# Parameterized methods and parameterized types

Type control for collections

# Parameterized method

The aim of a parameterized method is to specify some constraints on parameter types

Example:
a method that fills elements of a array with the same value:

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

# Problem

The code of fill() could screw up at run-time

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

Why ?

```
public static void main(String[] args) {
  fill(args, 3);
}
```

# Arrays of references

Sub-typing is allowed on arrays.. but is generally **wrong** from typing point of view

```
String[] strings = new String[2];
Object[] objects = strings;  // sub-typing on arrays
                             // compile and run

strings[0] = "blah";
String s0 = (String) objects[0]; // compile and run

objects[1] = new Object();       // compile !!! :-(

String s1 = (String) objects[1]; // compile…
                                 // but dev takes risks!
```

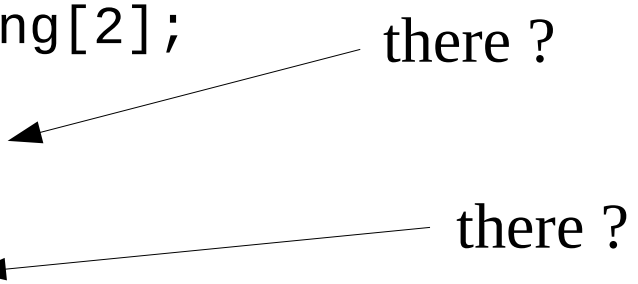
# Arrays of references

Where is the problem ?

```
String[] strings = new String[2];

Object[] objects = strings;

objects[1] = new Object();
```




- Either it is forbidden to allow sub-type on arrays
- Or we must verify at run-time that the reference on the array is of the right class with respect to the class of the reference

# Array of references in Java

- Sub-typing is allowed
- VM checks classes (with an instanceof) at each assignment
  - VM raises an **ArrayStoreException**

```
String[] strings = new String[2];
Object[] objects = strings;     // sub-typing on arrays
objects[1] = new Object();      // compile !!! :-(
   // but raise java.lang.ArrayStoreException at runtime
```

- **It is acceptable if**
  - We read values after sub-typing rather than write them
  - Compiler and VM have access to same information

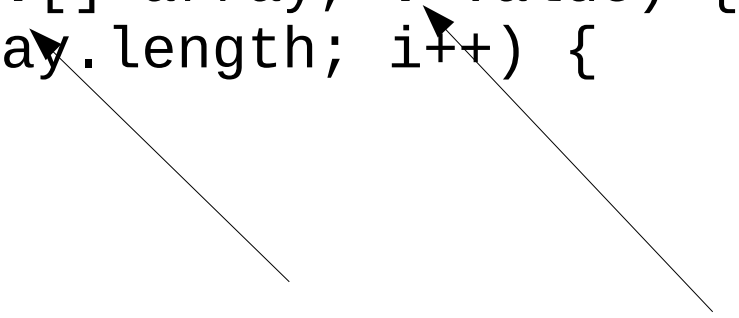
# Solution

Instead of

```
public static void fill(Object[] array, Object value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

we would like that the type of the array and the type of value be the same

```
public static void fill(T[] array, T value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

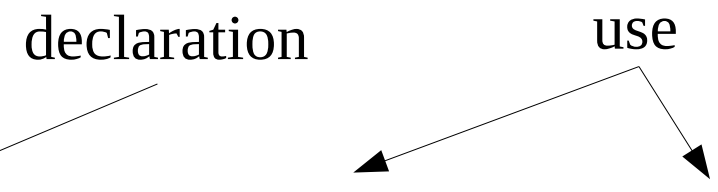


but it does not compile

We note T the common type

# Declaration of type variable

We have to declare a type variable before to use it




```
public static <T> void fill(T[] array, T value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

As any variable, a type variable must be declared

(between modifiers and return type)

# Solution

Since fill() is parameterized

```
public static <T> void fill(T[] array, T value) {
  for(int i = 0; i < array.length; i++) {
    array[i] = value;
  }
}
```

our wrong code doesn’t compile anymore

```
public static void main(String[] args) {
  fill(args, 3);   // compile error !,
                   // fill(String[], Integer)
}
```

# Type variable

Using type variable allows some type constraints to be verified

then, to reject some invalid codes at compile-time, instead of discovering bad surprises at run-time

It also allows type information to be propagated !

# Objects.requireNonNull

```
public class Person {
  private final String name;
  private final Address address;

  public Person(String name, Address address) {
    this.name = Objects.requireNonNull(name);
    this.address = Objects.requireNonNull(address);
  }
}
```

usually, Objects.requireNonNull() must be parameterized !

# Objects.requireNonNull

```
public class Objects {
  public static <T> T requireNonNull(T value) {
    if (value == null) {
      throw new NullPointerException();
    }
    return value;
  }
}
```

But the return type T is those in parameter!

- It can be « inferred » from the argument

# Type inference

In most cases, it is not necessary to specify the type of the argument for a type variable, since the compiler is able to deduce it from the context

- Compiler can use type of the arguments, but also expected type to deduce the return type

# Type inference

Method `Collections.emptyList()` provides an immutable empty list implementation

```
public class Collections {
  public static <E> List<E> emptyList() {
    ...
  }
}
```

# Type inference

```
public class Collections {
  public static <E> List<E> emptyList() { ... }
}
```

- From assignment
  - List<String> list = Collections.emptyList();  // ok
- From return type
  - List<String> foo() {
    return Collections.emptyList();  // ok
    }
- From type of parameter
  - void foo(List<String> list) { ... }
    ...
    foo(Collections.emptyList());   // ok, since Java 8

# Explicit argument type specification

In some cases, compiler cannot infer the type argument (for instance if there are several possibilities)

=> we must explicitely specify it

We give the type argument between the '.' and the method name, surrounded by '<' '>'

Example

```
Object o =Collections.<String>emptyList();
```

# Example

```
public class Foo {
  public void bar(Collection<String> c) { ... }
  public void bar(List<Integer> l) { ... }
}
...
Foo foo = ...
foo.bar(Collections.emptyList());  // compile error
foo.bar(Collections.<String>emptyList()); // ok
foo.bar(Collections.<Integer>emptyList()); // ok
```

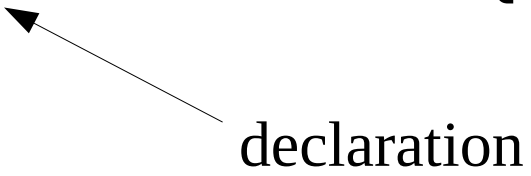
Note:

- Overload is usually not a good idea!

# Parameterized type

It is possible to declare parameterized types

```
public class ArrayList<E> extends ... {
  ...
}
```



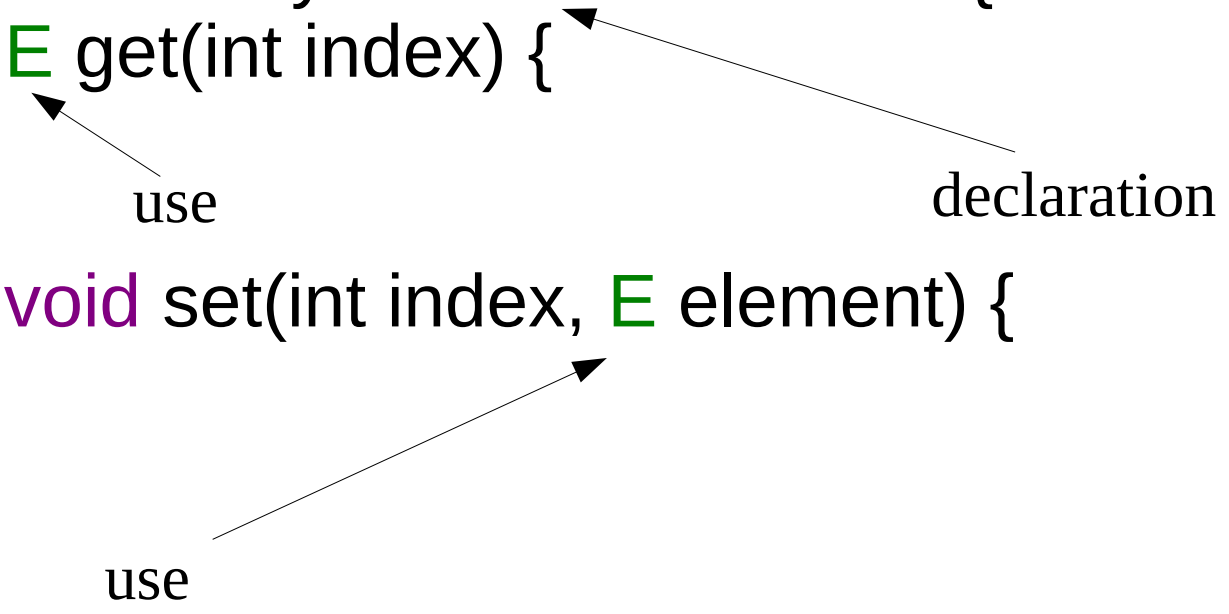

We associate a type argument to an instance of a paramerized type

```
al1 = new ArrayList<String>();  // for al1, E is String
al2 = new ArrayList<Integer>();// for al2, E is Integer
```

# Parameterized type

Allows the compiler to verify and propagate type information

```
public class ArrayList<E> extends ... {
  public E get(int index) {
    ...
  }
  public void set(int index, E element) {
    ...
  }
}
```



# Verify and propagate

```
public class ArrayList<E> extends ... {
  public E get(int index) { ... }
  public void set(int index, E element) { ... }
}
```

- Verify a type

```
ArrayList<String> list = ...
list.set(3, new Object());   // compile error
```

- Propagate a type information

```
ArrayList<String> list = ...
list.get(2).length();          // compile, since list.get(2)
                               // has type String
```

# Type inference <>

Diamond syntax allows type arguments to be inferred like with parameterized methods

```
ArrayList<String> list = new ArrayList<>();
```

works even with several type variables

```
HashMap<String, List<String>> =
                        new HashMap<>();
```

# Static context

A type variable of a parameterized type is not accessible in a static context

- (since it comes with the instance of the parameterized type)

```
public class Box<T> {
  private final T value;

  static void foo(T t) { } // compile error
  static T t;              // compile error
}
```

# Limitations

Implementation of parameterized types in Java is called ***generics***

*Generics* exist for the compiler but **not for the Virtual Machine**

Thus, all operations that require type checking by the VM at run time **cannot work** !

# A single code !

This limitation allows a single code at run-time

- ArrayList<Integer> and ArrayList<String> are the same single code at run-time in ArrayList.class

Avoid problems of C++

Explosion of code !

But how to manage primitive types?

- They are not managed :(

we use boxing/unboxing (but at a cost!)

# Problematic dynamic operations

- `instanceof T` ou `instanceof List<T>`
  - **forbiden**
- `new T(), new T[]`
  - **forbiden**
- `new ArrayList<String>[5]`
  - **forbiden**
    - Since impossible to check `instanceof` for `ArrayStoreException`
- Cast : `(T)` ou `(ArrayList<String>)`
  - Allowed but compiler signal a warning (because no verification possible at run-time)

# Raw Type

Java let you the possibility to use parameterized types without '<' '>', in order to call olb libraries (before jdk 1.5)

- In these cases, compiler signal a warning

Don’t use that with new code !

```
ArrayList list =
        new ArrayList<String>(); // ok, but beurk
ArrayList<String> list =
        new ArrayList(); // warning
```