Generics

Automatically generate parametrised data structures and methods.

```
Parameters: to parameterise a data type,
- included in class definitions,
- in classes and methods,
  use a type parameter instead of a specific data type
```

String Pouch

A Pouch which can hold a String

```
package de.uni_tuebingen.sfs.java2.StringPouch;

public class Pouch {
    private String value;

    public Pouch() {}

    public Pouch( String value ) { this.value = value; }

    public void set( String value ) { this.value = value; }

    public String get() { return value; }

    public boolean isEmpty() { return value == null; }

    public void empty() { value = null; }

}
```

More Pouches

To get Pouches for different data types we could duplicate our code and replace the content data type.

Ok. But when we add a feature we have to update all different Pouches. We might forget something….

Objects can refer to instances of all classes. Swap String with Object and we have a Pouch which works for all classes. New Pouch features are added easily.

Object Pouch

A Pouch which can hold an Object

```java
package de.uni_tuebingen.sfs.java2.ObjectPouch;

public class Pouch {
    private Object value;

    public Pouch() {}

    public Pouch( Object value ) { this.value = value; }

    public void set( Object value ) { this.value = value; }

    public Object get() { return value; }

    public boolean isEmpty() { return value == null; }

    public void empty() { value = null; }

}
```

☆

Use ObjectPouch

```
package de.uni_tuebingen.sfs.java2;

import de.uni_tuebingen.sfs.java2.ObjectPouch.Pouch;

public class ObjectPouchMain {
    public static void main(String[] args) {
        Pouch pouch = new Pouch(Integer.valueOf("12"));
        Pouch stringPouch = new Pouch("Umu");
        //Integer intValue = pouch.get();
        System.out.println("Pouch value: "+pouch.get());
        System.out.println("Pouch value: "+stringPouch.get());
    }
}
```

Whats nice is, we can create Pouches for different types. But we cannot refer to the original type of our Pouch data. The statement `Integer intValue = pouch.get();` does not compile. This is obviously not the perfect solution

因为 pouch 的数据类型是 Object，Object 可以包含多种数据类型。我们创建了一个值为整数 12 的 Pouch 对象，又创建了一个值为字符串 "umu" 的 Pouch 对象。因此，当 Integer intValue = pouch.get(); 时，没有指定 Object 中具体的数据类型，编译器无法进行类型转换，所以无法编译通过。(No explicit (type) casting)

Generic Pouch

What we want is:

Type safety. When we add an Integer we want to get an Integer back.

Flexibility. Update code in one place. All different datatypes share the same code base

```
public class Pouch<T> {
    private T value;

    public Pouch() {}

    public Pouch( T value ) { this.value = value; }

    public void set( T value ) { this.value = value; }

    public T get() { return value; }

    public boolean isEmpty() { return value != null; }

    public void empty() { value = null; }

}
```

Type of a Generic

When we declare a generic class we add <T> after the classname.

T stands for type. But it can also be <K> for key or <E> for element. The name of the character does not matter. K,E,T does not matter. Be consistent.

T specifically stands for **generic** type. According to Java Docs - A **generic** type is a **generic** class or interface that is parameterized over types.

When we create an Instance the <T> is replace with the actual data type.

When we declare `Pouch<String>` the T in <T> becomes `String`.

Using a generic Pouch

```
package de.uni_tuebingen.sfs.java2;

import de.uni_tuebingen.sfs.java2.GenericPouch.Pouch;

public class GenericPouchMain {
    public static void main(String[] args) {
        // Pouch which holds a String
        Pouch<String> stringPouch = new Pouch<>("Umu");
        // Pouch which holds an Integer
        Pouch<Integer> integerPouch = new Pouch<>(Integer.valueOf("12"));
        //Pouch which holds a Pouch which holds a String
        Pouch<Pouch<String>> pouchPouch = new Pouch<>(new Pouch<>("Fasel"));
        System.out.println("Pouch value: "+stringPouch.get());
        System.out.println("Pouch value: "+integerPouch.get());
        System.out.println("Pouch value: "+pouchPouch.get());
    }
}
```

Experiment with the code. Try to add different types. Add integer to String Pouch. See if you actually get an Integer from the integerPouch…..

Generic and interfaces

You can use generics the same way as we did it with classes:

```
public interface Set<E> extends Collection<E>
{
 …
}
```

```
public class HashSet<E> extends AbstractSet<E>
                        implements Set<E>,
                        Cloneable,
                        java.io.Serializable
{
 …
}
```