Generic Programming in Java
Generic Programming: an overview

• Generic: of, applicable to, or referring to all the members of a genus, class, group, or kind.

• We no longer have to create a new class for every new type of data.
  • If A, B and C are three different classes, and we want to create a linked list of As, Bs and Cs, we can write a single (generic) linked list class and instantiate all three using this one class.
  • In a generic class, the actual data type is a variable.
    • Think of elementary algebra. We don’t say \((2 + 3)^2 = 2^2 + 2 \times 2 \times 3 + 3^2\), and then \((3 + 7)^2 = 3^2 + 2 \times 3 \times 7 + 7^2\), ... and so on for every pair of numbers!
    • We use a single formula \((a + b)^2 = a^2 + 2ab + b^2\), expressed using variables, and then apply this formula when we actually have numbers.
  • Think of the actual objects (e.g., String, Circle) as numbers, and a generic class as a single formula. The type variable `<T>` is like the algebraic variables \(a, b, \text{ etc.}\)
Conversions

• Except for primitive types, any variable in Java is a reference (i.e., pointer) to an object.

```java
int x = 1;                 // the primitive int 1
Integer y = new Integer(1); // a reference to the ‘Integer’ object with value 1
```

• But, since any object in Java is a subclass of `java.lang.Object`, we can say something like

```java
int x = 1;                 // the primitive int 1
Integer y = new Integer(1); // a reference to the ‘Integer’ object with value 1
Object z = y;              // because ‘Integer’ is a subclass of ‘Object’
```

• This is a **widening conversion**.
  - Because the left-hand of the assignment is an object that is capable of referring to a wider variety of things than the right-hand side.
  - Java permits ALL widening conversions.
Conversions (contd.)

- What if we now want the variable $y$ to refer back to the original object?
  - After the last line, the original object has only one variable referring to it. The `java.lang.Object z`.

- But we cannot say $y = z$, because this is NOT a widening conversion. As far as Java compiler is concerned, $z$ could be any object!

```java
int x = 1;        // the primitive int 1
Integer y = new Integer(1); // a reference to the ‘Integer’ object with value 1
Object z = y;     // because ‘Integer’ is a subclass of ‘Object’

y = new Integer(2); // now, y refers to another ‘Integer’.
```

This is called **typecasting**, or simply **casting**. It is the programmer’s responsibility to ensure that $z$ really is an integer. Otherwise, Java will throw a `ClassCastException` during runtime.
Boxing and unboxing conversions

• We have seen primitive types and their wrapper classes.
• Java provides convenient ways to convert a primitive type to its wrapper class, and vice versa.

```java
int x = 42;
Integer u = new Integer(x); // boxing
int v = u.intValue();   // unboxing
Integer y = x;              // autoboxing
int z = y;              // auto-unboxing
```
Type Erasure

• Generic programming in Java is implemented using something called **type erasure**.
  
  • A mechanism that removes specific type information within the body of a generic class or method.
  
  • When we instantiate a generic class with an actual type
  
  ```java
  LinkedList<String> strings = new LinkedList<String>();
  ```
  
  we may think that every occurrence of the type parameter within <> gets by the actual type (in the above example, String).
  
  • But the Java compiler actually **erases** all the type information by replacing type parameters with `java.lang.Object`.
  
  • It then inserts typecasts wherever necessary to preserve the correct types.

• Why does it do this?
  
  • To ensure that no new classes are created for parameterized types. (Creating new classes would cause the code to slow down A LOT during runtime).
Erasure of generic types

```java
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() {
        return data;
    }
}
```

```java
public class Node {
    private Object data;
    private Node next;
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() {
        return data;
    }
}
```

Before type erasure | After type erasure
Erasure of generic methods

```java
public static <T> int count(T[] array, T t) {
    int count = 0;
    for (T x : array)
        if (x.equals(t))
            ++count;
    return count;
}
```

Before type erasure

```java
public static int count(Object[] array, Object t) {
    int count = 0;
    for (Object x : array)
        if (x.equals(t))
            ++count;
    return count;
}
```

After type erasure
Typecasting with arrays

// create an array of strings
String[] strings = new String[10];

// cast it to an array of objects
Object[] objects = strings;

// insert an object into the array
objects[0] = new Object();

This line will throw a very unique runtime error:
- java.lang.ArrayStoreException
Typecasting with arrays

// create an array of strings
String[] strings = new String[10];

// cast it to an array of objects
Object[] objects = strings;

// insert an object into the array
objects[0] = new Object();

• Remember, in Java, an array is also an object!
• If A is a subclass of B, you can put an instance of A into a variable of type B. For example, you can do
  • Object obj = new String(“s”);
  • We saw this in widening conversions.

• String[] is considered a subtype of Object[].  
  This is wrong!

Why?
• A is considered a subtype of B if and only if A fulfills all obligations of B. For example, a car is a subtype of vehicle because a car fulfills all the properties of a vehicle.
• You can put any object into an Object[], but you can only put a string into a String[].
Typecasting with arrays

// create an array of strings
String[] strings = new String[10];

// cast it to an array of objects
Object[] objects = strings;

// insert an object into the array
objects[0] = new Object();

• Remember, in Java, an array is also an object!
• If A is a subclass of B, you can put an instance of A into a variable of type B. For example, you can do
  • Object obj = new String(“s”);
  • We saw this in widening conversions.

• String[] is considered a subtype of Object[].

This is wrong!

Why?
• A is considered a subtype of B if and only if A fulfills all obligations of B. For example, a car is a subtype of vehicle because a car fulfills all the properties of a vehicle.
• You can put any object into an Object[], but you can only put a string into a String[].
• Therefore ...
Typecasting with generic collections

```java
public void test2() {
    // create a list of strings
    List<String> strings = new ArrayList<String>(10);

    // cast it to a list of objects
    List<Object> objects = (List<Object>) strings;

    // insert an object into the list
    objects.add(new Object());
}
```

This line will not compile. With generic collections, Java doesn't wait until you run the code. It throws a compilation error saying:
- Inconvertible types; cannot cast List<String> to List<Object>

Code doesn't even reach here.
Special behavior of arrays

• Arrays are older than generics.
• Arrays don’t support type erasure.

```java
public class GenericArrayTest<T> {
    // returns an array of the parameterized type
    public <T> T[] returnArray() {
        return new T[10];
    }
}
```

• Due to type erasure, this is not possible.
• Because at runtime, the type T does not exist!
• Throws a compile-time error saying:
  • Type parameter ‘T’ cannot be instantiated.

For more on this behavior, check out
• http://code.stephenmorley.org/articles/java-generics-type-erasure/