cis112

# Generics

BBBF
Yeditepe University

v2025-03-01

# Agenda

- Why Generic Programming?
- Writing Generic Classes
- Writing Generic Methods
- Generic Arrays
- Inheritance and Generic Classes

# Why Generic Programming?

- Generic programming = Writing code that can be reused for objects of many different types.
  - ❖ Don't want separate classes to collect *String* and *File* objects.

# Generic Types

- Java has syntax for *parameterized data types* referred to as ***Generic Types*** in most of the literature
- A traditional parameter *has* a data type and can store various values just like a variable

```
public void foo(int x)
```

- Generic Types are ***like*** parameters, but the values for the parameter are *data types*
  - like a variable that stores a data type
  - **this is an abstraction**. Actually, all data type info is erased at compile time and replaced with casts and, typically, variables of type `Object`

# Generics

- Generics is a language feature that enables the definition of classes and methods that are implemented independently of some type that they use as an **abstraction** by accepting a type parameter.
- The goal is to define types and algorithms that apply in **different contexts**, but where the interface and general functioning and implementation can be defined such that it **applies independently** of the context of application.
- Generic types are **instantiated** to form **parameterized types** by providing actual type arguments that replace the formal type parameters.

# Define a Simple Generic Class

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first; this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }
}
```

# Type Variables

- The `T` in public class `Pair<T>` is a *type variable.*
- The type variable is used throughout the class definition:

```
private T first;
```

  - Instantiate the type like `Pair<String>`, substituting a type for the variable.
  - You can think of the result as an ordinary class with these methods:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

# Type Variables

A class can have multiple type variables:

```
public class Pair<T, U>
{
   T first;
   U second;
   . . .
}
```

# Generic Methods

- Generic class = Class with type variables.
- Generic method = Method with type variables:

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

- When you call a generic method, the actual type comes before the method name:

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

# Bounds for Type Variables

Sometimes, a type variable cannot be instantiated with arbitrary types:

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

# Bounds for Type Variables

- How do we know that T has a *compareTo* method?
- Need to restrict T in the method declaration:

```
public static <T extends Comparable> T min(T[] a) ...
```

- Now **min** can be called with arrays of String, LocalDate, and so on, but not Rectangle.

**Note:** Comparable also has a type parameter which we will ignore for a little longer.

# Type Erasure

- The Java virtual machine has no notion of generic types or methods.
- Generic classes and methods turn into ordinary classes and methods.
- Type variables are "**erased**", yielding a raw type:

```
public class Pair
{
   private Object first;
   private Object second;
   public Pair(Object first, Object second) { . . . }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }
   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }
}
```

  - raw type
    - = generic type, with the type parameters removed
    - and replaced by their bounding types (or Object)

# Primitive Types

Since type variables are erased to Object, they don't work for primitive types

- Remedy 1: Use wrapper classes.

- Remedy 2: Make added versions for primitive types

  More painful than originally thought.

  java.util.function and java.util.stream have lots of baggage for primitive types.

  Example: Consumer, IntConsumer, LongConsumer, DoubleConsumer

# Java Wrapper Classes

Wrapper classes provide a way to use primitive data types ( `int` , `boolean` , etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

| Primitive Data Type | Wrapper Class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

# Creating Arrays

- You cannot instantiate arrays of parameterized types:

```
Pair<String>[] pairs = new Pair<String>[10]; // Error
```

- Suppose it worked. After erasure:

```
Pair[] pairs = new Pair[10];
```

- An array remembers its component type so that you can't store an element of the wrong type:

```
Object[] objects = pairs;
objects[0] = "Fred";
   // ArrayStoreException: Can't store a String into a Pair[]
```

# Creating Arrays

- But the array only remembers its raw type:

```
objects[0] = new Pair<Integer>(...);
   // No ArrayStoreException
```

- Since arrays of generic types are unsound, they are illegal.

**TIP:**

If you need to collect parameterized type objects, simply use an `ArrayList`: `ArrayList<Pair<String>>` is safe and effective.

# Instantiating Types

- You cannot instantiate a generic type:

```
public Pair() { first = new T(); second = new T(); } // Error
```

- Can remedy by making caller pass a constructor expression:

```
Pair<String> p = Pair.makePair(String::new);
```

Implementation:

```
public static <T> Pair<T> makePair(Supplier<T> constr)
{
   return new Pair<>(constr.get(), constr.get());
}
```

# Constructing Generic Arrays

- Illegal to have generic new expression for arrays since the component type would be erased.
- Suppose this was legal:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
   T[] mm = new T[2]; // Error
   . . .
}
```

Erasure would be:

```
public static Comparable[] minmax(Comparable[] a)
{
   Comparable[] mm = new Comparable[2];
   . . .
}
```

# Constructing Generic Array

- If the array is only used internally, can use Object[] array and casts:

```
public class ArrayList<E>
{
   private Object[] elements;
   . . .
   @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
   public void set(int n, E e) { elements[n] = e; } // no cast needed
}
```

# Static Contexts

- You cannot use type variables in static fields or methods.
- For instance, the following does not work!!:

```
public class Singleton<T>
{
   private static T singleInstance; // Error
   public static T getSingleInstance() // Error
   {
      if (singleInstance == null) construct new instance of T
      return singleInstance;
   }
}
```

# Exceptions

- You cannot throw or catch objects of generic type:

```
public class Problem<T> extends Exception { /* . . . */ }
   // Error--can't extend Throwable
```

- You cannot use a type variable in a catch clause:

```
public static <T extends Throwable> void doWork(Class<T> t)
{
   try { do work }
   catch (T e) // Error--can't catch type variable
   { Logger.global.info("It didn't work"); }
}
```

# Inheritance and Subtype Relationships

- Manager is a subclass of Employee.
  - Is Pair<Manager> a subclass of Pair<Employee>?

- No, subtype relationship between **GenericType<Type1>** and **GenericType<Type2>**