Java Generics

Java Generics

- Prior to the JDK 5.0 release, when you created a Collection, you could put any object in it.

```
List myList = new ArrayList(10);
myList.add(new Integer(10));
myList.add("Hello, World");
```

- Getting items out of the collection required you to use a casting operation:

```
Integer myInt = (Integer)myList.iterator().next();
```

Java Generics

- If you accidentally cast the wrong type, the program would successfully compile, but an exception would be thrown at runtime.

- Use instanceof to avoid a blind cast

```
Iterator listItor = myList.iterator();
Object myObject = listItor.next();
Integer myInt = null;
if (myObject instanceof Integer) {
   myInt = (Integer)myObject;
}
```

Java Generics

- J2SE 5.0 provides *compile-time* **type safety** with the Java Collections framework through ***generics***
- Generics allows you to specify, at compile-time, the types of objects you want to store in a Collection. Then when you add and get items from the list, **the list already knows what types of objects are supposed to be acted on**.
- So you don't need to cast anything. The "**<>**" characters are used to designate what type is to be stored. If the wrong type of data is provided, a compile-time exception is thrown.

Java Generics

- Example:

```
import java.util.*;
public class First {
  public static void main(String args[]) {
   List<Integer> myList = new
  ArrayList<Integer>(10);
   myList.add(10);  // OK ???
   myList.add("Hello, World"); // OK ???
  }
}
```

Java Generics

- If you don't specify the type of the collection your will get the following warning:

  ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

- If you want to get rid of the warning, you can add <Object> to the List declaration, as follows:

```
import java.util.*;
public class Second {
  public static void main(String args[]) {
   List<Object> list = new ArrayList<Object>();
   list.add(10);
  }
}
```

- Here, Object is the E, and basically says that any type of object can be stored in the List.

Java Generics

```
// Represents a list of values.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> {…
public class LinkedList<E> implements List<E> {…
```

Implementing Generics

// a parameterized (generic) class

public class **name**<**Type**> {

or

public class **name**<**Type**, **Type**, ..., **Type**> {

example:

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - You can require multiple type parameters separated by commas.

- The rest of your class's code can refer to that type by name.
  - The convention is to use a 1-letter name such as:
    T for Type, E for Element, N for Number, K for Key, or V for Value.

- The type parameter is *instantiated* by the client. (e.g. E → String)

Writing your own generic types

```
public class Box<T> {
    private List<T> contents;

    public Box() {
        contents = new ArrayList<T>();
    }

    public void add(T thing) { contents.add(thing); }

    public T grab() {
        if (contents.size() > 0) return contents.remove(0);
        else return null;
}
```

Writing your own generic types

```
public class Box<T> {
  private List<T> contents;
  private T sum;

  public Box() {
      contents = new ArrayList<T>();
  }

  public void add(T thing) {
     contents.add(thing);
     sum = sum + thing;
  }

  public T grab() {
      if (contents.size() > 0) return contents.remove(0);
      else return null;
}
```

What you will get if **T** is equal to 4?
What you will get if **T** is equal to A?
What you will get if **T** is student object?

Bounded Type Parameters

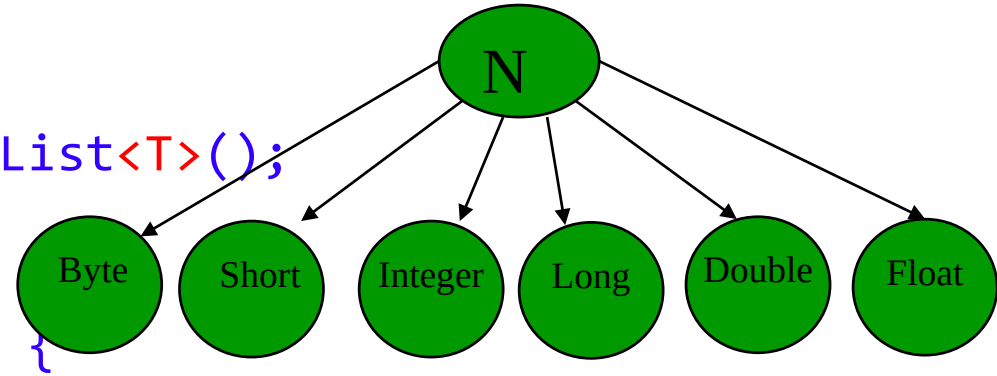
```
public class Box<T extends Number> {
  private List<T> contents;
  private T sum;

  public Box() {
      contents = new ArrayList<T>();
  }

  public void add(T thing) {
     contents.add(thing);
     sum = sum + T;
  }

  public T grab() {
      if (contents.size() > 0) return contents.remove(0);
      else return null;
}
```

You can pass number or any of it's children.



What you will get if **T** is equal to 4?
What you will get if **T** is equal to A?
What you will get if **T** is student object?

Syntax for bounded Type

- `public class Test <T extends X> {`

- }

If X is an interface then we can pass its implementation classes

If X is class, then we can pass either the class itself or it's child

Syntax for bounded Type

- public class Test <T extends Number> {

  }

```
public class Main2ForBounding {
 √   Test<Integer> Arr = New Test<Integer>();
 √   Test<Double> Arr = New Test<Double>();
 X   Test<String> Arr = New Test<String>();
}
```

Syntax for bounded Type

- public class Test <T extends Collection> {

  }

```
public class Main2ForBounding {
 X    Test<Collection> Arr = New Test<Collection>();
 X    Test<List> Arr = New Test<List>();
 √    Test<Stack> Arr = New Test<Stack>();
 X    Test<Integer> Arr = New Test<Integer>();
}
```

Syntax for bounded Type

- What if we needed create generic class to sort numbers?

- ```
  public class Test <T extends Number & Comparable>{

  }
  ```
  ```

Syntax for bounded Type

- You can implement more than one interface!

- ```
  public class Test <T extends Number & Comparable & Collection>{

  }
  ```

The key role is to start with class and then interface.
  ```

Syntax for bounded Type

- ```
  public class Test <T extends Comparable & Number>{

  }
  ```

This is incorrect!
The role is to start with class and then interface
  ```

Syntax for bounded Type

- If book is a class
  - public class Book {
    Private String Book;
  - }
- Cane we do this?
  - public class Test <T extends Number & Book>{

    }

Java doesn't allow you to inherent from two classes