**After java 1.5**

**Autoboxing** is the automatic conversion that the **Java** compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called **unboxing**.

import java.util.ArrayList;

import java.util.List;

public class Unboxing {

public static void main(String[] args) {

Integer i = new Integer(-8);

// 1. Unboxing through method invocation

int absVal = absoluteValue(i);

System.out.println("absolute value of " + i + " = " + absVal);

List<Double> ld = new ArrayList<>();

ld.add(3.1416); // Π is autoboxed through method invocation.

// 2. Unboxing through assignment

double pi = ld.get(0);

System.out.println("pi = " + pi);

}

public static int absoluteValue(int i) {

return (i < 0) ? -i : i;

}

}

List<**int**> a =**new** ArrayList<**int**>(); // not Allowed (only objects are allowed)

List<Integer> a =**new** ArrayList<Integer>();

Need of Primitive Type Functional Interfaces

**public** **class** ExampleOne {

**public** **static** **void** main(String[] args) {

**int** num[] = { 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };

Predicate<Integer> p = i -> (i % 2 == 0);

**for** (**int** i = 0; i < num.length; i++) {

**if** (p.test(i)) {

System.***out***.print(num[i] + " ");

}

}

}

}

Performance wise above is not recommended, because num array is primitive, but predicate accepts Integer (and there is performance hit in autoboxing and unboxing.

The issue is same for Function, Consumer, Supplier etc

To solve above issue, primitive type functional interfaces are introduced

@FunctionalInterface

**public** **interface** IntPredicate {

**boolean** test(**int** value);

}

@FunctionalInterface

**public** **interface** Predicate<T> {

**boolean** test(T t);

}

**import** java.util.function.IntPredicate;

**public** **class** ExampleTwo {

**public** **static** **void** main(String[] args) {

**int** num[] = { 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };

IntPredicate p = (k) -> k % 2 == 0;

**for** (**int** i = 0; i < num.length; i++) {

**if** (p.test(i)) {

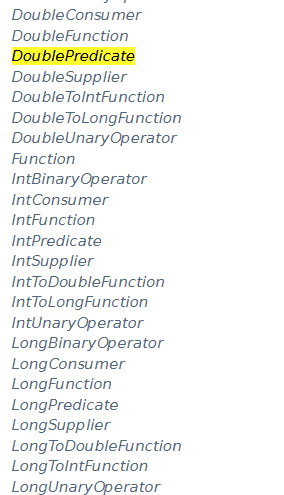
System.***out***.print(num[i] + " ");

}

}

}

}



java.util.function.ToIntFunction<String> strLength = (s) -> s.length();

System.***out***.println(strLength.applyAsInt("Hello World!"));

java.util.function.IntToDoubleFunction sqrtFunc = (s) -> Math.*sqrt*(s);

System.***out***.println(sqrtFunc.applyAsDouble(18));

**Primitive Type Functional Interfaces for Function:**

**The following are various primitive Type Functional Interfaces for Function**

**1. IntFunction: can take int type as input and return any type public R apply(int i);**

**2. LongFunction: can take long type as input and return any type public R apply(long i);**

**3. DoubleFunction: can take double type as input and return any type public R apply(double d);**

**4. ToIntFunction: It can take any type as input but always returns int type public int applyAsInt(T t)**

**5. ToLongFunction: It can take any type as input but always returns long type public long applyAsLong(T t)**

**6. ToDoubleFunction: It can take any type as input but always returns double type public int applyAsDouble(T t)**

**7. IntToLongFunction: It can take int type as input and returns long type public long applyAsLong(int i)**

**4**

**https://**[**www.youtube.com/durgasoftware**](http://www.youtube.com/durgasoftware)

**Java 8 New**

**Features in**

**Simple Way**

**8. IntToDoubleFunction: It can take int type as input and returns long type**

**public double applyAsDouble(int i)**

**9. LongToIntFunction: It can take long type as input and returns int type public int applyAsInt(long i)**

**10. LongToDoubleFunction: It can take long type as input and returns double type public int applyAsDouble(long i)**

**11. DoubleToIntFunction: It can take double type as input and returns int type public int applyAsInt(double i)**

**12. DoubleToLongFunction: It can take double type as input and returns long type public int applyAsLong(double i)**

**13. ToIntBiFunction:return type must be int type but inputs can be anytype public int applyAsInt(T t, U u)**

**14. ToLongBiFunction:return type must be long type but inputs can be anytype public long applyAsLong(T t, U u)**

**15. ToDoubleBiFunction: return type must be double type but inputs can be anytype**

**public double applyAsDouble(T t, U u)**

**Primitive Version for Consumer:**

**The following 6 primitive versions available for Consumer:**

**1. IntConsumer**

**public void accept(int value)**

**2. LongConsumer**

**public void accept(long value)**

**3. DoubleConsumer**

**public void accept(double value)**

**4. ObjIntConsumer<T>**

**public void accept(T t,int value)**

**5. ObjLongConsumer<T>**

**public void accept(T t,long value)**

**6. ObjDoubleConsumer<T>**

**public void accept(T t,double value)**

**Primitive Versions for Supplier:**

**The following 4 primitive versions available for Supplier:**

**1. IntSupplier**

**public int getAsInt();**

**2. LongSupplier**

**public long getAsLong()**

**3. DoubleSupplier**

**public double getAsDouble()**

**4. BooleanSupplier**

**public boolean getAsBoolean()**

**UnaryOperator<T>:**

⚽ **If input and output are same type then we should go for UnaryOperator**

⚽ **It is child of Function<T,T>**

Function<Integer, Integer> f = i -> i \* i;

System.***out***.println(f.apply(5));

UnaryOperator<Integer> f2=i->i\*i;

System.***out***.println(f2.apply(6));

**The primitive versions for UnaryOperator:**

**IntUnaryOperator:**

**public int applyAsInt(int)**

**LongUnaryOperator:**

**public long applyAsLong(long)**

**DoubleUnaryOperator:**

**public double applyAsDouble(double)**

**BinaryOperator:**

**It is the child of BiFunction<T,T,T>**

**The primitive versions for BinaryOperator:**

**1. IntBinaryOperator**

**public int applyAsInt(int i,int j)**

**2. LongBinaryOperator**

**public long applyAsLong(long l1,long l2)**

**3. DoubleBinaryOperator**

**public double applyAsLong(double d1,double d2)**

BiFunction<String,String,String> f3=(s1,s2)->s1+s2;

System.***out***.println(f3.apply("Hello","World!"));