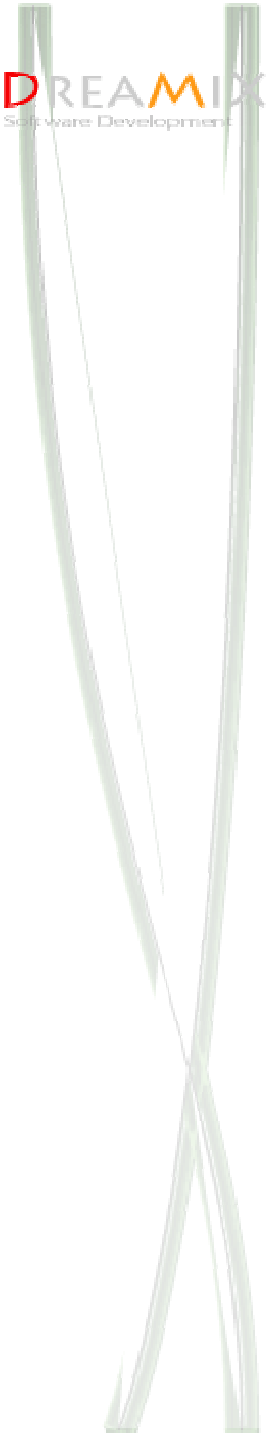

Generics

Angel Gruev
Dreamix ltd.

Terms

- A class or interface whose declaration has one or more *type parameters is a generic class or interface*
  - List<E> (read “list of E”)
- Compile time check!
- Raw types
  - Compatibility

Raw types

- List vs. List<Object>
  - Not the same!

```
List a;
List<Object> b;

List<String> c = new List<String>();

a = c; //??
b = c; //??
```

- Subtyping rules!

Do not use raw types

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
  int result = 0;
  for (Object o1 : s1)
    if (s2.contains(o1))
      result++;
  return result;
}

 // Unbounded wildcard type - typesafe and flexible
 static int numElementsInCommon(Set<?> s1, Set<?> s2) {
   int result = 0;
   for (Object o1 : s1)
     if (s2.contains(o1))
       result++;
   return result;
 }
```

Exceptions for raw types

- Generic type information is erased at runtime
- You must use raw types in class literals
  - `List.class, String[].class` – **OK**
  - `List<String>.class, List<?>.class` – **Not OK**
- instanceof operator
  - illegal to use the **instanceof** operator on parameterized types

```
    // Legitimate use of raw type - instanceof operator
    if (o instanceof Set) { // Raw type
    Set<?> m = (Set<?>) o; // Wildcard type
    ...
}
```

Eliminate unchecked warnings

- **Eliminate every unchecked warning that you can**
  - It means that you won't get a ClassCastException at runtime
- **If you can't eliminate a warning**
  - **@SuppressWarnings("unchecked") annotation**

Prefer lists to arrays

- arrays are *covariant*
  - *Sub -> Super ⇔ Sub[] -> Super[]*
- Generics, by contrast, are *invariant*
  - *Sub -> Super ⇔ List<Sub> ≠ List<Supper>*

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException

// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

Convert to generics

DEMO

Generic static factory method

```
// Parameterized type instance creation with constructor
Map<String, List<String>> anagrams =
new HashMap<String, List<String>>();
```

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
return new HashMap<K,V>();
}
```

```
// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

Recursive type bound

```
// Using a recursive type bound to express
// mutual comparability
public static <T extends Comparable<T>>
                                T max(List<T> list) {...}
```

Unbounded wildcards

```
public class Stack<E> {
  public Stack();
  public void push(E e);
  public E pop();
  public boolean isEmpty();
}
```

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
  for (E e : src)
  push(e);
}
```

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Unbounded wildcards

```
public class Stack<E> {
  public Stack();
  public void push(E e);
  public E pop();
  public boolean isEmpty();
}
```

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
```

StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number> cannot be applied to (Iterable<Integer>)
numberStack.pushAll(integers);

```
Stack<Numb          Stack = new Stack<Number>();
Iterable<Int        > integers = ... ;
numberStack.pushAll(integers);
```

Unbounded wildcards

- Solution

```
// Wildcard type for parameter that serves
// as an E producer
public void pushAll(Iterable<? extends E> src) {
  for (E e : src)
  push(e);
}
```

Unbounded wildcards

- Different example - popAll:

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
  while (!isEmpty())
  dst.add(pop());
}
```

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

- Again error similar to the previous one!!

Unbounded wildcards

- Solution

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
  while (!isEmpty())
  dst.add(pop());
}
```

- **PECS stands for producer-extends, consumer-super.**

Questions

- How to transform:

```
static <E> E reduce(List<E> list, Function<E> f, E initVal)
```

- How to transform:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

wildcards

If the user of a class has to think about wildcard types, there is probably something wrong with the class's API.

Wildcards

- A swap method declaration – which is better?

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Relax

- A question for relaxation

```
public class Elementary {
	public static void main(String[] args) {
		System.out.println(12345 + 5432l);
	}
}
```

- What is the result?

Not relaxed enough?

```
class Base { public String className = "Base"; }

class Derived extends Base {
  private String className = "Derived";
}

public class PrivateMatter {
  public static void main(String[] args) {
    System.out.println(new Derived().className);
  }
}
```

DREAMIX
Software Development

Questions?