



Object Oriented Programming with Java

PART 1: JAVA PROGRAMMING BASICS

Generics, By Aphrodice Rwagaju



Java's collection classes

- The JDK contains many useful classes to help you store collections of objects without writing your own storage classes
 - ArrayList
 - HashMap
 - LinkedList
- We can use these in our programs by importing the package `java.util`

```
import java.util.*;  
  
ArrayList list = new ArrayList();
```

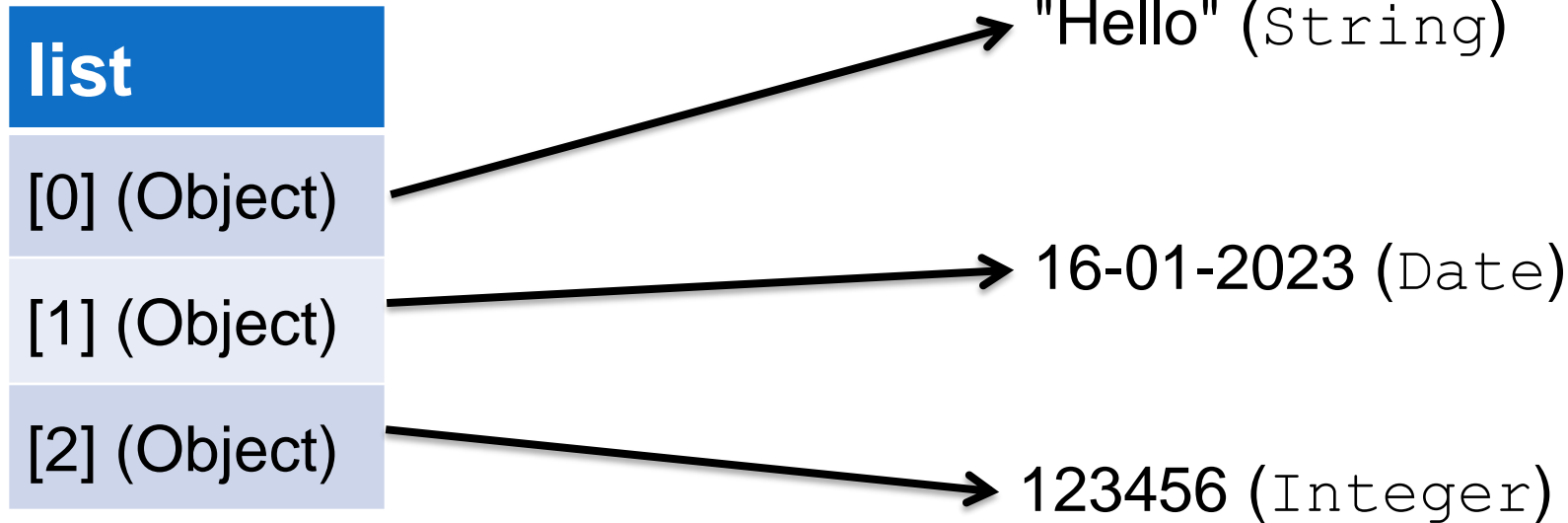
Without generics

- These classes were developed to store any kind of Java object
- Internally they use references of type `Object` to store each element
- Thus they were not strongly-typed, you had to remember what kind of object you were storing

```
ArrayList list = new ArrayList();  
list.add("Hello");  
String s = (String)list.get(0);
```

ArrayList Example

```
ArrayList list = new ArrayList();  
list.add("Hello");  
list.add(new Date());  
list.add(123456);
```



ArrayList Example

```
ArrayList list = new ArrayList();  
list.add("Hello");  
list.add(new Date());  
list.add(123456);
```

```
String s = (String)list.get(0);  
Date d = (Date)list.get(1);  
int n = (Integer)list.get(2);
```

```
String e = (String)list.get(2);
```

Runtime
Error!

Behold generics

- So sometimes it can be useful to not be strongly typed – it means you can store anything
- But its easy to forget what you've stored, and end up with an `ClassCastException`
- Better to have some control over what goes into your `ArrayList` (or other collection class)

Behold generics

- The new version of these classes support something called generics, which means we can specify a type for our collection
- For example, an `ArrayList` that only accepts strings...

```
ArrayList<String> list = new ArrayList<String>();  
list.add("Hello");  
list.add("World");  
list.add(123456);
```

Compiler
error

How does it work?

- Supposing we have the following simple class

```
class Point {  
    protected int x, y;  
}
```

- But we want different versions for other data types....

```
class DoublePoint {  
    protected double x, y;  
}
```

```
class ShortPoint {  
    protected short x, y;  
}
```


Going Generic

- We can make our Point class generic...

```
class Point<T> {  
    protected T x, y;  
}
```

- And then specify the data type when we create an instance...

```
Point<Double> p1 = new Point<Double>();  
Point<Short> p2 = new Point<Short>();
```

`p1.x` and `p1.y` are now of type `Double`

Methods of generic classes

- We can define generic methods to work with our generic variables...

```
class Point<T> {  
    protected T x, y;  
  
    public T getX() {  
        return x;  
    }  
    public void setX(T x) {  
        this.x = x;  
    }  
}
```

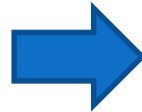
T is the **type variable**

We can call it anything,
uppercase T is just
a convention

Just like a template

- You can think of a generic class as a template for creating new more specific classes

```
class Point<T> {  
    protected T x, y;  
  
    public T getX() {  
        return x;  
    }  
}
```



```
class Point<Double> {  
    protected Double x, y;  
  
    public Double getX() {  
        return x;  
    }  
}
```

```
Point<Double>d=new Point<Double>();
```

Multiple Type Variables

- You are not limited to just one type variable...

```
class PolarPoint<T, R> {  
    protected T angle;  
    protected R radius;  
  
    public T getAngle() { return angle; }  
    public R getRadius() { return radius; }  
}
```

```
PolarPoint<Double, Integer> p = new PolarPoint<Double, Integer>();
```

```
PolarPoint<Float, Integer> p2 = new PolarPoint<Float, Integer>();
```

Generic methods

- Any class can have generic methods...

```
class EqualityTest {  
    public static <T> boolean test(T o1, T o2) {  
        return o1.equals(o2);  
    }  
}
```

Note that type variable comes just before the return type

Generic methods

- The compiler tries to infer the type variable from the parameters...

```
EqualityTest.test("Same", "Same");
```

- ... but we can also explicitly declare the type variable as follows...

```
EqualityTest.<String>test("Same", "Same");
```

Generic types as parameters

- Supposing we have a function which expects a `ArrayList<Integer>` as its only parameter...

```
public void func(ArrayList<Integer> list) { ... }
```

```
ArrayList<Integer> list1 = new ArrayList<Integer>();  
ArrayList<String> list2 = new ArrayList<String>();
```

```
func(list1);
```

```
func(list2);
```

Compiler
error

Wildcard type parameters

- If we want our function to accept an `ArrayList` with any type parameter, we can use the **?** wildcard

```
public void func(ArrayList<?> list) { ... }
```

```
ArrayList<Integer> list1 = new ArrayList<Integer>();  
ArrayList<String> list2 = new ArrayList<String>();
```

```
func(list1);    // No problem  
func(list2);    // No problem
```


Bounded type parameters

- We can limit the accepted types using the **extends** keyword...

```
public void func(ArrayList<? extends Number> list) {}
```

```
ArrayList<Integer> list1 = new ArrayList<Integer>();  
ArrayList<Double> list2 = new ArrayList<Double>();  
ArrayList<String> list3 = new ArrayList<String>();
```

```
func(list1);
```

```
func(list2);
```

```
func(list3);
```

Compiler error

Bounded types

- We can also limit which types are used with our generic classes and methods...
- **extends** is used to limit types to subclasses of the specified class...

```
class Point<T extends Number> {  
    T x, y;  
}
```

```
public <T extends MyClass> void print(T obj) {  
    // We can only print instances of MyClass  
    // and its subclasses  
}
```

Subclasses vs Generic Subtypes

- **Recap:** `Integer` and `Double` inherit from (extend) `Number`, therefore we can say
 - `Integer` **is a** `Number`
 - `Double` **is a** `Number`
- ...and so the following code works...

```
Number n = new Integer(4);  
Double d = new Double(2.3);  
  
ArrayList<Number> list = new ArrayList<Number>();  
list.add(n);  
list.add(d);  
list.add(new Float(2.0f));
```

Subclasses vs Generic Subtypes

- However `ArrayList<Integer>` does not inherit from `ArrayList<Number>`, so this doesn't work...

```
public void func(ArrayList<Number> list) { ... }
```

```
ArrayList<Number> list1 = new ArrayList<Number>();  
ArrayList<Integer> list2 = new ArrayList<Integer>();  
ArrayList<Double> list3 = new ArrayList<Double>();
```

```
func(list1);
```

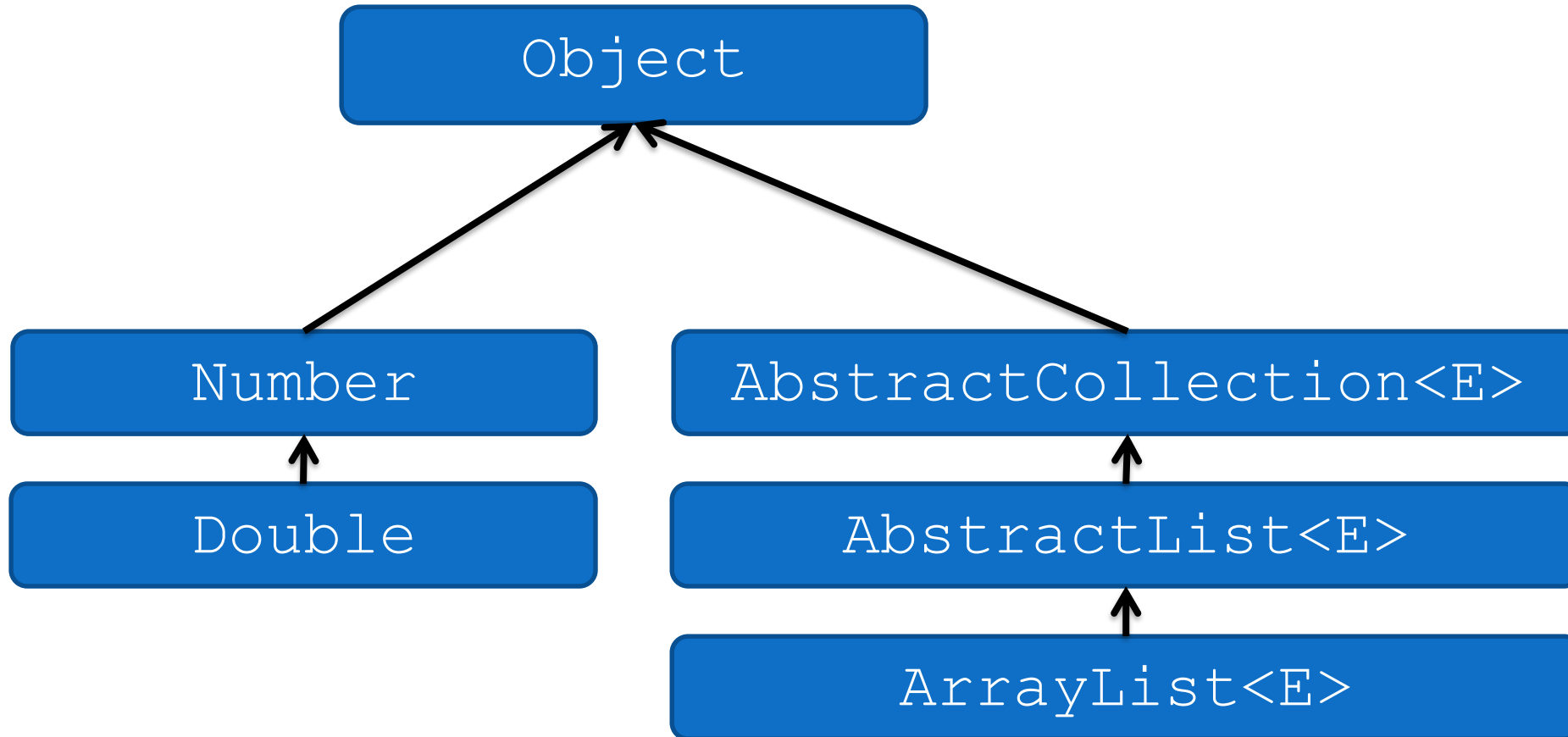
```
func(list2);
```

```
func(list3);
```

Compiler
errors

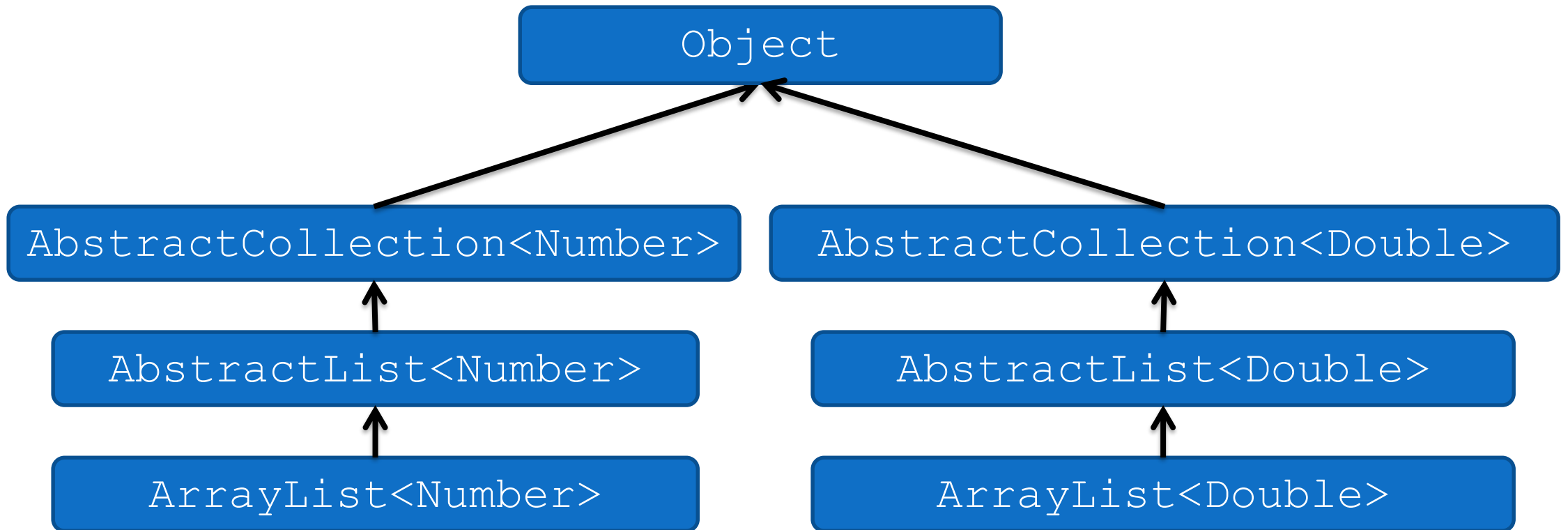
Subclasses vs Generic Subtypes

- Looking at the inheritance diagrams makes this clear...



Subclasses vs Generic Subtypes

- `ArrayList<Number>` is not even a cousin of `ArrayList<Double>`!



Reference

<https://docs.oracle.com/javase/8/docs/technotes/guides/language/generics.html>



EoF

