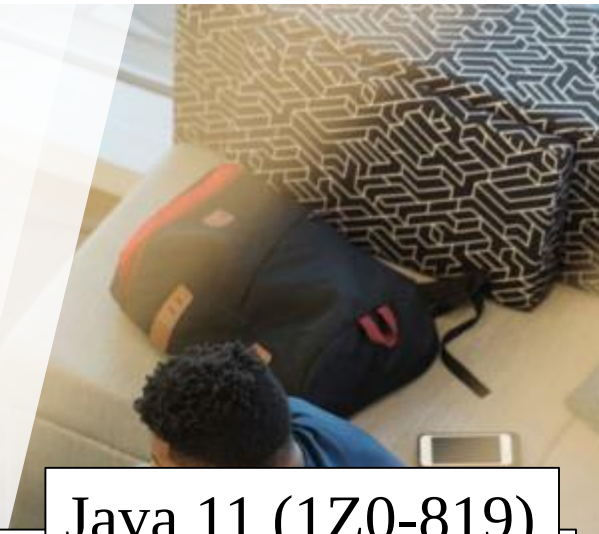
Collections

Generics

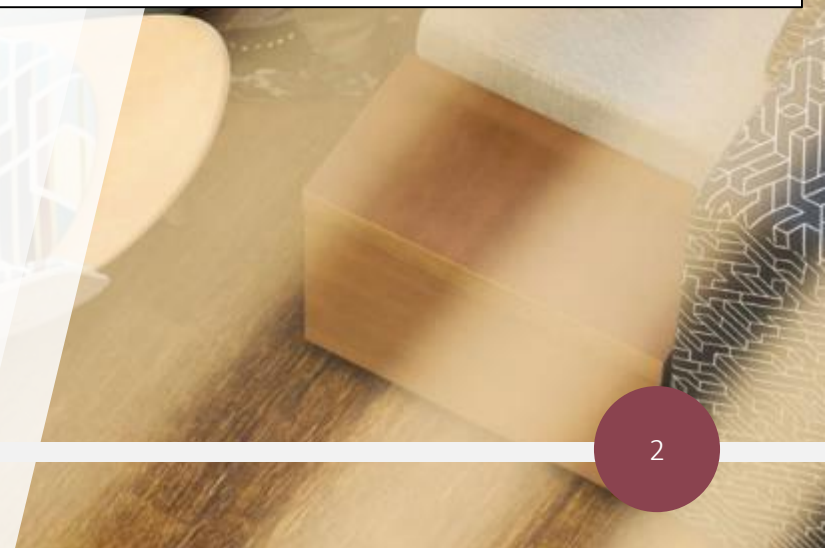
Collections



Java 11 (1Z0-819)

Working with Arrays and Collections

- Use generics, including wildcards
- Use a Java array and List, Set, Map and Deque collections, including convenience methods
- Sort collections and arrays using Comparator and Comparable interfaces

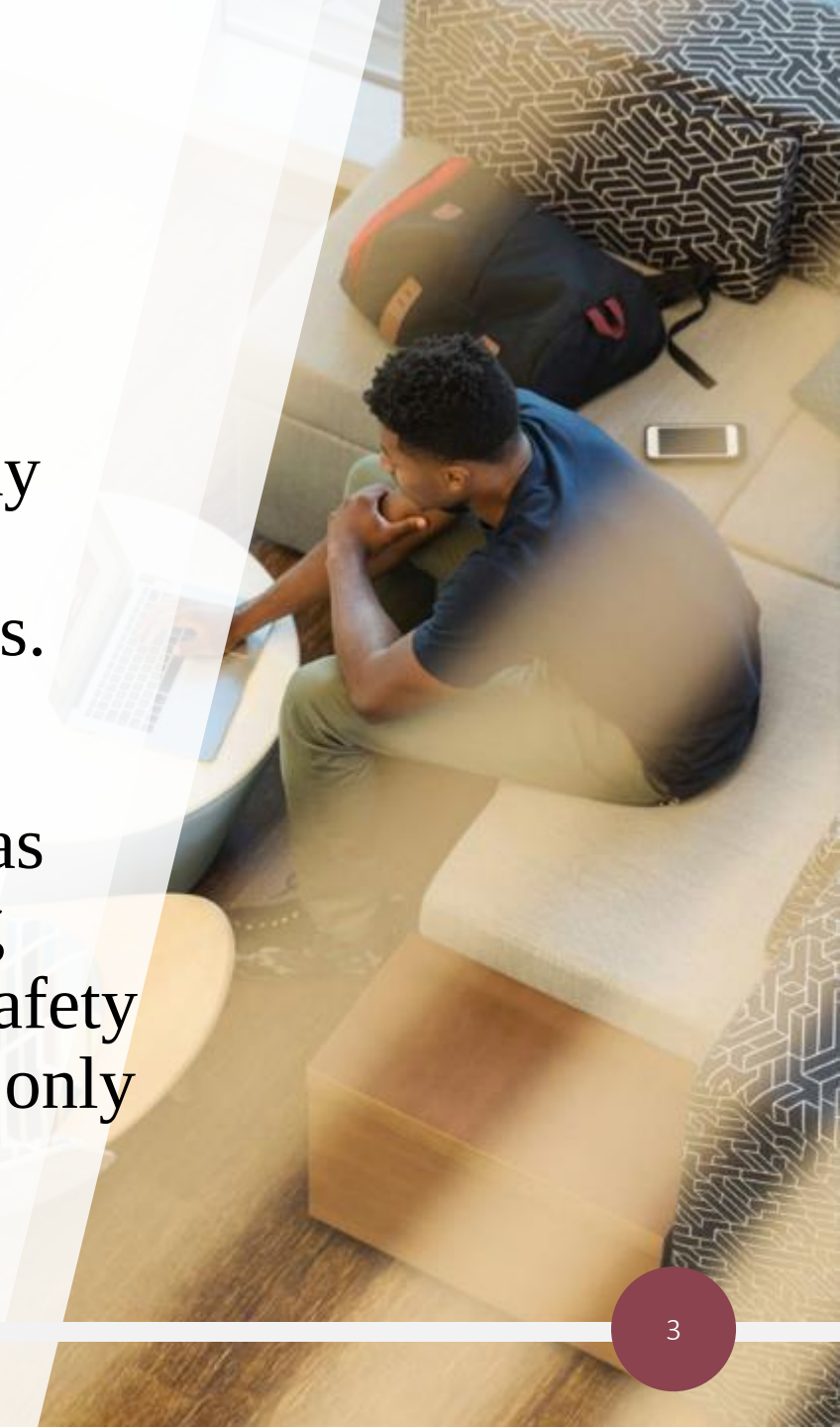


Generics

- Generics were introduced in Java 1.5

- Arrays in Java have always been type-safe - an array declared as type *String* (*String* []) cannot accept *Integer*s (or *int*s), *Dog*s or anything other than *String*s.

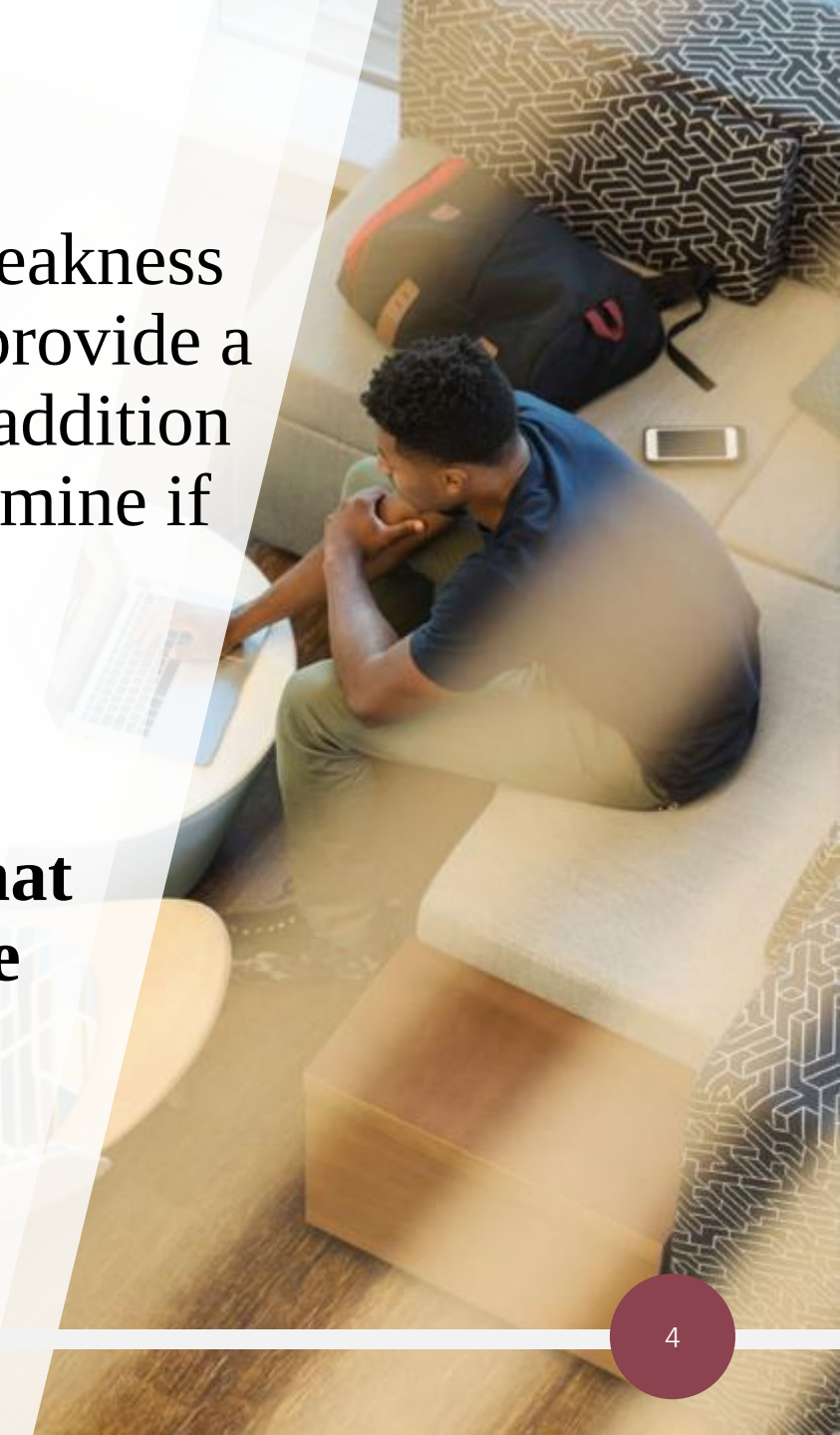
- Prior to generics, where the collections are known as "raw" collections, the *Object* class is used for storing elements in containers. This however, leads to type safety issues which cannot be detected by the compiler and only manifests themselves at runtime.



Generics

- A container that stores *Object* types has a critical weakness - type information is lost. This means that you must provide a cast when retrieving elements from the container. In addition and more importantly, the compiler is unable to determine if the cast is correct - a coding error. This results in a *ClassCastException* at runtime.

- **With generics, you can specify to the compiler that only elements of a certain type can be added to the container.**



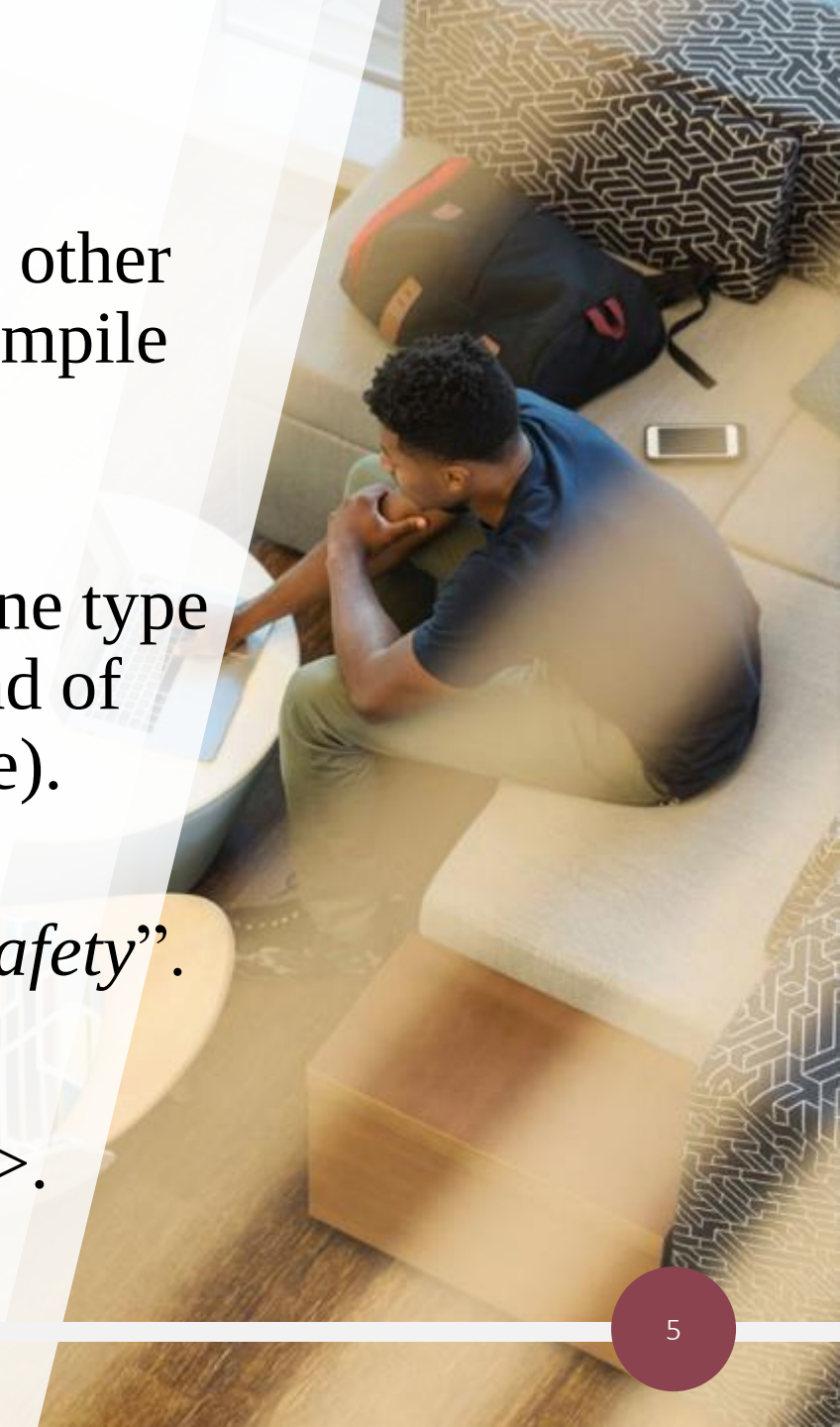
Generics

- Generics will ensure that any attempt to add a type other than the particular type specified will be caught at compile time. This is known as “type-safety”.

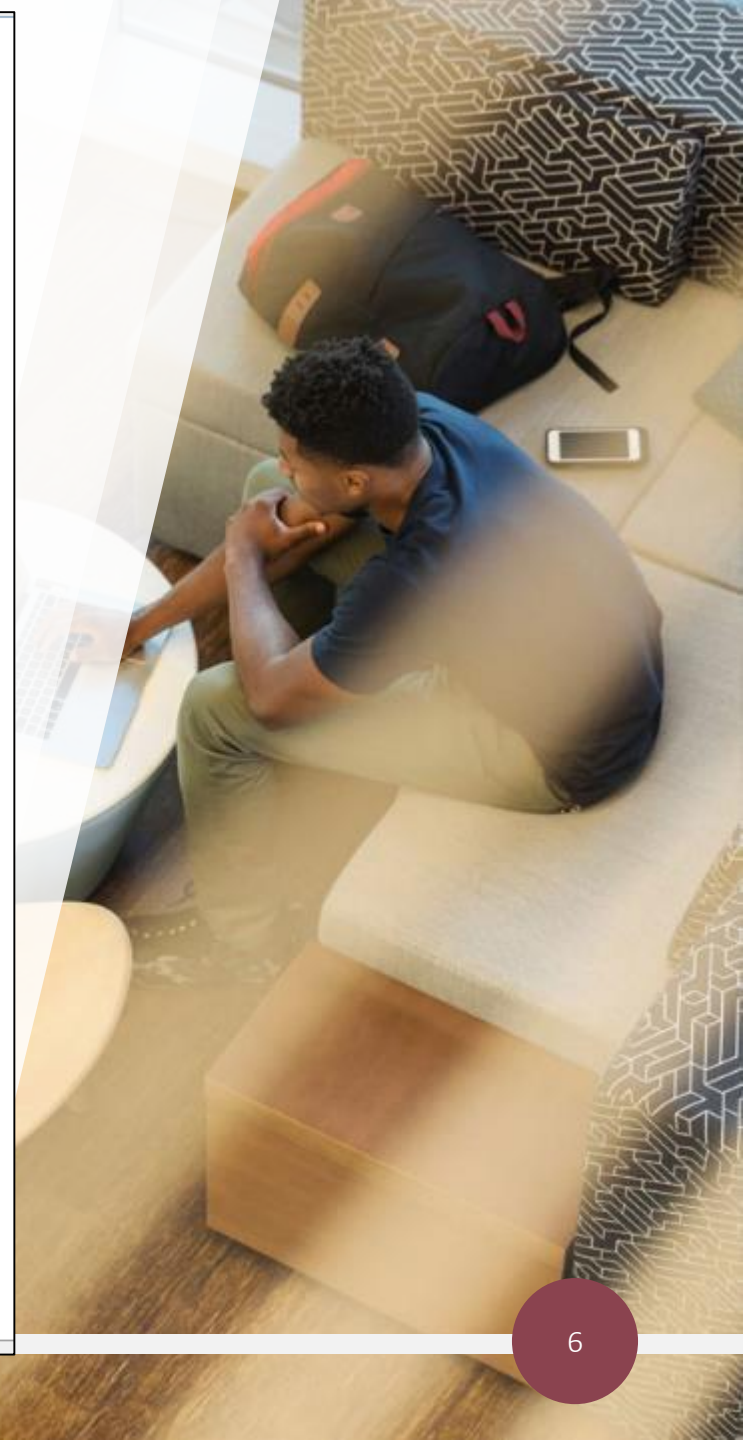
- In addition, generics enable you to write code for one type (for example *T*) that is applicable for all types (instead of having to write separate classes for each specific type).

- Generics offer “*generic implementation with type safety*”.

- The generic type is the type in the angle brackets <>.



```java
package lets_get_certified.generics;

import java.util.ArrayList;
import java.util.List;

public class PreGenerics {
    public static void main(String []args) {
        // A raw collection can hold any type of Object (except a primitive).
        List myList = new ArrayList();  // can't enforce a type
        myList.add("Fred");             // will hold Strings
        myList.add(new Dog());          // and Dogs
        myList.add(43);                 // and Integers (autoboxing)

        // As everything is treated as an Object, when you are getting something out of
        // the collection, all you ha were Object's - therefore a cast was required to
        // get your String back.
        String s = (String)myList.get(0);// cast required else compiler error
        // and as we could not guarantee that what was coming out
        // was really a String (as we were allowed to put anything in),
        // this cast could fail at runtime
        String s1 = (String)myList.get(1);// ClassCastException at runtime

        // Generics takes care of both ends (putting in and getting out)
        // by enforcing the type of your collections.
        // Note: generic syntax means putting the type in angle brackets
        List<String> myList2 = new ArrayList<String>();
        myList2.add("Fred");            // will hold Strings
        myList2.add(new Dog());         // compiler error
        // Because what is going IN is guaranteed, what is coming OUT is
        // also guaranteed => no need for the cast
        String s2 = myList2.get(0);// cast no longer required
    }
}
```

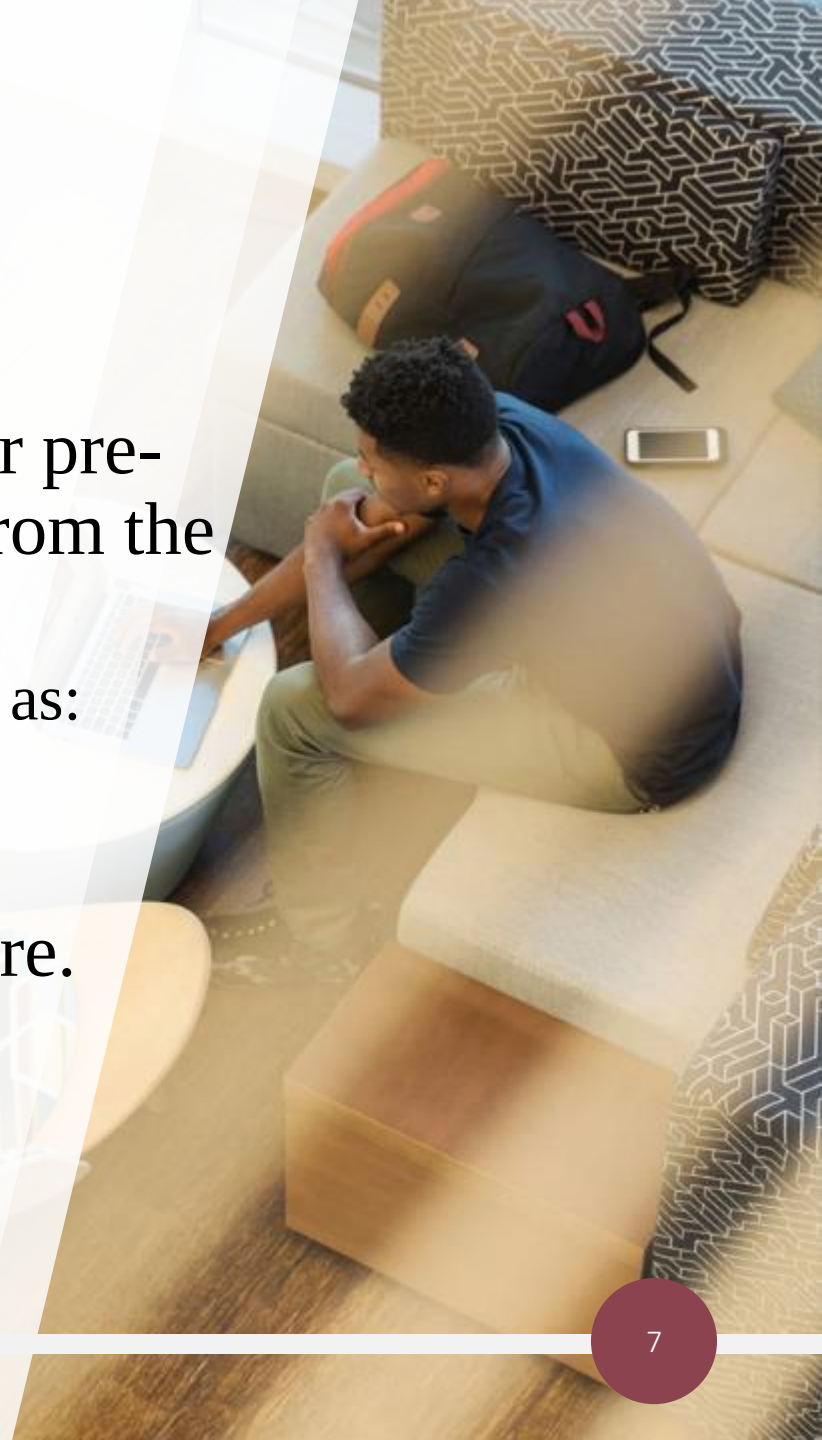


Type Erasure

- Pre-generics, Java used *Object* types in collections.

- In order for generic code to be compatible with older pre-generic code, all of the type information is removed from the bytecode. This means that:
  - *List<String> list = new ArrayList<String>();* is the same as:
    List list = new ArrayList(); // legacy syntax, *Object* used

- Generics are strictly a compile-time protection feature.



Polymorphism and Generics

- Polymorphism applies to the **base** type:
  - **List**<Integer> myList = new **ArrayList**<Integer>();

- Polymorphism does NOT apply to the **generic** type:
  - List<**Number**> myList = new ArrayList<**Integer**>(); // **NO**

- Issue:

```
// The issue
List<Double> doubles = new ArrayList<Double>();
doubles.add(12.3);
List<Object> objects = doubles; // COMPILER ERROR
objects.add("This is a String");
```

```java
package lets_get_certified.generics;

import java.util.ArrayList;
import java.util.List;

public class PolymorphicIssueWithGenerics {
    public static void showList(List<Object> list){
        for(Object o:list){
            System.out.println(o);
        }
    }
    public static void main(String[] args) {
        // The issue
        List<Double> doubles = new ArrayList<Double>();
        doubles.add(12.3);
        List<Object> objects = doubles; // COMPILER ERROR
        objects.add("This is a String");

        // A different variation
        List<String> names = new ArrayList<String>();
        names.add("Sean");
        showList(names); // List<Object> list = new ArrayList<String>();
    }
}
```

Wildcard Generic Type

- To solve the polymorphism issue for generics, we use the wildcard question mark symbol i.e. ?.

Type	Syntax	Example	Add items?
Unbounded wildcard	?	List<?> l = new LinkedList<Integer>();	No – readonly
Upper bound wildcard	? extends type	List<? extends Number> l = new LinkedList<Integer>();	No - readonly
Lower bound wildcard	? super type	List<? super Number> l = new LinkedList<Object>();	Yes

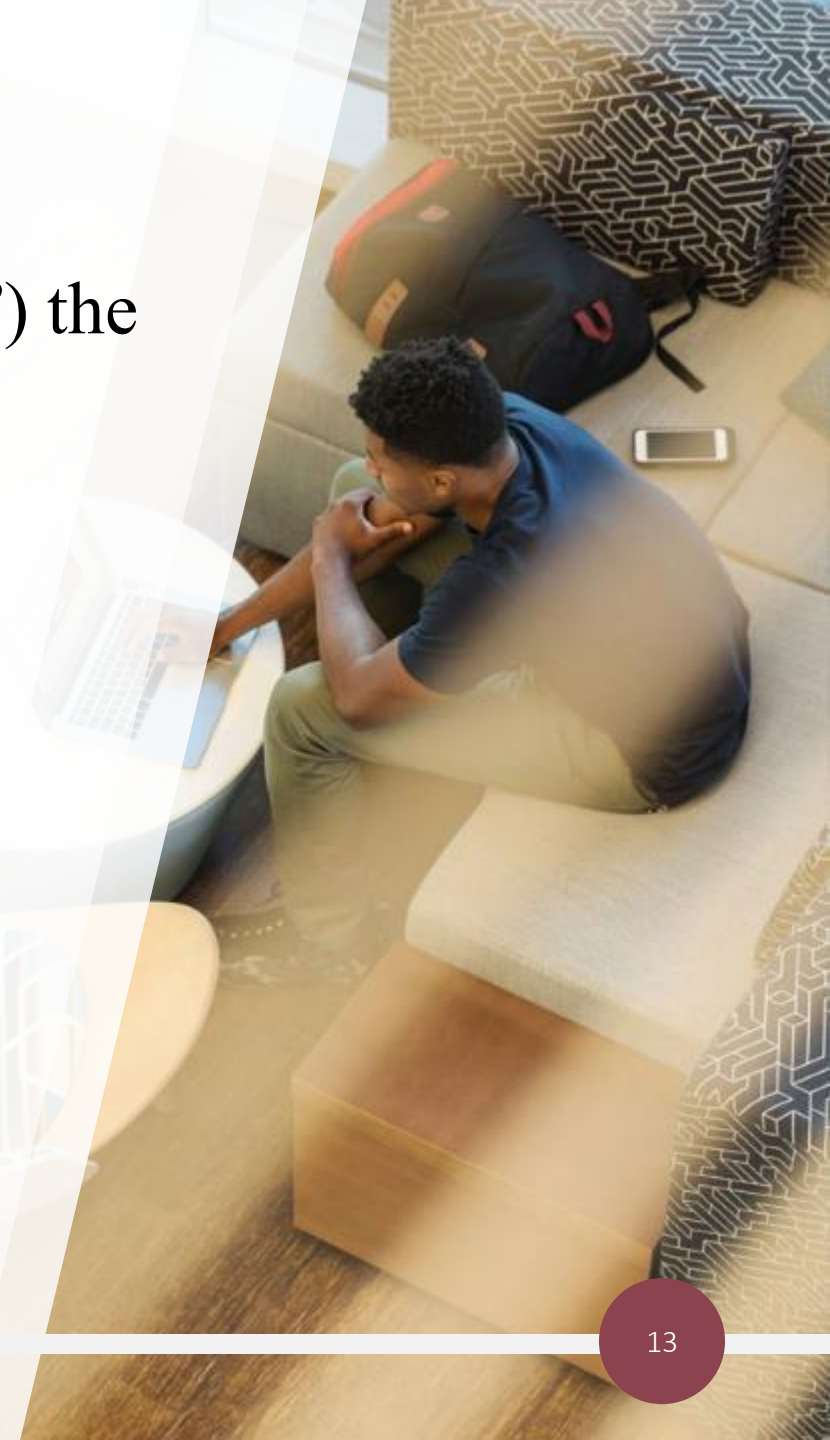
Unbounded wildcard

```java
package lets_get_certified.generics;

import java.util.ArrayList;
import java.util.List;

public class UnboundedWildcard {
//    public static void showList(List<Object> list){
    public static void showList(List<?> list){ // any type is ok
        for(Object o:list){
            System.out.println(o);
        }
        list.add("test"); // <?> implies read-only
    }
    public static void main(String[] args) {
        // A different variation
        List<String> names = new ArrayList<String>();
        names.add("Sean");
        showList(names); // List<?> list = new ArrayList<String>();
        List<Dog> dogs = new ArrayList<Dog>();
        dogs.add(new Dog());
        showList(dogs); // List<?> list = new ArrayList<Dog>();
        List<Cat> cats = new ArrayList<Cat>();
        cats.add(new Cat());
        showList(cats); // List<?> list = new ArrayList<Cat>();

    }
}
```

Bounded Wildcards

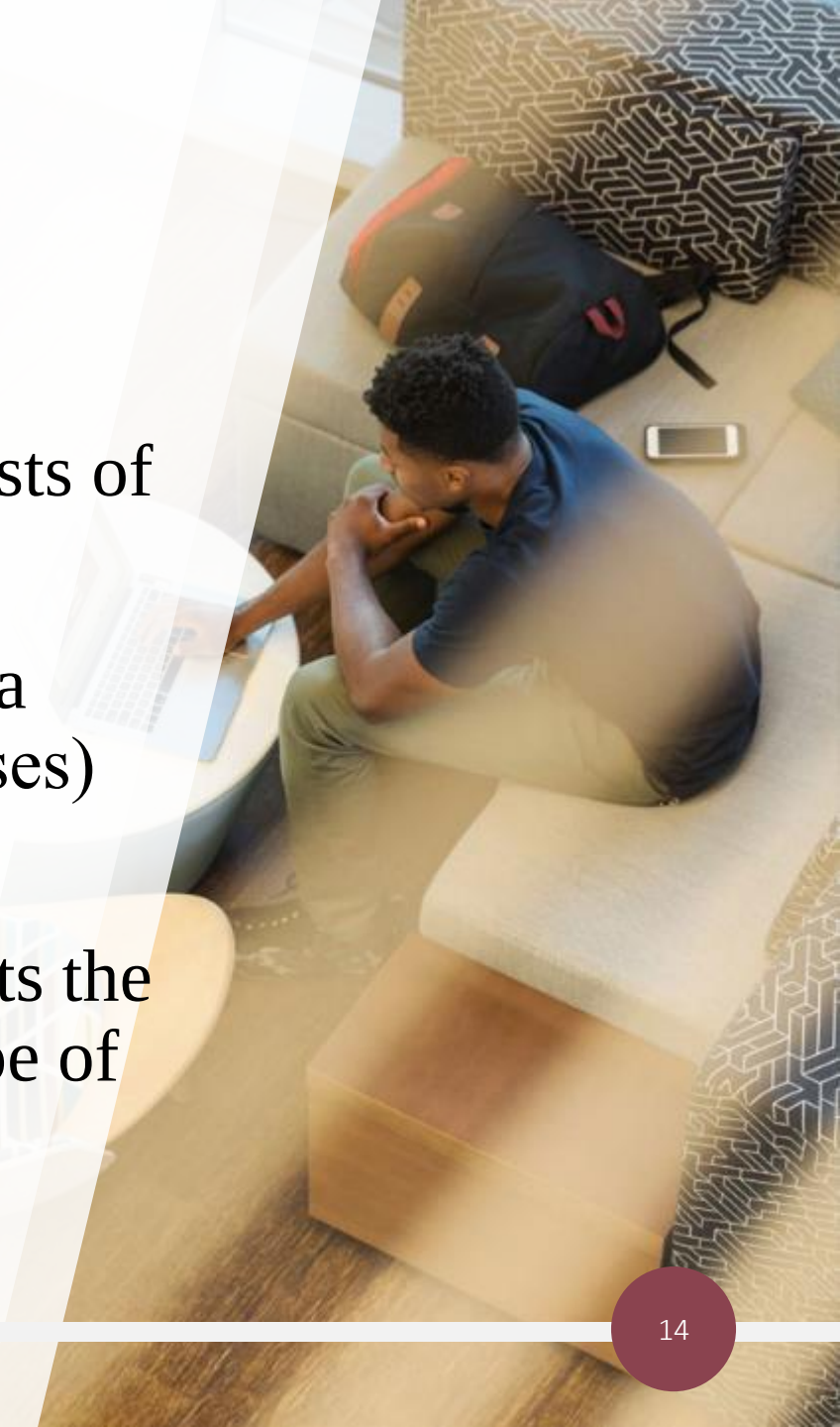
- Bounded wildcards are a way to limit (or “bound”) the types that can be used.

- You can bound in both directions i.e. upward and downward.



extends

- Downward syntax is:
  - *someMethod(List<**? extends** Number> list)*
    - *list* is a method parameter that can handle lists of *Number*, *Integer*, *Double* etc...
    - note, that in this context, *extends* is used in a general sense to mean “extends” (as in classes) but **also “implements” (as in interfaces)**.
  - known as “*upper bounded wildcards*” - restricts the unknown type to be a specific type or a subtype of that type
  - read-only



super

- Upward syntax is:
  - *someMethod(List<? **super** Integer> list)*
    - *list* is a method parameter that can handle lists of *Integer* or any super type of *Integer*

  - known as "*lower bounded wildcards*" - restricts the unknown type to be a specific type or a super type of that type

  - safe to add to the collection

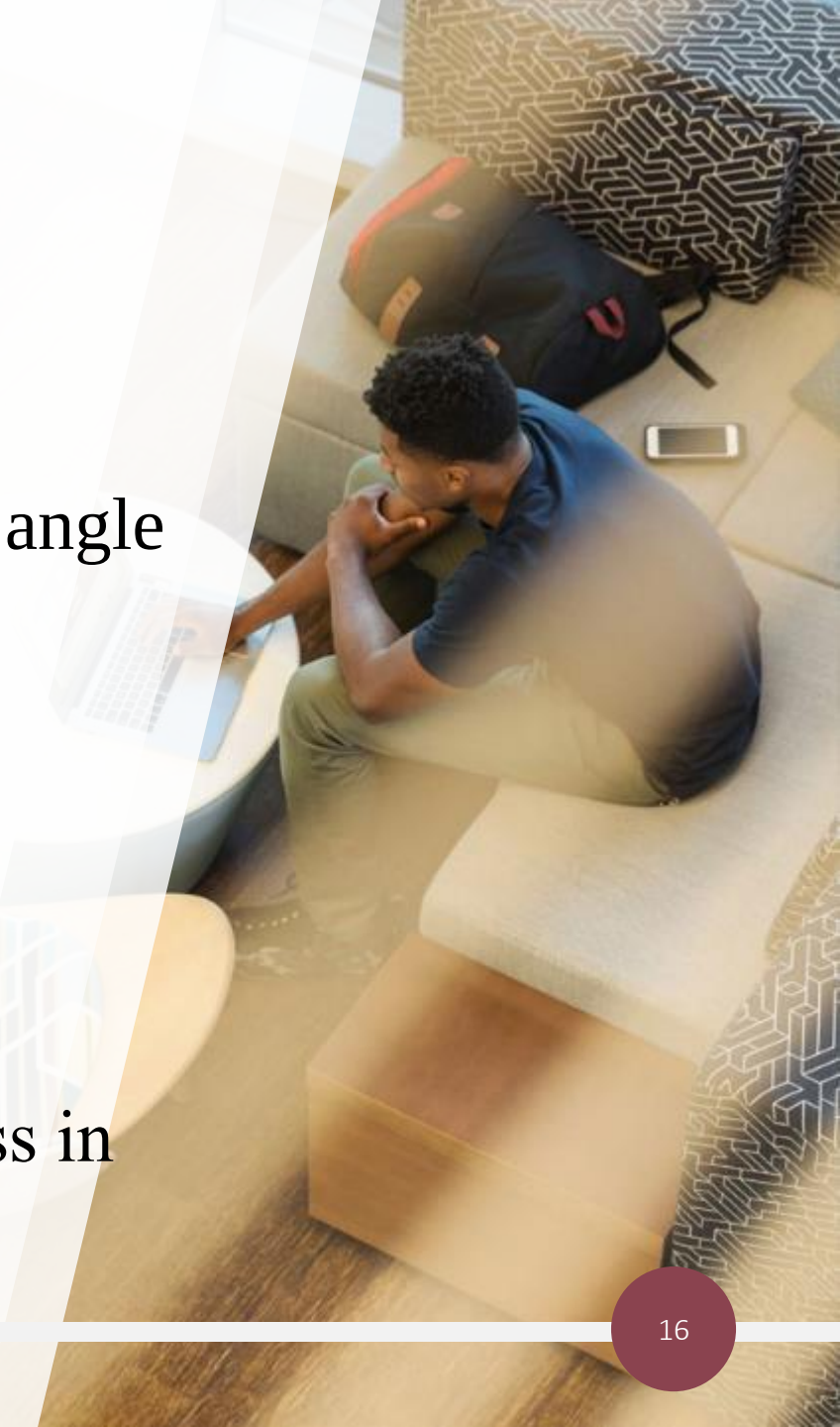
Generic Classes

- We can add generics to our own types (classes and interfaces).

- The syntax is to declare a formal type parameter in angle brackets.

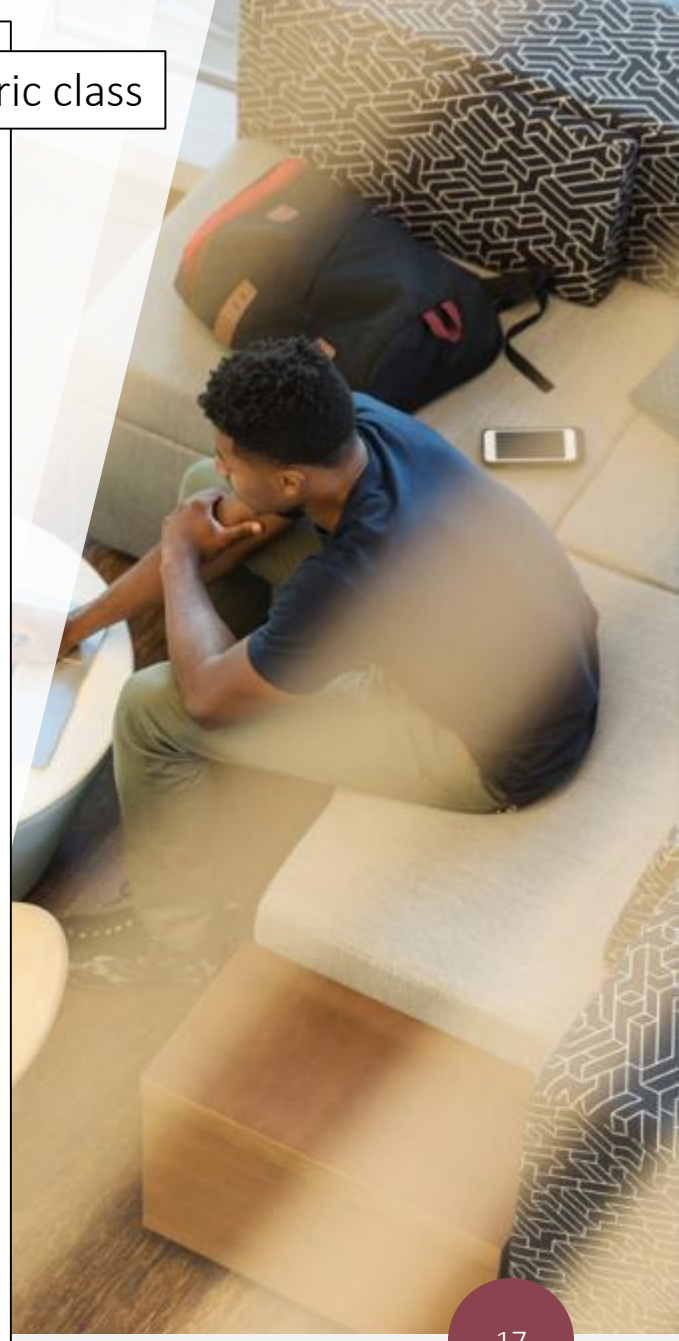
- This can be seen in the API:
  - *public interface List<**E**>* with:
    - *boolean add (**E** e)*

- The “E” above is a placeholder for the type you pass in e.g. *List<String>* where *String* replaces E.



Generic class

```java
package ch11_generics.generic_class;

import java.util.ArrayList;
import java.util.List;

class MyGeneric<T>{
    T instance;
    MyGeneric(T instance){
        this.instance=instance;
    }
    T getT(){
        return instance;
    }
}
public class TestGenericClass {
    public static void main(String []args){
        // String on LHS maps to T and "SK" on RHS maps to 'instance'
        MyGeneric<String> g = new MyGeneric<>("SK");
        System.out.println(g.getT());
        // Integer on LHS maps to T and 1 on RHS maps to 'instance'
        MyGeneric<Integer> g2 = new MyGeneric<>(1);
        System.out.println(g2.getT());
    }
}
```



Naming Conventions

- Can be anything but the convention is to use single uppercase letters.
- E is for element; T is for a generic type
- K is a map key; V is a map value
- N is a number
- S, U, V are for multiple generic types

```java
package lets_get_certified.generics;

class Register<T, U, V>{
    private T type;
    private U name;
    private V age;

    Register(T type, U name, V age){
        this.type = type;
        this.name = name;
        this.age  = age;
    }
    public T getType(){
        return type;
    }
    public U getName(){
        return name;
    }
    public V getAge(){
        return age;
    }
}
public class AnimalRegister {
    public static void main(String[] args) {
        new Register(new Dog(), "Shep", 3);
        new Register(new Cat(), "Whiskers", 2);
    }
}
```

Multiple types.

Generic Interfaces

- Interfaces can declare formal type parameters also.

```
interface Moveable<T>{
    void move(T t);
}
```

Generic interfaces.

```
package lets_get_certified.generics;

interface Moveable<T>{
    void move(T t);
}
class MoveFeline implements Moveable<Cat>{
    public void move(Cat c){}
}
class MoveCanine implements Moveable<Dog>{
    public void move(Dog d){}
}
class SomeMoveable<U> implements Moveable<U>{
    public void move(U u){}
}
public class GenericInterface {
    public static void main(String[] args) {
        new MoveFeline().move(new Cat());
        new MoveFeline().move(new Dog()); // compiler error
        new MoveCanine().move(new Dog());
        new MoveCanine().move(new Cat()); // compiler error

        new SomeMoveable<Dog>().move(new Dog());
        new SomeMoveable<Cat>().move(new Cat());
    }
}
```

Generic Methods

- Formal type parameters can also be used on methods.

- The generic marker (formal type parameter) is declared just before the return type. Note that the return type can also incorporate the generic marker (which can make the code tricky).

Generic methods.

```java
package lets_get_certified.generics;

public class GenericMethods {

    public static <T> void genericMethod(T t){
        MyGeneric<T> myGen = new MyGeneric<>(t);
        System.out.println(myGen.getT());
    }
    public static <T, U, V> void register(T t, U u, V v){
        Register<T, U, V> register = new Register<>(t, u, v);
        System.out.println("Register: "+register.getName()+"; "+register.getAge());
    }
    public static <T> MyGeneric<T> createGeneric(T t){
        return new MyGeneric<>(t);
    }

    public static void main(String[] args) {
        genericMethod("SK");     // SK
        genericMethod(1.1);      // 1.1

        register(new Dog(), "Shep", 3);      // Register: Shep; 3
        register(new Cat(), "Whiskers", 2); // Register: Whiskers; 2

        MyGeneric<Integer> myGenI = createGeneric(4);
        System.out.println(myGenI.getT());  // 4
    }
}
```