Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Java Genericss

CS2013: Programming with Data Structures

Introduction to Generics

- Generics was introduced to Java in version JDK 1.5.

- Put simply, ***generics*** is a way to parameterize data types.

- What does ***parameterize*** mean?
  - Recall that a parameter in programming is a placeholder in a method signature that determines what type of data can be passed into a method.
  - Method parameters accept any *value* as long as that value is of the correct *type* specified by the parameter.
  - To parameterize a data type means that the data type itself becomes a parameter, which can accept multiple types of data.

An Example

- Prior to the introduction of generics, all of the data structures in Java stored things as the **Object** type.
  - Recall that **Object** is the parent class of all reference types in Java.

- Before generics, If you wanted to retrieve a specific object type from a data structure, you would have to cast it:

```
List cats = new List();
Cat cat1 = new Cat();
cats.add(cat1);

Cat cat2 = (Cat)cats.get(0); <-- Casting required
```

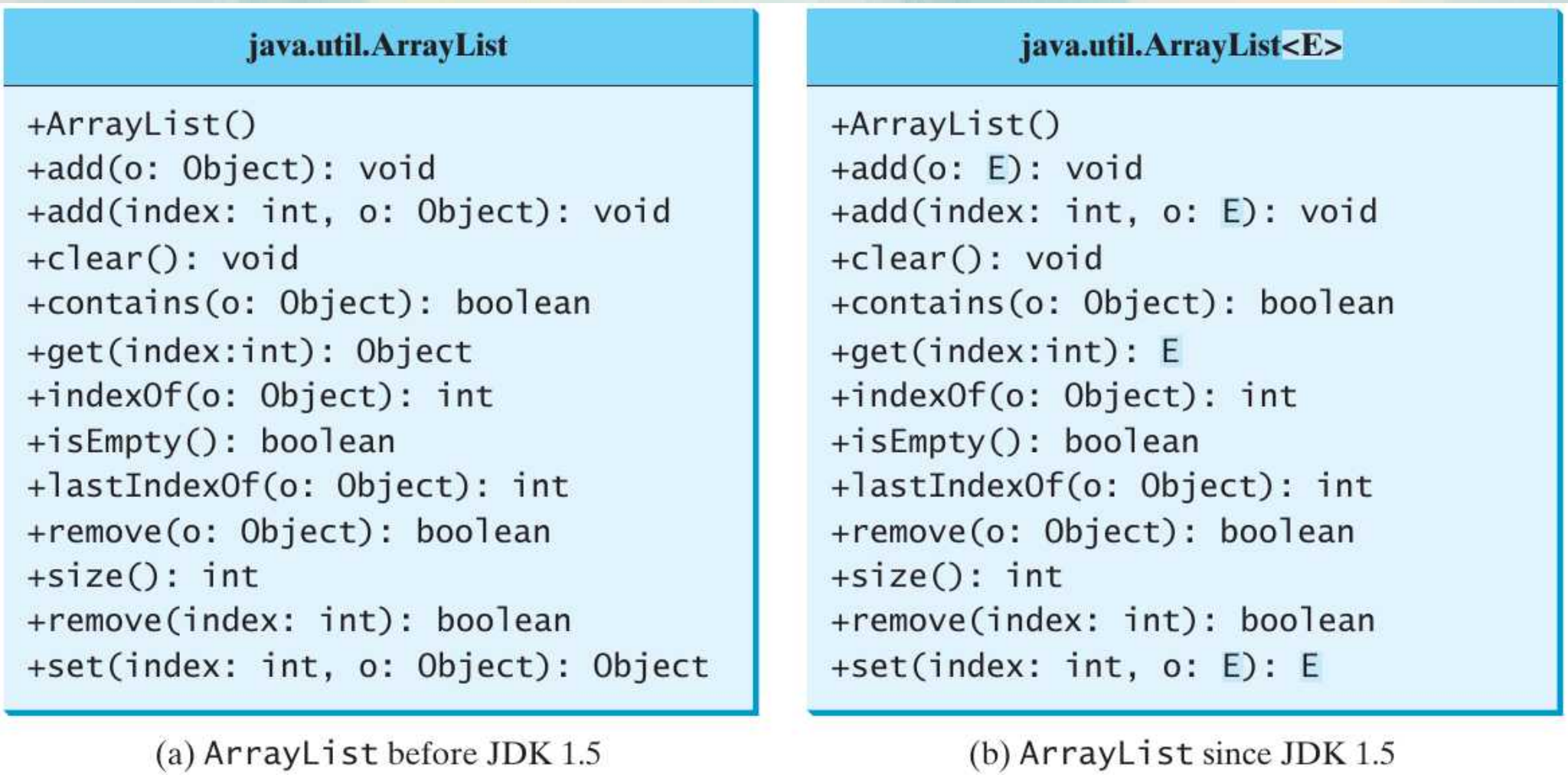
- With generics, no casting is necessary:

```
List<Cat> cats = new List<Cat>();
Cat cat1 = new Cat();
cats.add(cat1);

Cat cat2 = cats.get(0); <-- Casting NOT required
```

ArrayList Before and After Generics Comparison

- The following two UML diagrams illustrate how the ArrayList class was defined before and after generics was introduced.



(a) ArrayList before JDK 1.5

(b) ArrayList since JDK 1.5

ArrayList Before and After Generics Comparison

- Prior to generics you had to cast each element in the ArrayList to the type you were expecting.

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

- With generics you do not need to cast each element in the List because the compiler already knows the type.

```
ArrayList<Double> list = new
ArrayList<Double>();
list.add(5.5);
list.add(3.0);
Double num1 = list.get(0);
double d = list.get(1);
```

Main Advantages

- Type errors can be caught at **compile time** instead of during **runtime**.
  - Why are compile errors better than runtime errors?
- As mentioned previously, casting can be taken out of the equation when using generics.
- You can define one class for the data structure, and that data structure can accept objects of multiple types.
  - Instead of having to define multiple classes where each class can only accept one type of data.
- In addition to having generic data structures, you can also have generic algorithms which can work with collections of different types of data.

What Can Be Generic?

- You can define the following using generics:
  - Classes
  - Interfaces
  - Methods
  - Constructors

- NOTE: In the definition of the previous, you will use what I call "generic variables" but you cannot define variables "generically" by themselves. These variables MUST be defined inside of a generic class, interface, method, or constructor.

Generic Classes and Interfaces

Generic Classes and Interfaces

- A class or interface is "generic" if it has one or more *type variables*.

- A generic class or interface is defined with the following syntax:
  - `public class ClassName<T1, T2, ..., Tn>`
  - `public interface InterfaceName<T1, T2, ..., Tn>`
    - The generic type parameters are specified between a pair of angle brackets (< >) after the class or interface name.
    - T1, T2, …, Tn are called ***type parameters*** or ***type variables***.
    - NOTE: Multiple types parameters can be specified separated by commas. Each type must have a different name.

Generic Classes and Interfaces

- Examples:
  - `public class MyGenericClass<T>`
  - `public interface MyList<E>`

- Anywhere in your class you can use the type T or E and all T's will be replaced with the type that you specify when the class is **instantiated**.

- NOTE: Classes can define more than one type variable, separate each type in the angular braces with a comma:
  - `public class Pair<T, U>`

Type Parameters

- By convention, type parameters are a single uppercase letter and the letter that is used indicates the type being defined.
  - **<E>** for an element of a collection;
  - **<T>** for type;
  - **<K, V>** for key and value. (Dictionary / Map type data structures).
  - **<N>** for number
  - S,U,V, etc. for 2nd, 3rd, 4th type parameters

- NEVER give your generic types actual names, this will make your code very confusing to read.

How do we Instantiate a Generic Class?

- In the previous example we had two classes defined:
  - `public class MyGenericClass<T>`
  - `public class Pair<T, U>`

- When you instantiate each class you would do the following:
  - `MyGenericClass<String> mgc1 = new MyGenericClass<String>();`
  - `Pair<Integer, Double> mgp1 = new Pair<Integer, Double>();`

- After the instantiation, all instances of your generic data type are replaced with the concrete type that you specified.

How do we Instantiate a Generic Class?

- An alternative syntax was introduced in Java 7 which allows you to omit the Type between the <> on the right-hand side of the assignment statement. Note that the <> is still required.
  - `MyGenericClass<String> mgc2 = new MyGenericClass<>();`
  - `Pair<Integer, Double> mgp3 = new Pair<>();`
- In the previous examples, NOTE the use of the wrapper classes for the numeric types (**`Integer`**, **`Double`**, etc.). It is not possible to use the primitive versions of these types (**`int`**, **`double`**, etc).
  - Why is this? Think back to CS2012.

Parameterized Type

- A ***parameterized type*** is a type such as **`List<String>`**, that is, a class declaration that uses generics.

- Parameterized types can also be used in the instantiation of a class such as in the following:
  -
```
ArrayList<ArrayList<String>> complexList =
    new ArrayList<>();
```

- This may look strange, but it does work. This means that you basically have an ArrayList, where each element of that list is another list whose elements are all Strings.
  - Sound familiar? Where have you seen this concept before?

```
public class Pair<T, U> {
   private T item1;
   private U item2;

   public Pair() { /*body left empty*/}

   public Pair(T item1, U item2) {
      this.item1 = item1;
      this.item2 = item2;
   }

   public T getItem1() { return item1; }

   public void setItem1(T item1) { this.item1 = item1;}

   public U getItem2() { return item2; }

   public void setItem2(U item2) { this.item2 = item2;}
}
```

Generic Methods

Generic Methods

- ***generic methods*** are methods that have their own self-contained type parameters.

- The syntax is similar to declaring a generic class, but the generic type parameters in these methods are limited in scope to just the method in which they are declared.

- Note1: Most often used in static methods that are not tied to a particular instance of a class.

- Note2: Any methods in a class which use the type parameters defined by the class are also generic methods. These methods share the type parameters with the class and do not need to define their own.
  - A class method COULD define its own type parameters, but keep in mind that these parameters would be separate from the ones defined in the class header, and would be local in scope to just that method.

Declaring Generic Methods

- The general syntax to declare a generic method:
  - `public static <T1, T2, ... Tn> returnType methodName(parameter list)`
  - The type parameters are specified before the return type of the method and follow the same format and naming as with classes.
  - The parameter list of the method may contain concrete types (all the normal parameter types you are used to working with), in addition to any generic parameters specified in the angle brackets.
- Example:
  - `public static <T> void myGenericMethod(T item, int value)`

Generic Method Example

```
public class Util {
    public static <T, U> boolean comparePair(Pair<T, U> p1,
        Pair<T, U> p2) {

        return p1.getItem1().equals(p2.getItem1()) &&
               p2.getItem2().equals(p2.getItem2());

    }
}
```

- To invoke this method you would do the following:
  - `Util.<Integer, Double>comparePair(p1, p2);`
  - `Util.comparePair(p1, p2);`

- In the second invocation above, note the omission of the concrete types and the angle brackets. This is known as ***type inference***,
  - You can invoke a generic method as you would a regular method without specifying the type. The compiler will infer the correct type (if it can) based on the context.

Bounded Type Parameters

Bounded Type Parameters

- In defining generic classes or methods, there may be instances where you wish to restrict the types that can be used as type arguments
  - a method or class that only operates on numbers
  - a method or class that only operates on Strings
  - etc.

- To declare a bounded type parameter, in the angle brackets you would list the name of the parameter followed by the keyword **extends** and the **upper bound** type you wish to allow.
  - The use of extends here is similar to extending a class, but it is more general and works for both class and interface type bounds.
  - The upper bound is the upper most bound of a hierarchy of classes. Any type that is bounded can use the upper bound and any subclasses of that upper bound.

Generic Method with Bounded Types

```
public static <T extends Number, U extends Number>
    Pair<Double, Double> getPairAsDouble(Pair<T, U> p1) {

    Double item1 = p1.getItem1().doubleValue();
    Double item2 = p1.getItem2().doubleValue();

    Pair<Double, Double> doublePair = new Pair<>(item1, item2);

    return doublePair;
}
```

- This method will only accept a **`Pair`** object whose data types are both Number or a subtype of Number.
  - Note each type can extend a different class if necessary.

- By bounding a type, you have access to all methods within the bounded type.
  - Example, in the above you can use any method that is defined in the **`Number`** class.
  - You cannot use methods defined in subclasses of **`Number`**, Why is this?

Generics, Inheritance, and Subtypes

Generics, Inheritance, and Subtypes

- You should already know from CS2012, that it is possible to assign an object of one type, to an object of another type assuming that the types are compatible (i.e. one is a subclass of another.)

- Assume we have a parent class **`Animal`** with a subclass **`Cat`**.

- The following lines of code are valid:

```
Animal a1 = new Animal();
Cat c1 = new Cat("tabby");
a1 = c1; //Valid
```

- Remember that this is called an "is-a" relationship meaning that a Cat "is-a(n)" Animal.

Generics, Inheritance, and Subtypes

- The same concept also applies to generics, You can perform a generic type invocation with a parent class, and use any subclass of that parent as in the following:

```
Cat c1 = new Cat("maine coon");
Cat c2 = new Cat("siamese");

Pair<Animal, Animal> p1 =
    new Pair<>(c1, c2);
```

- Now for the tricky part, consider the following method:

```
public void m(Pair<Animal, Animal> p)
```

  - Does this method accept an argument of type **`Pair<Animal, Animal>`**?
  - Does this method accept an argument of type **`Pair<Cat, Cat>`**?

Generics, Inheritance, and Subtypes

- **`Pair<Animal, Animal>`** works!
- **`Pair<Cat, Cat>`**? does not!

- This is a very common misunderstanding of generics.
  - Although **`Cat`** is a subclass of **`Animal`**, **`Pair<Cat, Cat>`** is NOT a subtype of **`Pair<Animal, Animal>`**.

- Given two concrete class types **`A`** and **`B`**, **`MyClass<A>`** has no relationship to **`MyClass<B>`**, regardless of whether or not **`A`** and **`B`** are related. The common parent of **`MyClass<A>`** and **`MyClass<B>`** is **`Object`**.

Generic Classes and Subtyping

- You can subtype a generic class or interface by extending or implementing it.

- Lets look at the Java Collections classes for an example.
  - The Collections Classes are all the classes which define the main data structures built into Java.

- **`ArrayList<E>`** implements the **`List<E>`** interface, and **`List<E>`** extends **`Collections<E>`**.
  - This means that **`ArrayList<String>`** is a subtype of **`List<String>`** which is a subtype of **`Collections<String>`**.
  - As long as the type argument does not change, the subtyping relationship is preserved throughout.

Wildcards

Wildcards

- In generics there is a ***wildcard*** symbol which is the question mark (?).
  - This symbol can be used in various situations: the type of a parameter, data field, or local variable.
  - Can also be used as a return type, but this is very rarely if ever used.

- The wildcard is never used as a type argument for a generic method invocation, generic class instantiation, or a supertype.

Upper Bounded Wildcards

- Upper bounded wildcards can be used to relax the restrictions on a variable.
  - Remember previously when we said that **`List<Integer>`** is not a subtype of **`List<Number>`**.
  - A method defined as: **`public static void m(List<Number>)`** would accept an argument of type **`List<Number>`** but NOT an argument of **`List<Integer>`**.
  - What if we wanted to allow the second type of argument?

- An upper bound wildcard can solve this dilemma.

Upper Bounded Wildcards

- An upper-bounded wildcard is declared with a ? followed by the keyword extends and the upper bound.

- The previous example can be augmented to allow both types of arguments to work by doing the following:

```
public static void m(List<? extends Number>)
```

  - This method will now accept any type of List<A> as long as A is a subclass of Number.

Unbounded Wildcards

- Unbounded wildcards are types specified using ONLY the ?
  - Example: List<?> is a list of unknown type

- Where is this useful?:
  - Writing a method that can be implemented using functionality ONLY found in the Object class.
  - When the code is using methods in the generic class that do not depend on the type parameter
    - Example, List.size() or List.clear()

- Here is a method which can print a list of any type and uses the unbounded wildcard:

```
public static void printList(List<?> list) {
   for (Object elem: list) {
      System.out.print(elem + " ");
   }

   System.out.println();
}
```

Lower Bounded Wildcards

- Recall that the upper bound wildcard restricts the generic type to be an upper bound type or any subtype of that upper bound.

- A lower bounded wildcard is expressed using the ? character followed by the **`super`** keyword followed by the lower bound.
  - An example method:
    `public static void m(List<? super Integer>)`
  - **`List<Integer>`** would only accept the type **`List<Integer>`**, but...
  - This method will accept the types **`List<Integer>`**, **`List<Number>`**, or **`List<Object>`**.

Wildcards and Subtyping

- Recall: Generic classes or interfaces are not related just because there is a relationship between their types.

- Wildcards can be used to create a relationship between generic classes or interfaces.

- Lets say we have the following two normal (non-generic class definitions):

```
public class A {/* … */}
public class B extends A {/* … */}
```

- Then the following is valid:

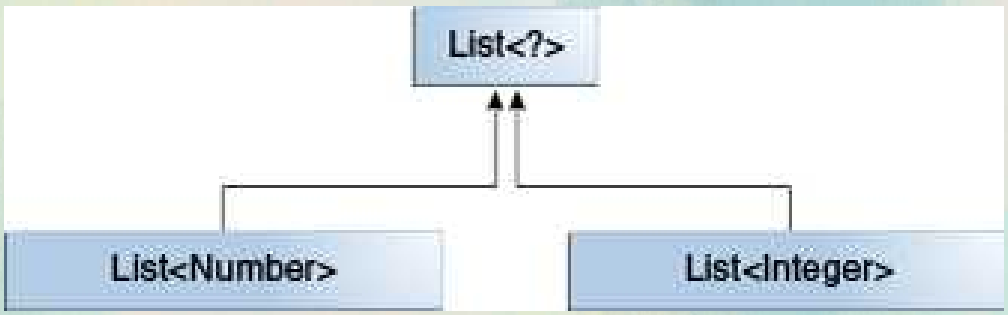
```
B b = new B();
A a = b;
```

- But remember that the inheritance rules of regular classes do not apply to generic types:

```
List<B> listB = new ArrayList<>();
List<A> listA = listB; // compile-time error
```

Wildcards and Subtyping

- Integer is a subtype of Number, what then is the relationship between **`List<Integer>`** and **`List<Number>`** ?
  - As stated before, **`List<Integer>`** is not a subtype of **`List<Number>`**
  - BUT **`List<Integer>`** and **`List<Number>`** are both subtypes of **`List<?>`**



- To create a relationship between these classes so the code can access **`Number`**'s methods through **`List<Integer>`**'s elements, use an upper bounded wildcard:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK
```

  - Since **`Integer`** is a subtype of **`Number`** and **`numList`** is a list of Number objects, we now have a relationship between **`intList`** and **`numList`**.

¿When Should I Use Wildcards?

- For the following discussion lets think of variables as serving two functions:
  - **An "IN" variable**:
    - offers data to the code
    - consider a method **`copy(src, dest)`**, here **`src`** would be an "IN" variable providing data to be copied.
  - **An "OUT" variable**:
    - holds data for use elsewhere in the program
    - consider a method **`copy(src, dest)`**, here **`dest`** would be an "OUT" variable that holds the resulting data.
- The IN/OUT idea is helpful for deciding:
  - when to use a wildcard
  - what type of wildcard to use.

¿When Should I Use Wildcards?

- General Guidelines:
  - If a variable is an "IN" variable, define an upper bounded wildcard using the **`extends`** keyword.
  - If a variable is an "OUT" variable, define a lower bounded wildcard using the **`super`** keyword.
  - If an "IN" variable only needs methods defined in the **`Object`** class, use an unbounded wildcard.
  - If a variable is both an "IN" and "OUT" variable, do not use a wildcard.

- NOTE: These do not apply to a method's return type, avoid using a wildcard as a return type.

¿When Should I Use Wildcards?

- This is also known as the **GET-PUT** Rule:
  - GET corresponds to IN type variables.
  - PUT corresponds to OUT type variables

- GET-PUT Rules:
  - Use an extends wildcard when you only **GET** values out of a variable / data structure.
  - Use a super wildcard when you only **PUT** values into a data structure.
  - Don't use a wildcard when you both get and put from/to a data structure

- Exceptions:
  - You can't put anything into a data structure declared with an extends wildcard except for the **null** value.
  - You cannot get anything out of a data structure declared with a super wildcard except for a value of type **Object**.

Example from StackOverflow

- Let's say you have a bunch of bananas and you define the bananas as:
  - **List<? extends Fruit>**
  - This is a collection of a particular kind of Fruit, but you don't know from the declaration, what kind of Fruit it is a collection of.
  - You can **get** an item from this collection and you know it will definitely be a fruit, but you can't add anything to this collection.
  - Why? Because you might be trying to add an apple to a bunch of bananas.

Example from StackOverflow

- Now, let's say we have a fruit bowl defined as:
  - **`List<? super Banana>`**
  - From the definition we know that this is a collection of some type of fruit "greater than" a banana (i.e. **`List<Fruit>`** or **`List<TropicalFruit>`**).
  - We can for sure add a banana to this bowl, but if we were to try to fetch a banana from this bowl we wouldn't know what we are getting. It could very well NOT be a banana.
  - All you know for sure is that it will be a valid Object reference.

Type Erasure

Type Erasure

- Generics use a technique known as ***type erasure***.
  - The compiler uses generic type information to compile the code, but erases / replaces those generic types afterward.

- The compiler applies type erasure to the following scenarios:
  - Replace all generic type parameters with their bounds (if you use super or extends) or with Object if the type is unbounded.
    - The resulting bytecode will only contain ordinary classes, interfaces, and methods.
  - Insert type casts if necessary to preserve ***type safety***.
    - type safety means you are not using any incompatible types that would cause a compile / runtime error.
  - Generate helper methods to preserve polymorphism in extended generic types.

- Type erasure ensures that no new classes are created for parameterized types.
  - Generics incur no runtime overhead as a result.

Type Erasure

- Example before and after type erasure with unbounded types.

BEFORE

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(
      T data,
      Node<T> next) {

        this.data = data;
        this.next = next;
    }

    public T getData(){}
}
```

AFTER

```
public class Node {

    private Object data;
    private Node next;

    public Node(
      Object data,
      Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() {}
}
```

Type Erasure

- Example before and after type erasure with bounded types.

BEFORE

```
public class
  Node<T extends Comparable<T>> {

   private T data;
   private Node<T> next;

   public Node(
     T data,
     Node<T> next) {

      this.data = data;
      this.next = next;
   }

   public T getData() { }
}
```

AFTER

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(
      Comparable data,
      Node next) {

        this.data = data;
        this.next = next;
    }

    public Comparable getData() { }

}
```

Generics Restrictions

Generics Restrictions

- **Cannot instantiate generic types with primitive types:**
  - Consider a class **`public class MyClass<T>`**
  - You cannot create an instance of this class using primitive types.
    - **`MyClass<int> mc1 = new MyClass<>();`** //Error
    - **`MyClass<char> mc2 = new MyClass<>();`** //Error
    - Any other primitive will also give an error.
  - You can only use the wrapper class versions of the primitives.
    - **`MyClass<Integer> mc1 = new MyClass<>();`** //OK
    - **`MyClass<Character> mc2 = new MyClass<>();`** //OK
    - Any other primitive will also give an error.

Generics Restrictions

- **Cannot create instances of type parameters:**
  - The following causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

  - A workaround is to create an object of a type parameter using reflection. (Reflection is a type of design pattern, Google it for your own edification).

```
public static <E> void append(
    List<E> list,
    Class<E> cls) throws Exception {

    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

Generics Restrictions

- Cannot use new **E[]**
  - cannot create an array using a generic type parameter
  - the following is **wrong!!!**:

```
E[] elements = new E[capacity];
```

- You can circumvent this by creating an array of Object type then casting it to E[]:

```
E[] elements = (E[]) new Object[capacity];
```

  - cast to (**E[]**) causes an unchecked compiler warning since the compiler has no idea if the casting will succeed or not.

Generics Restrictions

- **Cannot declare static fields generically**
  - A static data field in a class is shared by all non-static objects of that class.
  - Consider the following:

```
public class List<T> {
   private static T item;
}
```

  - If generic static data fields were allowed, the following lines of code would be ambiguous:

```
List<Cat> cats = new List<>();
List<Apple> apples = new List<>();
List<Book> books = new List<>();
```

  - Since the static field **item** is shared by all instances of this class, what is the actual type of item? It can't be **`Cat`**, **`Apple`**, and **`Book`** at the same time.
  - Generic static data fields are not allowed.

Generics Restrictions

- **Cannot use casts and instanceof with Parameterize Types:**
  - Recall that the compiler erases all type parameters in the generic code and replaces it with **`Object`** or a specified bound. You cannot verify which type for the generic is actually being used at runtime.
  - Example:

```
public static <E> m(List<E> list) {
   if (list instanceof ArrayList<Integer>) { //error
      //...
   }
}
```

  - Generally speaking, you cannot cast to a parameterized type unless it is parameterized with an unbound wildcard:
  - Example:

```
List<Integer> list1 = new ArrayList<>();
List<Number> list2 = (List<Number>)list1; //error
```

Generics Restrictions

- **Cannot create arrays of parameterized types:**
  - The following does not compile:
    - `List<Integer>[] arrayOfLists = new List<Integer>[2];`
  - This next example shows what would happen when different types are inserted into a normal array:

```
Object[] strings = new String[2];
strings[0] = "hi"; //OK
strings[1] = 1; //throws ArrayStoreException
```

  - Now here is an example with generics:

```
Object[] stringLists = new List<String>[]; //error
  (pretend it is allowed for this example)
stringLists[0] = new ArrayList<String>(); //OK
stringLists[1] = new ArrayList<Integer>();
  //ArrayStoreException should be thrown, but the
  runtime cannot detect it because of the generics.
```

  - If arrays of parameterized types were allowed, the code would fail to throw the correct ArrayStoreException error.

Generics Restrictions

- **Cannot create, catch, or throw objects of parameterized types.**
  - A generic class cannot extend the **Throwable** class either directly or indirectly.
  - Example, the following will not compile:

```
//Extends Throwable indirectly
public class AnimalException<T> extends Exception

//Extends Throwable directly
public class CatException<T> extends Throwable
```

  - A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void
 execute(List<J> jobs) {
   try {
      for (J job : jobs)
      // …
   } catch (T e) {   // compile-time error }
}
```

  - You can use a type parameter in a **throws** clause:

```
public class Parser<T extends Exception> {
   public void parse(File file) throws T { // OK  }
}
```

Generics Restrictions

- **Cannot overload a method where the formal parameter types of each overload erase to the same raw type:**
  - This makes sense if you recall how overloading works:
    - ```
      public class Example {
          public void print(Set<String> strSet) { }
          public void print(Set<Integer> intSet) { }
      }
      ```
  - Once the types were erased, the methods would have the same signature.
      ```

References

- Oracle Java Tutorial on Generics

- Liang, Introduction to Java Programming, Chapter 19 Generics

- Goodrich, Data Structures & Algorithms, Chapter 2.5.2