Generics

Problem

Class Box

```
public class Box {
    private Object item;

    public Object getItem() {
        return item;
    }

    public void setItem(Object item) {
        this.item = item;
    }
}
```

Class Box

```
Box box = new Box();
box.setItem("Test");
// code
Object item = box.getItem();
Integer itemInt = (Integer) item;
```

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

Generics

Identificator: Number vs. String

```
public class Account {
    private Object id;
    private int sum;

    public Account(Object id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public Object getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Identificator: `Number` vs. `String`

```
public class Program {
    public static void main(String[] args) {
        Account acc1 = new Account(2334, 5000);
        int acc1Id = (int) acc1.getId();
        System.out.println(acc1Id);

        Account acc2 = new Account("sid5523", 5000);
        System.out.println(acc2.getId());
    }
}
```

Generics (Обобщения)

```
public class Account<T> {
    private T id;
    private int sum;

    public Account(T id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public T getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Generics (Обобщения)

```
public class Program {
    public static void main(String[] args) {
        Account<String> acc1 = new Account<>("2345", 5000);
        String acc1Id = acc1.getId();
        System.out.println(acc1Id);

        Account<Integer> acc2 = new Account<>(2345, 5000);
        Integer acc2Id = acc2.getId();
        System.out.println(acc2Id);
    }
}
```

Generics and Interfaces

```
public interface Accountable<T> {
    T getId();

    int getSum();

    void setSum(int sum);
}
```

Generics and Interfaces

```
public class Account implements Accountable<String> {
    private String id;
    private int sum;

    public Account(String id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public String getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Generics and Interfaces

```
public class Program {
    public static void main(String[] args) {
        Accountable<String> acc1 = new Account("1235rwr", 5000)
        Account acc2 = new Account("2373", 4300);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
    }
}
```

Generics and methods

```
public class Printer {
    public <T> void print(T[] items) {
        for (T item : items) {
            System.out.println(item);
        }
    }
}
```

Generics and methods

```
public class Program {
    public static void main(String[] args) {
        Printer printer = new Printer();
        String[] people = {"Tom", "Alice", "Sam", "Kate", "Bob"
        Integer[] numbers = {23, 4, 5, 2, 13, 456, 4};
        printer.<String>print(people);
        printer.<Integer>print(numbers);
    }
}
```

Generics and constructors

```
public class Account {
    private String id;
    private int sum;

    public <T> Account(T id, int sum) {
        this.id = id.toString();
        this.sum = sum;
    }

    public String getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Generics and constructors

```
public class Program {
    public static void main(String[] args) {
        Account acc1 = new Account("cid2373", 5000);
        Account acc2 = new Account(53757, 4000);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
    }
}
```

Multiple Generics

Multiple Generics

```
public class Account<T, S> {
    private T id;
    private S sum;

    public Account(T id, S sum) {
        this.id = id;
        this.sum = sum;
    }

    public T getId() {
        return id;
    }

    public S getSum() {
        return sum;
    }

    public void setSum(S sum) {
        this.sum = sum;
    }
```

Multiple Generics

```
public class Program {
    public static void main(String[] args) {
        Account<String, Double> acc1 = new Account<>("354", 500
        String id = acc1.getId();
        Double sum = acc1.getSum();
        System.out.printf("Id: %s  Sum: %f \n", id, sum);
    }
}
```

Limitations of Generics

Limitation with superclass

```
public class Account {
    private String id;
    private int sum;

    public Account(String id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public String getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Limitation with superclass

```
public class Transaction<T extends Account> {
    private T from;
    private T to;
    private int sum;

    public Transaction(T from, T to, int sum) {
        this.from = from;
        this.to = to;
        this.sum = sum;
    }

    public void execute() {
        if (from.getSum() > sum) {
            from.setSum(from.getSum() - sum);
            to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %
                from.getId(), from.getSum(), to.getId(), to.g
        } else {
            System.out.printf("Operation is invalid");
```

Limitation with superclass

```
public class Program {
    public static void main(String[] args) {
        Account acc1 = new Account("1876", 4500);
        Account acc2 = new Account("3476", 1500);

        Transaction<Account> tran1 =
                new Transaction<Account>(acc1, acc2, 4000);
        tran1.execute();
        tran1 = new Transaction<Account>(acc1, acc2, 4000);
        tran1.execute();
    }
}
```

Limitation with Generic Types

```
public class Account<T> {
    private T id;
    private int sum;

    public Account(T id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public T getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Limitation with Generic Types

```
public class Transaction<T extends Account<String>> {
    private T from;
    private T to;
    private int sum;

    public Transaction(T from, T to, int sum) {
        this.from = from;
        this.to = to;
        this.sum = sum;
    }

    public void execute() {
        if (from.getSum() > sum) {
            from.setSum(from.getSum() - sum);
            to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %
                from.getId(), from.getSum(), to.getId(), to.g
        } else {
            System.out.printf("Operation is invalid");
```

Limitation with Generic Types

```
public class Program {
    public static void main(String[] args) {
        Account<String> acc1 = new Account<String>("1876", 4500
        Account<String> acc2 = new Account<String>("3476", 1500

        Transaction<Account<String>> tran1 =
                new Transaction<Account<String>>(acc1, acc2, 40
        tran1.execute();
        tran1 = new Transaction<Account<String>>(acc1, acc2, 40
        tran1.execute();
    }
}
```

Limitation with Interfaces

```
public interface Accountable {
    String getId();

    int getSum();

    void setSum(int sum);
}
```

Limitation with Interfaces

```
public class Account implements Accountable {
    private String id;
    private int sum;

    public Account(String id, int sum) {
        this.id = id;
        this.sum = sum;
    }

    public String getId() {
        return id;
    }

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
```

Limitation with Interfaces

```
public class Transaction<T extends Accountable> {
    private T from;
    private T to;
    private int sum;

    public Transaction(T from, T to, int sum) {
        this.from = from;
        this.to = to;
        this.sum = sum;
    }

    public void execute() {
        if (from.getSum() > sum) {
            from.setSum(from.getSum() - sum);
            to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %
                    from.getId(), from.getSum(), to.getId(),
        } else {
            System.out.printf("Operation is invalid");
```

Limitation with Interfaces

```
public class Program {
    public static void main(String[] args) {
        Account acc1 = new Account("1235rwr", 5000);
        Account acc2 = new Account("2373", 4300);
        Transaction<Account> tran1 =
                new Transaction<Account>(acc1, acc2, 1560);
        tran1.execute();
    }
}
```

Multiple Limitations

Multiple Limitations

```
public class Person {
}

public interface Accountable {
}

public class Transaction<T extends Person & Accountable> {
}
```

Inheritance and Generics

Basic generic class

```
public class Account<T> {
    private T id;

    public Account(T id) {
        this.id = id;
    }

    public T getId() {
        return this.id;
    }
}
```

Basic generic class

```
public class DepositAccount<T> extends Account<T> {
    public DepositAccount(T id) {
        super(id);
    }
}
```

Basic generic class

```
public class Program {
    public static void main(String[] args) {
        DepositAccount dAccount1 = new DepositAccount(20);
        System.out.println(dAccount1.getId());

        DepositAccount dAccount2 = new DepositAccount("12345");
        System.out.println(dAccount2.getId());
    }
}
```

Basic generic class

```
public class Account<T> {
    private T id;

    public Account(T id) {
        this.id = id;
    }

    public T getId() {
        return this.id;
    }
}
```

Basic generic class

```
public class DepositAccount<T, S> extends Account<T> {
    private S name;

    public S getName() {
        return this.name;
    }

    public DepositAccount(T id, S name) {
        super(id);
        this.name = name;
    }
}
```

Basic generic class

```
public class Program {
    public static void main(String[] args) {
        DepositAccount<Integer, String> dAccount1 =
                new DepositAccount(20, "Tom");
        System.out.println(dAccount1.getId() + " : " + dAccount

        DepositAccount<String, Integer> dAccount2 =
                new DepositAccount("12345", 23456);
        System.out.println(dAccount2.getId() + " : " + dAccount
    }
}
```

Generic subclass

Generic subclass

```
public class Account {
    private String name;

    public Account(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

Generic subclass

```
public class DepositAccount<T> extends Account {
    private T id;

    public DepositAccount(String name, T id) {
        super(name);
        this.id = id;
    }

    public T getId() {
        return this.id;
    }
}
```

Generic Type Converion

```
public class Account<T> {
    private T id;

    public Account(T id) {
        this.id = id;
    }

    public T getId() {
        return this.id;
    }
}
```

Generic Type Converion

```
public class DepositAccount<T> extends Account<T> {
    public DepositAccount(T id) {
        super(id);
    }
}
```

Generic Type Converion

```
public class Program {
    public static void main(String[] args) {
        DepositAccount<Integer> depAccount = new DepositAccount
        Account<Integer> accountInteger = depAccount;
        System.out.println(accountInteger.getId());

        Account<String> accountString = depAccount; // compile
        Account<String> accountString2 = (Account<String>) depA
    }
}
```

Wildcard

Wildcard

Возникает необходимость в метод обобщенного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом.

Wildcard

В этом случае при определении метода следует применить **wildcard** ?.

```
public class Generic<T> {
    // code

    boolean compare(Generic<?> o) {
        return o.getObject() == obj;
    }
}
```

Wildcard

Wildcard также может использоваться с ограничением `extends` для передаваемого типа: `<? extends Number>`

Total

- Операции, для выполнения которых нужно точно знать типы в *runtime*, работать не будут:
  - **Приведение типов**: `(T) var;`
  - **instanceof**: `var instaceof T;`
  - **new**: `T var = new T(); T[] array = new T[size]`
  - **Создание массива конкретного типа**: `Type<Integer> arr = new Type<Integer>[10];`