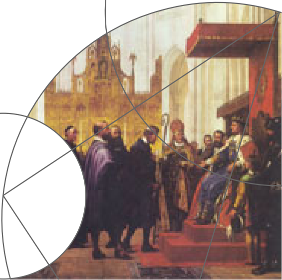
Generics and reflection

Frederik M. Madsen
fmma@diku.dk



Generics

Motivation - Dynamic checking

- Consider a *pair* class that can hold a pair of objects:

```
class DynamicCheckedPair {
  Object first;
  Object second;
}
```

- Given a pair `p`, we can put *any* value into `first` and `second`:

  `p.first = "hello";`
- We cannot get anything but an `Object` out of the pair.
- We have to *cast*:

  `String x = (String)p.first;`
- Casting can cause runtime exception. ☹
  - Dynamic/runtime check.

Motivation - Static checking

- To get static checking, we cannot use casts. Instead make a class for every pair needed:

```
class PairStringInteger {
  String first;
  Integer second;
}
```

- Like before we can put a string into `first`:
  `p.first = "hello";` ☺
- … but not an integer:
  `p.first = 42;` ☹
- No need to cast:
  `String x = p.first;` ☺
- Many different pairs needed → lots of boilerplate.
- `Pair` as a library class → all pairs (not possible).

Generic classes

- No more boilerplate with *generics*:

```
class Pair<A,B> {
  A first;
  B second;
}
```

- *Parametric polymorphism*. `A` and `B` are *type parameters*.
- `Pair` is a type indexed family of classes.
  - `Pair` : *Type* $\times$ *Type* $\rightarrow$ *Class*.
  - `Pair<String, Double>` is a class.
    - Equivalent to `PairStringDouble`.
    - `String` and `Double` are called *type arguments*.

Generic classes

- Type parameters can also be used in methods and constructors:

```
class Pair<A,B> {
  private A first;
  private B second;

  Pair(A x, B y) {
    first = x; second = y;
  }

  A getFirst() {
    return first;
  }

  void setFirst(A x) {
    first = x;
  }
}
```

Generic classes

- Type parameters can also be used in methods and constructors:

```
class Pair<A,B> {
  private A first;
```

Discuss:

Say we want pairs to have a method `swap` that return a new pair with `first` and `second` swapped.

1. What would the method signature be?
2. What would the implementation look like?

```
  void setFirst(A x) {
    first = x;
  }
}
```

Generic classes

- Type parameters can also be used in methods and constructors:

```
class Pair<A,B> {
```

Answers:

```
public Pair<B, A> swap() {
    Pair<B, A> p = new Pair<B, A>();
    p.first = this.second;
    p.second = this.first;
    return p;
}
```

```
    void setFirst(A x) {
      first = x;
    }
}
```

Generic interfaces

- It is also possible to declare generic interfaces.

```
interface Mutation<A> {
  void mutate(A x);
}

class LowerCaseName implements Mutation<Employee> {
  void mutate(Employee e) { ... }
}

class IdentityMutation<A> implements Mutation<A> {
    void mutate(A x) { ; }
}
```

Generic methods

- Individual methods can also be generic:

```
public <A> A printReturn(A x) {
    System.out.println(x);
    return x;
}
```

- Here, type parameter `A` is not associated with the class instance. The caller must supply type argument:

```
obj.<String>printReturn("hello");
obj.<Integer>printReturn(42);
```

- Type argument can usually be inferred:

```
obj.printReturn("hello");
```

Generic classes - Instances

- Constructors take type arguments as well:

```
Pair<String, Double> p1 = new Pair<String, Integer>();
Pair<Double, Double> p2 = new Pair<Double, Double>(45.3, 0.1);

p1.setFirst("Hello");
Double x = p2.getFirst() + p2.getSecond();
```

- In Java 7, you can use the “diamond operator” <> to infer type arguments:

```
Pair<String, Double> p1 = new Pair<>();
Pair<Double, Double> p2 = new Pair<>(45.3, 0.1);
```

Generics - Other examples

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`
- `Map<K,V>`
  - `V put(K key, V value)`
  - `V get(Object o)`
- `Comparator<T>`
  - `int compare(T o1, T o2)`
- `Iterator<E>`
  - `E next()`
- `Class<T>`
  - `T newInstance()`

Eclipse demonstration: Making assignment 1 generic.

Generic classes - Instances

- Constructors take type arguments as well:

```
Pair<String, Double> p1 = new Pair<String, Integer>();
Pair<Double, Double> p2 = new Pair<Double, Double>(45.3, 0.1);

p1.setFirst("Hello");
Double x = p2.getFirst() + p2.getSecond();
```

- Omitting type arguments gives *raw* types

```
Pair p1 = new Pair();
```

  - Type parameters are then *erased* to most general type (`Object`).
  - Equivalent to `DynamicCheckedPair`.
  - Not equivalent to `Pair<Object, Object>`.
    - Subtle difference related to subtyping.
    - This is where generics gets messy.
  - Never use raw types.

Generic classes - Runtime representation

- Bytecode does not support parametric polymorphism.
  - Generics retrofitted into Java.
- Actual implementation of generics is sort of hacky.
- For all type arguments, a generic class is represented as the raw class.
  - `Pair<String, Double>` represented as `Pair`.
  - `Pair<Foo, Bar>` represented as `Pair`.
  - Called *Type erasure*.
- Type indexed family of classes is only an illusion.
  - ... but it is a type-checked illusion:

```
Pair<Double, Double> p = new Pair<Double, Double>();
p.first = "Helloo"; //Compile-time error, would run.
Double x = p.first; //Insert (Double) cast.
String y = p.first; //Compile-time error.
```

Generic classes - Runtime representation

- Bytecode does not support parametric polymorphism.

Discuss:

1. What would be the result of the following?

```java
Pair<String, Integer> p1 = ...;
Pair<Double, Double> p2 = ...;
return p1.getClass()== p2.getClass();
```

2. Is `Pair<A,B>` a complete replacement for `DynamicCheckedPair`?
3. Is `Pair<String, Integer>` a complete replacement for `PairStringInteger`?
4. Is the following legal?

```java
Pair<String, Integer> p = ...;
Pair<Object, Object> p2 = p;
```

```java
String y = p.first; //Compile-time error.
```

Generic classes - Runtime representation

- Bytecode does not support parametric polymorphism.

Answers:

1. true, both classes are the raw pair.
2. Yes, even though a single `DynamicCheckedPair` instance can be reused with different types, an instance of `Pair` (the raw type) can be used in exactly the same way.
3. Not quite. Since we only have the raw types at runtime, we cannot do stuff like
   `p instanceof Pair<String, Integer>`, but
   `p instanceof PairStringInteger` is perfectly fine.
4. No, `p2.first = 42;` is legal but that violates the type of `p`.

```
String y = p.first; //Compile-time error.
```

Subtyping in Java

- Subtype `A` $<$ `B` in Java means:
  - Subclass: `A extends B`.
  - Implementation: `A implements B`.
  - Transitive closure: `A` $<$ `C` and `C` $<$ `B` implies `A` $<$ `B`.
- The point is: If `A` $<$ `B`, then `A` can be used in place of `B`.
  - Instances of `A` is *assignable* to variables of type `B`.
  - Variables: Local variables, fields, method parameters.

Subtyping example

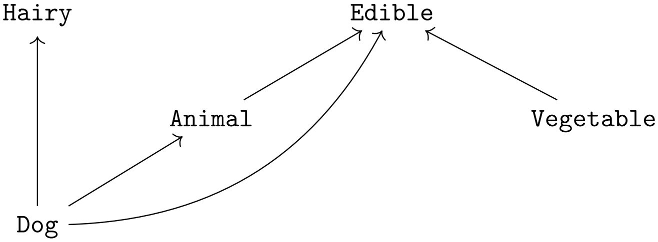
```
interface Edible {
  void eat();
}
interface Hairy {
  void groom();
}

class Animal implements Edible { .. }

class Vegetable implements Edible { .. }

class Dog extends Animal implements Hairy { .. }
```

Subtyping example



- `Vegetable` $<$ `Edible`
- `Animal` $<$ `Edible`
- `Dog` $<$ `Edible`
- `Dog` $<$ `Animal`
- `Dog` $<$ `Hairy`

Subtyping example

```
Dog dog = new Dog();
Vegetable veg = new Vegetable();
Vegetable veg2 = veg;

Hairy hairy1 = dog;
Hairy hairy2 = veg; // Error

Animal animal1 = dog;
Animal animal2 = hairy; // Error

Edible e = dog;
Edible e2 = veg;
Edible e3 = animal;
Edible e4 = hairy; // Error
```

Subtyping - Arrays

- Array types: `A` < `B` implies `A[]` < `B[]`.
  - Terminology: Arrays are *covariant*.
    - Subtyping two array types is the same as subtyping the two type arguments.
  - Looks reasonable at first glance.
  - ... but, can cause runtime exception.
- Example:
  - `Dog` < `Edible` implies `Dog[]` < `Edible[]`.

```
Dog[] dogs = new Dogs[10];
Edible[] edibles = dogs; // Ok, Dog[] < Edible[]
edibles[0] = new Vegetable(); // Ok, Vegetable < Edible
```

  - Trying to put a `Vegetable` into an array of `Dog`'s $\implies$ Runtime exception.
    - Array types are dynamically checked.

Subtyping - Arrays

- Array types: `A` $<$ `B` implies `A[]` $<$ `B[]`.
  - Terminology: Arrays are *covariant*.
    - Subtyping two array types is the same as subtyping the two type arguments.
  - Looks reasonable at first glance.
  - ... but, can cause runtime exception.

Question:
`A` $<$ `B` implies `List<A>` $<$ `List<B>`? E.g. `List<Dog>` $<$ `List<Animal>`.

```
Dog[] dogs = new Dogs[10];
Edible[] edibles = dogs; // Ok, Dog[] < Edible[]
edibles[0] = new Vegetable(); // Ok, Vegetable < Edible
```

- Trying to put a `Vegetable` into an array of `Dog`'s $\Longrightarrow$ Runtime exception.
  - Array types are dynamically checked.

Subtyping - Arrays

- Array types: `A` $<$ `B` implies `A[]` $<$ `B[]`.
  - Terminology: Arrays are *covariant*.
    - Subtyping two array types is the same as subtyping the two type arguments.
  - Looks reasonable at first glance.
  - ... but, can cause runtime exception.

Answer:
No.

```
Dog[] dogs = new Dogs[10];
Edible[] edibles = dogs; // Ok, Dog[] < Edible[]
edibles[0] = new Vegetable(); // Ok, Vegetable < Edible
```

- Trying to put a `Vegetable` into an array of `Dog`'s $\Longrightarrow$ Runtime exception.
  - Array types are dynamically checked.

Subtyping - Generic classes

- Generic classes are *invariant*.
  - `List<A>` $\not<$ `List<B>`, *no matter how A and B are related*.
  - Note, we still have: `ArrayList<A>` $<$ `List<A>`.
- Type erasure makes dynamic checking impossible.
  1. Putting a `Vegetable` into a list of `Dog`'s must succeed.
  2. Retrieving said `Vegetable`, type cast to `Dog` fails.
     - Cast no longer guaranteed to succeed.
- Implications:

```
void eatAll(List<Edible> xs) { .. }

List<Dog> dogs = ... ;

eatAll(dogs); // Error
```

- How to fix: Type parameter bounds.

Generics - Bounded type parameters

- Bounded type parameters introduces covariance and *contravariance* in generics:
- Covariant example (`extends` keyword):

```
<T extends Edible> void eatAll(List<T> xs) { .. }

eatAll(dogs);
eatAll(animals);
eatAll(vegetables);
eatAll(edibles);
```

- Catch: Limits the way we can use `xs` in function body.
  - `T t = xs.get(0);` ☺
  - `xs.add(t);` ☺
  - `Edible e = xs.get(0);` ☺
  - `xs.add(e);` ☹

Generics - Wildcard

- If the type parameter `T` is not used, `eatAll` can be written using *wildcard* type parameter `?`:

```
void eatAll(List<? extends Edible> xs) {
  for(Edible e : xs)
     e.cook();
}
```

- Covariant list of `Edible`'s:
  `Dog` $<$ `Edible` implies
  `List<Dog>` $<$ `List<? extends Edible>`
- Remember: `List<Dog>` $\not<$ `List<Edible>`

Generics - Bounded type parameters

- Contravariant example (`super` keyword):

```
void addADog(Dog dog, List<? super Dog> xs) {
  xs.add(dog);
}

addADog(dogs);
addADog(edibles);
```

- Limitations on `xs`:
  - `Dog d = xs.get(0);` ☹
  - `xs.add(d)` ☺
- Contravariant list of `Dogs`:

  `Dog` $<$ `Edible` implies

  `List<Edible>` $<$ `List<? super Dog>`

Generics - Bounded type parameters

- Stand-alone wildcard:
  - Sometimes we want an unrestricted type parameter that is not used anywhere. E.g.:

```
void printAll(List<?> xs) {
  for(Object x : xs)
    System.out.println(x);
}

printAll(dogs);
printAll(employees);
```

  - `List<Foo>` $<$ `List<?>`.
  - Both covariant and contravariant restrictions apply to the use of `xs`.

Generics - Bounded type parameters

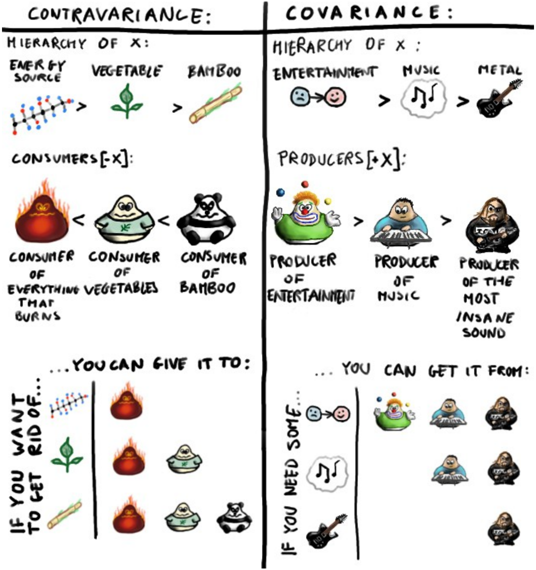
- How co- and contra-variant limitations are enforced.

```
class Box<T> {
  T x;
  T get() { return x; }
  void set(T x) { this.x = x; }
}
```

- Fields and method return types are covariant.
  - If we have a covariant box, `Box<? extends A>`, `get` returns an `A`.
  - If we have a contravariant box, `Box<? super A>`, `get` returns an `Object`.
- Method arguments are contravariant.
  - If we have a covariant box, `Box<? extends A>`, `set` takes only `null`.
  - If we have a contravariant box, `Box<? super A>`, `set` takes an `A`.

Generics - Bounded type parameters

- Bounds in general:
  - When bounding a type in method signature, you
    1. Make the type “bigger”.
    2. Limit how you can use values of the type.
  - Good coding practice: Use “biggest” types possible in your interface.
    - Rule of thumb: Producer extends, consumer super (PECS).



Credits to anoopelias on stackoverflow.com.

Reflection

Reflection - Motivation

- In some languages, it is possible to refer to names or identifiers by using strings. E.g. in PHP the following is legal:

```
$functionName = "foo";
$foo();
```

- This would be nonsense in Java:

```
String x = "foo";
x();
```

- Several reasons for this: Error prone, obscure, no type checking, etc.
- But it has benefits: Execution configuration file.

- Similarly, there is no way of printing the names of the methods of an object in Java.

```
for(String method : Employee.methods)
   System.out.println(method);
```

```
increaseSalary
fire
promote
```

- Benefits: Package explorer, IDE autocompletion, Javadoc compiler, etc.
- Reflection was added to Java to enable these features.

Reflection - Overview

- Basics: Strings instead of names.
    - Inspect classes.
    - Create instances.
    - Change fields.
    - Invoke methods.
- Allows dynamic self-changing behavior.
- Bypass type system.
    - Loose static guarantees.
    - Override visibility.

Reflection - Inspecting classes

- The `Class<T>` object:
  - `Class<String>` represents the class of strings.

```
Class<String> stringClass = String.class;
Class<?> stringClass2 =
     Class.forName("java.lang.String");
```

  - Using a class object:

```
Method[] methods = stringClass.getDeclaredMethods();
Field[] fields = stringClass.getDeclaredFields();
Constructor[] constructors =
     stringClass.getDeclaredConstructors();

Method m = stringClass.getDeclaredMethod("replaceAll",
     stringClass, stringClass);

m.invoke("abc", "a", "x"); // Returns "xbc"
```

Eclipse demonstration:

1. ClassDeclerationSpy from Oracle's Java documentation.
2. Interface proxy.
3. Custom class loader.