p1; //Risky, what if the elements of the pair
        doSet(p3); //are not of the corresponding type ?
    }
}
```

BUT ...

Dr. Petru Florin Măhănea

26

... many errors may appear
because of incompatible types and
they will be observed only at
runtime

```
public class Utilities {
    private static PairClockMakerClockType doSet(PairClockMakerClockType p) {
        ClockMaker man = p.getFirst();
        ClockType device = p.getSecond();
        man.fixTime(device);
        return p;
    }
    public static void createSetAndDisplay(Clock device) {
        ClockMaker om = new ClockMaker();
        PairClockMakerClockType p =
            new PairClockMakerClockType(man,device);
        PairClockMakerClockType res = doSet(p);
        ClockType c = res.getSecond();
        System.out.println(c);
    }
    public static void iWouldLikeToKnowAtCompileTime() {
        PairIntegerInteger p1 =
            new PairIntegerInteger(new Integer(5),new Integer(6));
        doSet(p1); //Compile error
        PairClockMakerClockType p2 =
            new PairClockMakerClockType(
                new ClockMaker(),new Clock());
        doSet(p2);
        PairClockMakerClockType p3;
        p3 = p2;
        doSet(p3);
        p3 = p1; //Compile error
        doSet(p3);
    }
}
```

olution 2"

```
public class PairClockMakerClockType {
    private ClockMaker p1; ClockType p2;
    public Pair(ClockMaker p1, ClockType p2) {
        this.p1 = p1; this.p2 = p2;
    }
    public void setFirst(ClockMaker p1) {this.p1 = p1;}
    public ClockMaker getFirst() {return p1;}
    public void setSecond(ClockType p2) {this.p2 = p2;}
    public ClockType getSecond() {return p2;}
}
```

```
public class PairIntegerInteger {
    private Integer p1, p2;
    public Pair(Integer p1, Integer p2) {
        this.p1 = p1; this.p2 = p2;
    }
    public void setFirst(Integer p1) { this.p1 = p1;}
    public Integer getFirst() { return p1;}
    public void setSecond(Integer p2) { this.p2 = p2;}
    public Integer getSecond() { return p2;}
}
```

Dr. Petru Florin Măhănea

27

but many other similar classes ...

Solution

type parameters

```

public class Pair<T,K> {
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}

```

generic type/
class

type argument
e.g. Integer

Pair<Integer, Integer>
Pair<Integer, Clock>
...

Parameterized types
- or "invocations" of the
generic type

in UML

```

classDiagram
    class Pair {
        -p1 : T
        -p2 : K
        +Pair(p1 : T, p2 : K)
        +setFirst(p1 : T) : void
        +getFirst() : T
        +setSecond(p2 : K) : void
        +getSecond() : K
    }
    Pair "T,K"

```

Creating objects of a
parameterized type

new Pair<Integer, Integer> ();

Dr. Petru Florin Mihances

28

Solution

```

public class ClockCommand {
    public static Pair<ClockMaker, ClockType>
        doSet(Pair<ClockMaker, ClockType> p) {
        ClockMaker man = p.getFirst();
        ClockType device = p.getSecond();
        man.fixTime(device);
        return p;
    }
    public static void createSetAndDisplay(Clock device) {
        ClockMaker man = new ClockMaker();
        Pair<ClockMaker, ClockType> p =
            new Pair<ClockMaker, ClockType>(man, device);
        Pair<ClockMaker, ClockType> res = doSet(p);
        ClockType c = res.getSecond();
        System.out.println(c);
    }
    public static void iWouldLikeToKnowAtCompileTime() {
        Pair<Integer, Integer> p1 =
            new Pair<Integer, Integer>(new Integer(5), new Integer(6));
        doSet(p1); // Compile error
        Pair<ClockMaker, ClockType> p2 =
            new Pair<ClockMaker, ClockType>(
                new ClockMaker(), new Clock()); // We can put subtypes
        doSet(p2);
        Pair<ClockMaker, ClockType> p3;
        p3 = p2;
        doSet(p3);
        p3 = p1; // Compile error
        doSet(p3);
    }
}

```

```

public class Pair<T,K> {
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}

```

Dr. Petru Florin Mihances

29

Implements/Extends (I)

```

public interface PairInterface<T,K> {
    public T getFirst();
    public K getSecond();
}

```

```

public class Pair<T,K>
    implements PairInterface<T, K>{
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}

```

```

public class Triple<T,K,Z> extends Pair<T,K> {
    private Z p3;
    public Triple(T p1, K p2, Z p3) {
        super(p1, p2);
        this.p3 = p3;
    }
    public void setThird(Z p3) {
        this.p3 = p3;
    }
    public Z getThird() {
        return p3;
    }
}

```

Dr. Petru Florin Mihances

30

Implements/Extends (II)

```

public class MySpecialBooleanPair implements PairInterface<Boolean, Boolean> {
    private boolean b1, b2;
    public MySpecialBooleanPair(boolean b1, boolean b2) {
        this.b1 = b1;
        this.b2 = b2;
    }
    public Boolean getFirst() {
        return b1;
    }
    public Boolean getSecond() {
        return b2;
    }
}

```

```

public class TripleWith2Integers extends Pair<Integer, Integer> {
    private Integer p3;
    public TripleWith2Integers(Integer p1, Integer p2, Integer p3) {
        super(p1, p2);
        this.p3 = p3;
    }
    ...
}

```

Dr. Petru Florin Mihances

31

Polymorphism (subtype)

```
PairInterface<Integer,Integer> a;  
a = new Pair<Integer,Integer>(new Integer(5), new Integer(6));  
System.out.println(a);  
a = new Triple<Integer,Integer,Integer>(new Integer(5), new Integer(6), new Integer(8));  
System.out.println(a);  
a = new TripleWith2Integers(new Integer(0), new Integer(9), 0);  
System.out.println(a);  
a = new MySpecialBooleanPair(false, false); //Compile error
```

As long as the type arguments do not vary, the supertype/subtype relation holds (with all the other implications)

Dr. Petru Florin Mihances

32

Many Limitations (in Java)

a. Type arguments cannot be of primitive type

e.g. `Pair<int,int>` - compile error (solution: wrapper classes e.g. `Integer`)

b. We cannot instantiate a type parameter

e.g. `new T()`; - compile error (solution: use reflection)

c. We cannot create arrays containing a type parameter references

e.g. `new T[...]`; - compile error (but we can declare references e.g. `T[] x`); (solution: use collections/containers)

d. We cannot use instanceof

e.g. `x instanceof T` - compile error (solution: use reflection)

... (see also the next slide)

Dr. Petru Florin Mihances

33

Many Limitations (in Java)

a. Type arguments cannot be of primitive type

e.g. `Pair<int,int>` - compile error (solution: wrapper classes e.g. `Integer`)

b. We cannot instantiate a type parameter

e.g. `new T()`; - compile error (solution: use reflection)

c. We cannot create arrays containing a type parameter references

e.g. `new T[...]`; - compile error (but we can declare references e.g. `T[] x`); (solution: use collections/containers)

d. We cannot use instanceof

e.g. `x instanceof T` - compile error (solution: use reflection)

... (see also the next slide)

These limitations appear due to the way in which generics are implemented in the Java language (using erasure - does not exist / it is not maintained information regarding what actual type arguments are used for the type parameters); in other languages, genericity is implemented in other ways and thus, it is stronger

Dr. Petru Florin Mihances

33

Bounded type parameters (I)

```
public class NamedGenericPair<T,K> {  
    private T p1;  
    private K p2;  
    public NamedGenericPair(T p1, K p2) {  
        this.p1 = p1;  
        this.p2 = p2;  
    }  
    public void setFirst(T p1) {  
        this.p1 = p1;  
    }  
    public T getFirst() {  
        return p1;  
    }  
    public void setSecond(K p2) {  
        this.p2 = p2;  
    }  
    public K getSecond() {  
        return p2;  
    }  
    public String toString() {  
        return "(" + p1.toString() + "," + p2.toString() + ")";  
    }  
}
```

on a variable having as type a type parameter we can invoke only methods from the Object class

Dr. Petru Florin Mihances

34

Bounded type parameters (II)

```
public class NamedGenericPair<T extends NamedEntity, K extends NamedEntity> {
    private T p1;
    private K p2;
    public NamedGenericPair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
    public String toString() {
        return "(" + p1.getName() + "," + p1.toString() + "," + p2.getName() + "," + p2.toString() + ")";
    }
}
```

```
public interface NamedEntity {
    public String getName();
}
```

In general, the form is
T extends Type1 & Type2 & ... & TypeName
(one class at most and it must be the first)

We set an upper limit to the type parameters: to be of NamedEntity type or of one of its subtypes
NamedGenericPair<Integer,Integer> x; //Compile error
NamedGenericPair<NamedEntity,NamedEntity> y;

Dr. Petru

35

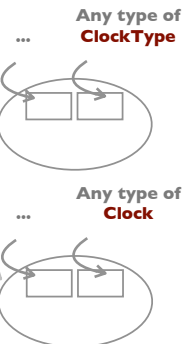
Dr. Petru Florin Mihai

36

Some “evident” and some “weird” things (when first seen)

```
public class Warning {
    public static void main(String[] args) {
        ClockMaker person = new ClockMaker();
        Clock c = new Clock();
        EnhancedClock ec = new EnhancedClock();

        Pair<ClockMaker,ClockType> a = new Pair<ClockMaker,ClockType>(person,c);
        a.setSecond(ec); //We can use a reference of the type argument type or of one of its
                        //subtypes
        Pair<ClockMaker,Clock> b = new Pair<ClockMaker,Clock>(person,c);
        b.setSecond(c);
        b.setSecond(ec); //Compile error - b must be of Clock type/subtype
        a = b; //Compile error !!!
        a.setSecond(ec);
    }
}
```



a - refers a pair object that may contain a **ClockMaker** and any object having the **ClockType** type

b - refers a pair object that may contain a **ClockMaker** and any object having the **Clock** type

Explanation: If the assignment would be correct then in the pair referred by a and b we could put as the second element an EnhancedClock (next line) although the pair object **CANNOT** contain an EnhancedClock (it can contain only Clock types); thus, a runtime error would be raised

Dr. Petru Florin Mihai

37

B

Some more advanced elements ...

Dr. Petru Florin Mihai

36

Quiz

Is this line ok ?

```
Pair<ClockMaker,ClockType> x = new Pair<ClockMaker,Clock>(person,c);
```

NO!

x - refers a pair object that can contain a **ClockMaker** and any kind of **ClockType**

... but the created object can contain only **ClockMaker** and **Clock** !!!

Dr. Petru Florin Mihai

38

Upper Bounded Wildcard

```
public class Warning {  
    public static void main(String[] args) {  
        ClockMaker person = new ClockMaker();  
        Clock c = new Clock();  
        EnhancedClock ec = new EnhancedClock();  
  
        Pair<ClockMaker, ? extends ClockType> a = new Pair<ClockMaker, Clock>(person, c);  
        a.setSecond(c); //Compile error - the compiler cannot guarantee that the previous situation does not  
                        //appear at runtime (that the referred object can contain a Clock)  
        Pair<ClockMaker, Clock> b = new Pair<ClockMaker, Clock>(person, c);  
  
        a = b; //Ok.  
        a.setSecond(ec); //Compile error - as the previous one; we can do only a.setSecond(null);  
    }  
}
```

a - reference to

a pair **object** that can contain a **ClockMaker** and any kind of **ClockType**

OR

a pair **object** that can contain a **ClockMaker** and any kind of **Clock**

OR

When we have `class G<T> {}`
and a reference

`G<? extends Something> g;`

at the invocations of the methods
having an argument of type T we
can use only null as a value for
that argument

there is also the possibility to have
lower bounded wildcards :)