Java SE 7

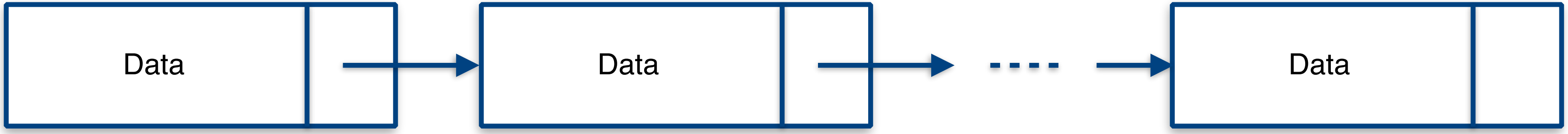
Module 4
Generics

LUXOFT TRAINING

LXFT
LISTED
NYSE

What if you need a container with **dynamic** size?

List



List

generics.c0list

```
public class Node
{
    private Node next;
    private final Object data;

    public Node(Object data) {
        this.data = data;
    }

    public Object getData() {
        return data;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }
}
```

java.util.List

```
public interface List<E> extends Collection<E> {

    int size();

    boolean isEmpty();

    boolean contains(Object o);

    boolean add(E e);

    boolean remove(Object o);

    void clear();
}
```

java.util.List some of the implementations

- ArrayList
- LinkedList
- Vector

Generics

Before Java 5 – Object used as a universal class

```
public class ArrayList
{
    private Object[] elementData;

    public Object get(int i) { .. }

    public void add(Object o) { .. }
}
```

Generics

Using Object you may get into problems

> ClassCastException
> type casting

```java
List array = new ArrayList();

array.add(10);
array.add("Str");

for (Object o : array)
{
    Integer number = (Integer) o;
}
```

Generics

Generic is simple

> check errors on compilation stage
> no type casting

```
List<Integer> array = new ArrayList<>();

array.add(10);

for (Integer i : array)
{
    Integer number = i + 10;
}
```

Generics

generics.os.Basket

Generic class example

```
public class Basket<T>
{
    List<T> products;

    public Basket() { .. }

    public void add(T p) { .. }

    public T remove(T p) { .. }

    public List<T> getProducts() { .. }
}
```

Generics

generics.os.UsingBasket

Use of generic class

```
Basket<Product> basket = new Basket<>();

basket.add(new Axe(1L, 2.5));
basket.add(new Monitor(2L, 17));

double totalPrice = 0;

for (Product p : basket.getProducts())
{
    totalPrice += p.getPrice();
}
```

Generics

Runtime type erasing

```
public class Basket
{
    List<Object> products;

    public Basket() { .. }

    public void add(Object p) { .. }

    public Object remove(Object p) { .. }

    public List<Object> getProducts() { .. }
}
```

Generics

Restricting of T: `public class Basket<T extends Product>`

```
// After erasing with the restriction
public class Basket
{
    List<Product> products;

    public Basket() { .. }

    public void add(Product p) { .. }

    public Product remove(Product p) { .. }

    public List<Product> getProducts() { .. }
}
```

Two implementations after type erasure

```
public class Basket<T>
{
    public boolean equals(T obj)
    {
        return super.equals(obj);
    }
}
```

will not compile
both methods have same erasure

Will be 2 implementations:

```
boolean equals(String) // defined in Basket<T>
boolean equals(Object) // inherited from Object
```

But on erasing we get T -> Object.
2 same methods? Disallowed.

Generics Restrictions

- **work with primitive types**

```
Basket<int> // will not compile
```

- **get type at execution time**

```
a instanceof Basket<Integer>
```

- **generic type cannot extend Throwable**

```
class Problem<T> extends Exception
```

- **cannot be used in catch**

```
catch(T t) // will not compile
```

Generics Restrictions

- Generic type **instance cannot be created**

```
static class Primitive<T>
{
    void create()
    {
        T t = new T(); // will not compile
    }
}
```

`Class<T> can be used for that:`

```
public class Basket<T>
{
    public Basket<T> makeBasket(Class<T> clazz) throws
Exception {
        return (Basket<T>) clazz.newInstance();
    }}
```

Generic Method

You can generify only specific method(s) of the class.

```
public <K, V> put(K key, V value) { }
```

Generic Method

```
class ArrayAlg
{
    public static <T> T getMiddle(T[] arr)
    {
        return arr[arr.length / 2];
    }
}
```

Usage example:

```
String[] names = {"John", "Q.", "Public"};
String middle = ArrayAlg.getMiddle(names);
```

Generic Method

Restricting: T should extend Comparable

```
public static <T> T min(T[] arr)
{
    T smallest = null;

    if (arr != null || arr.length > 0)
    {
        smallest = arr[0];
        for (T currentElement : arr)
        {
            if (smallest.compareTo(currentElement) > 0)
            {
                smallest = currentElement;
            }
        }
    }
    return smallest;
}
```

The method **compareTo(T)** is undefined for the type **T**

Generic Method

Restricting: T extends Comparable

```
public static <T extends Comparable> T min(T[] arr)
{
    T smallest = null;

    if (arr != null || arr.length > 0)
    {
        smallest = arr[0];
        for (T currentElement : arr)
        {
            if (smallest.compareTo(currentElement) > 0)
            {
                smallest = currentElement;
            }
        }
    }
    return smallest;
}
```

Generic Method

Limitations:

```
public static <T extends Comparable & Serializable> T
    send(T[] arr) { .. }
```

Generics Restrictions Static

- You cannot use generics in static context

```
public class StaticRestrictions<T>
{
    // will not compile
    private static T instance;

    // will not compile
    public static T getInstance()
    {
        return null;
    }
}
```

Generics Restrictions Static

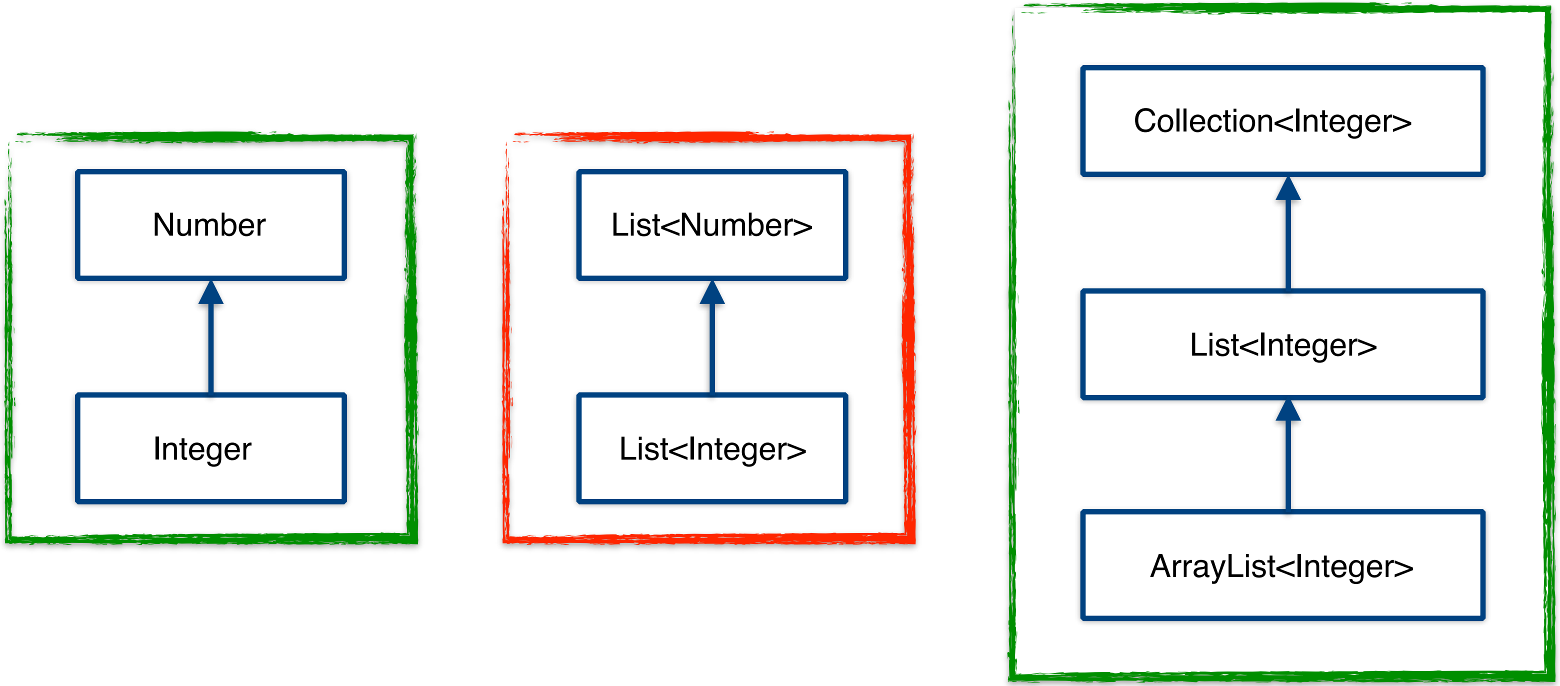
If **static** fields of type parameters were allowed, then the following code would be confusing:

```
    Singleton<Bank> bank = new Singleton<>();
    Singleton<Client> client = new Singleton<>();
    Singleton<Account> account = new Singleton<>();
```

Because the static field is shared by bank, client, and account, what is the actual type? It cannot be Bank, Client and Account at the same time.

You cannot, therefore, create **static** fields of type parameters.

Wildcards



Why this will not work?

generics.g1list.D5

```
List<Number> numbers = new ArrayList<>();

numbers.add(new Long(34L));

List<Integer> ints = numbers; // will not compile

Integer i = ints.get(0);
```

Because of unavoidable exception here!

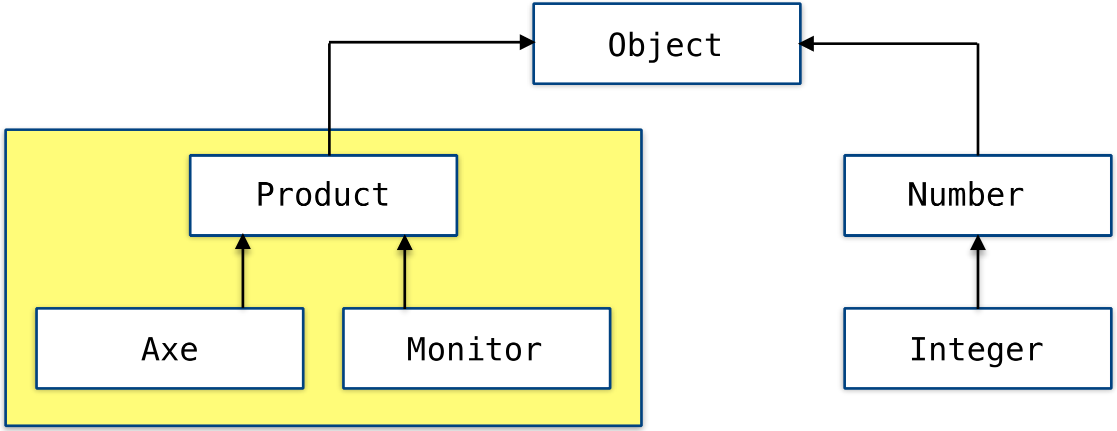
Old Style, no Wildcard

```
Basket basket = new Basket();

basket.add(new Monitor(counter++, 17));
basket.add(new Monitor(counter++, 21));
basket.add(new Axe(counter++, 2.5));

Monitor monitor = (Monitor) basket.get(2);
Axe axe = (Axe) basket.get(3);
```

Nothing have changed, still should use casting here!

Wildcards

```
public <T extends Product> T getNeededProduct(long uid)
```



Wildcards

You can also use **interfaces** separated with **&** to narrow down **generalized type** even more.

```
public <Type extends A & B & C & D> Type methodName(..) { .. }
```

Example:

```
public <T extends Product & Serializable & Comparable> T
    getNeededProduct(long uid)
```

Method Wildcard

```
public class Basket
{
    public <T extends Product> T getNeededProduct(long uid)
    {
        …
    }
}
```

Now you can get need object **with no casting**!

```
Product product = basket.getNeededProduct(1);

Axe axe = basket.getNeededProduct(3);

Grill grill = basket.getNeededProduct(4);
```

No code cange needed when you add new Product types!

Generic Method

We need a method that can print any list of objects that instanceof Product.

```
void printProductsOnly(List<Product> products)

void printAxesNotMonitorsOrProducts(List<Axe> axes)
```

But what about `List<Monitor>` or `List<`**`TypeThatWillBeAddedSoon`**`>` ?

Generic Method

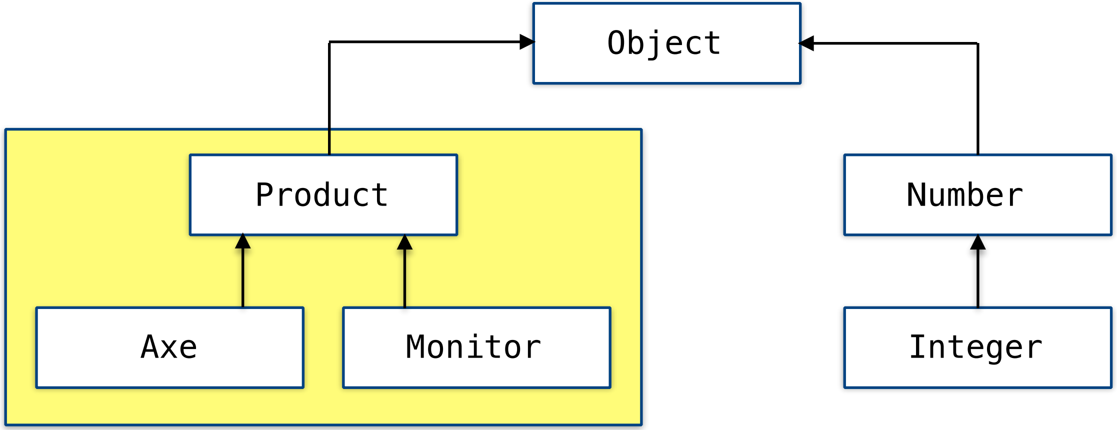
generics.g6methods.D2, D3

```
Product find(List<? extends Product> products, Product p)

find(products, monitor);
find(products, axe);

find(monitors, monitor);
find(monitors, axe); // weird isn’t it
```

Can we fix this with **Generics**?

Wildcards

```
public class Basket<T extends Product>
{
    List<T> products;
}
```



Wildcards

You can also use **interfaces** separated with **&** to narrow down **generalized type** even more.

```
public class ClassName<Type extends A & B & C & D>
```

Examples:

```
public class Basket<T extends Product & Serializable & Comparable>

public class TreeMap<K,V> extends AbstractMap<K,V>
```

Class Wildcard

```
public class Basket<T extends Product>
{
    List<T> products;

    public void add(T p) { … }
}
```

If you need **Basket** of **Axe** **only**!

```
Basket<Axe> basket = new Basket();

basket.add(new Axe(counter++, 2.5));

basket.add(new Monitor(counter++, 17)); // will not compile
```

Generic Method

generics.g7inout

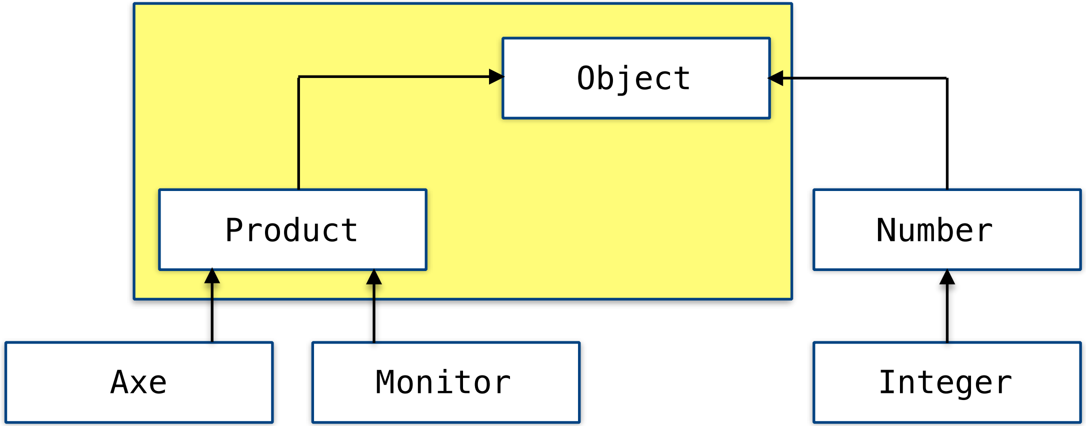
```
public void copy(List src, List dest) { }

copy(axes, monitors);  // weird isn't it?
copy(products, axes);  // weird isn't it?
```

Can we fix this with **Generics**?

Wildcards, super

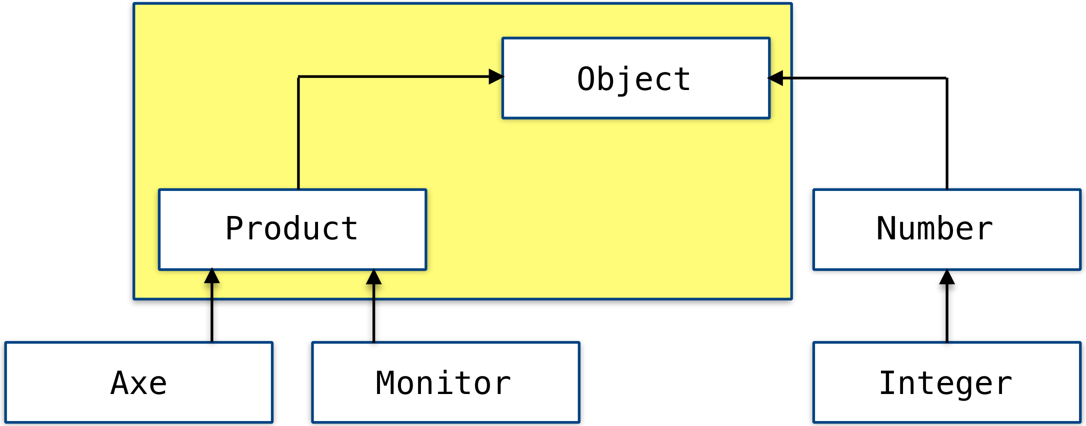
`<T extends Product> – at least Product`

`<T super Product> –` **`Product or higher in object hierarchy`**



Wildcards, super

```
void copy(List<? extends Product> src, List<? super Product> dest)
{
   dest.addAll(src);
}
```



- PECS principle: "Producer Extends, Consumer Super"

- The Get and Put Principle: use an extends wildcard when you only get values out of a structure, use super wildcard when you only put values into a structure, and don't use a wildcard when you both get and put.

Recursive Type Bound

generics.g8recursivewc

Advanced topic. Warning!

```
public abstract
    class Product<T extends Product<T>>
        implements Comparable<T>
{
   @Override
    public int compareTo(T o)
    {
        return subCompare(o);
    }

    public abstract int subCompare(T o);
}
```

Generics

- Exersice 1 - Bank Application.