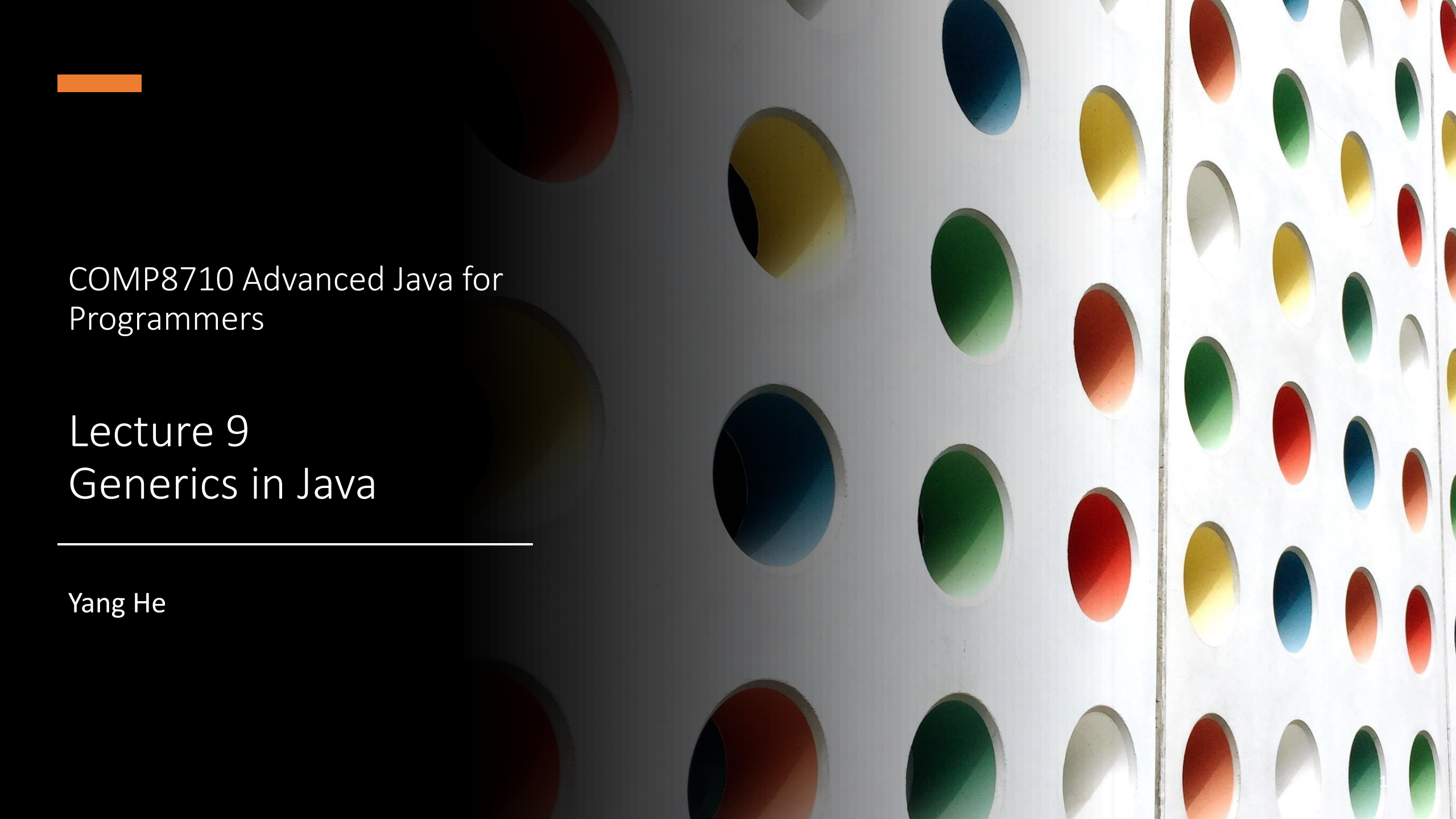

COMP8710 Advanced Java for Programmers

# Lecture 9
# Generics in Java

Yang He

# Topics

- Overview
- Generic type
- Iterable & Iterator
- Type erasure
- Wildcards
- Bounded wildcards

# Generics

- Generics were introduced in Java 5
  - Provide tighter type checks at compile time
  - Support generic programming
- Java compiler checks for type mismatch errors, so less need for casting
  - E.g.

```
List<String> names = new ArrayList<>();
names.add("Michael");
Names.add(3);     // compile error
```

- Generics aid readability, reliability, and reuse of code
- However, they can be quite complex
  - See *Generics in the Java Programming Language* by Gilad Bracha

# Defining simple generic types

- The diamond operator <> containing the type
- Type parameter E represents any type
- E.g. String, Book, etc.

```
List<String> names;
List<Book> books;
```

```
interface List<E> extends Iterable<E> {
   void add(E x);
}
interface Iterable<E> {
   Iterator<E> iterator();
}
interface Iterator<E> {
   E next();
   boolean hasNext();
   void remove();
}
```

# For-each loop

- Java collections (e.g. List, Map, Set) are Iterable
- E.g. print all names in a list using a for-loop

```
for (Iterator<String> it = names.iterator(); it.hasNext();) {
    System.out.println(it.next());
}
```

- For-each loop is more concise and easier to use

```
for (String name : names) {
    System.out.println(name);
}
```

# An example (1)

- Suppose we want an online catalogue of products which we want to allow clients to iterate through

```
public class Catalogue {
    private List<Product> products = new ArrayList<>();
}
```

- We could add an accessor for the list:

```
public List<Product> getProducts() {
   return products;
}
```

# An example (2)

- But this relinquishes our control over the collection!
- E.g.

```
catalogue.getProducts().remove(0);    // BAD IDEA!!
```

- We could make clone it:

```
public List<Product> getProducts() {
    return products.clone();
}
```

- But the list might be huge!

# Iterable and Iterator (1)

- A better solution:
  - Make the Catalogue class Iterable, and
  - Define our own Iterator, i.e. SafeIterator class

```
class Catalogue implements Iterable<Product> {
    private List<Product> products;

    // constructor, add, etc.

    @Override
    public Iterator<Product> iterator() {
        return new SafeIterator(products);
    }
}
```

# Iterable and Iterator (2)

```
public class SafeIterator implements Iterator<Product> {
    private Iterator<Product> it;
    public SafeIterator(List<Product> list) {
        it = list.iterator();
    }
    @Override
    public boolean hasNext() {
        return it.hasNext();
    }
    @Override
    public Product next() {
        return it.next();
    }
    @Override
    public void remove() {
        throw new RuntimeException("remove() attempted!");
    }
}
```

```
Catalogue catalogue = new Catalogue();

catalogue.add(new Product("Item 1", 15));
catalogue.add(new Product("Item 2", 10));

for (var product : catalogue) {
    System.out.println(product);
}

catalogue.remove();  // raise exception
```

# Generic classes and methods (1)

- We can define our own generic classes and methods for any type
- E.g. Replace Product by a type parameter E

```
class Catalogue<E> implements Iterable<E> {
    private List<E> list;

    // constructor, add, etc.

    @Override
    public Iterator<E> iterator() {
        return new SafeIterator<E>(list);
    }
}
```

# Generic classes and methods (2)

```
public class SafeIterator<E> implements Iterator<E> {
    private Iterator<E> it;
    public SafeIterator(List<E> list) {
        it = list.iterator();
    }

    @Override
    public boolean hasNext() {
        return it.hasNext();
    }

    @Override
    public E next() {
        return it.next();
    }

    @Override
    public void remove() {
        throw new RuntimeException("remove() attempted!");
    }
}
```

```
Catalogue<Product> products = new Catalogue<>();

Catalogue<Book> books = new Catalogue<>();
```

# Type parameter for a method (1)

- E.g. A method named replaceAll takes 3 parameters: `list`, `src`, `des`; it replaces all values `src` by `des` in `list`, and return the result

```
public List<String> replaceAll(List<String> list,
                                String src, String des) {
    List<String> result = new ArrayList<>(list);
    for (var i = 0; i < result.size(); i++) {
        if (result.get(i).equals(src)) {
            result.set(i, des);
        }
    }
    return result;
}
```

# Type parameter for a method (2)

- Using a type parameter T, we can define replaceAll method to be a generic method that works for any type
- E.g.

```
public <T> List<T> replaceAll(List<T> list, T src, T des) {
    List<T> result = new ArrayList<>(list);
    for (var i = 0; i < result.size(); i++) {
        if (result.get(i).equals(src)) {
            result.set(i, des);
        }
    }
    return result;
}
```

# Subtyping

- A variable of a given type may be assigned a value of its subtypes

```
Animal animal = new Cat();
```

*Cat is a subtype of Animal*

- Subtyping extends naturally to generic types

```
List<Cat> cats = new ArrayList<>();
```

*ArrayList<Cat> is a subtype of List<Cat>*

- Is this legal?

```
List<Animal> animals = cats;
```

*No, because List<Cat> is **not** a subtype of List<Animal>*

# Type erasure (1)

- Java compiler applies type erasure to implement generics
  - All generics types are only available at compile-time
  - Once compiled, all type parameters in generic types are replaced with Object or their bounds if the type parameters are bounded
  - The byte-code generated (and hence the JVM) knows nothing about generics
  - The JVM must still perform run-time type checks for code that uses generics

# Type erasure (2)

- E.g.

```
List<String> names = new ArrayList<>();
List<Integer> marks = new ArrayList<>();
names.add("Alan Turing");
marks.add("50");      // Compile error
System.out.println(names == marks);      // compile error
System.out.println(names.getClass() == marks.getClass());      // true

// instanceof returns dynamic type
System.out.println(names instanceof ArrayList<String>);      // true
```

*Once compiled, both names and marks become ArrayList<Object>*

# Wildcards (1)

- How might we print the elements in a collection, in a generic way?

```
void printAll(Collection<Object> collection) {
    for (Object obj : collection)
        System.out.println(obj);
}
```

- But this won't work because Collection<Object> is NOT the supertype of all collections, e.g. List<Book>

# Wildcards (2)

- To solve the problem, we can use the wildcard type parameter ?

```
void printAll(Collection<?> collection) {
    for (Object obj : collection)
        System.out.println(obj);
}
```

- Read Collection<?> as “Collection of some unknown type”. It is the supertype of all collections

# Bounded wildcards (1)

- Suppose that we don't want to allow any collection to be used, and instead we want to impose some restrictions
- Consider a drawing application that deals with Shapes such as Circle, Box, and so on. Then suppose that we want to draw all the shapes in a collection

```
public interface Shape {...}
public class Box implements Shape {...}
public class Circle implements Shape {...}

public static void drawAll(Collection<? extends Shape> list) {
  // ...
}
```

Upper bound

*Collection of elements of any subtype of Shape*

# Bounded wildcards (2)

- E.g.

```
List<Box> boxes = new ArrayList<>();
drawAll(boxes);

Set<Circle> circles = new HashSet<>();
drawAll(circles);

Collection<Book> books = new ArrayList<>();
drawAll(books);      // compile error
```

# Bounded wildcards (3)

- Suppose there is a type dependency between two parameters of a method
- E.g. copy all elements from an array to a list
- We  can define a generic method using a type parameter:

```
public <T> void fromArrayToList(T[] arr, List<T> list) {
    for (T obj : arr)
        list.add(obj);
}
```

# Type inference

- Case 1:

```
Object[] obj_arr = new Object[100];
List<Object> obj_list = new ArrayList<>();
fromArrayToList(obj_arr, obj_list);   // T inferred to be Object
```

- Case 2:

```
String[] str_arr = new String[100];
List<String> str_list = new ArrayList<>();
fromArrayToList(str_arr, str_list);   // T inferred to be String
```

- Case 3:

```
fromArrayToList(str_arr, obj_list);   // T inferred to be Object
```

# Wildcard or generic type parameter?

- Used a wildcard when:
  - The type parameter is only used once
  - It does not depend on any other argument to the method
  - E.g.

```
void printCollection(Collection<?> c) {...}
```

- Used a generic type parameter T when:
  - The type parameter is used more than once
  - There is a type dependency between the arguments (or the return type)
  - E.g.

```
<T> void fromArrayToCollection(T[] a, Collection<T> c) {...}
```

# Summary

- Generics ensure type parameters agree

```
Collection<String> list = new ArrayList<>();      //  ArrayList<String>
```

- `Collection<String>` is not a subtype of `Collection<Object>`
- `Collection<?>` is the supertype of any collections
- Use generic methods where there is a dependency

```
<T> List<T> findMatch(T[] arr, List<T> list)
```

- If you are going to make anything other than the simplest use of generics, be careful! Read up on them, consult books/tutorials, etc.