1. -Parameterized types enable you to create classes, interfaces, and methods in which the type of data on which they operate is specified as a parameter.

-The <T> syntax indicates that T is a type parameter or a placeholder for the actual type.

-Generics makes type casts automatic and implicit, and therefore add type safety.

-Generics work only with reference types, i.e. we can't use primitive type.

-Generic types differ based on their type arguments, i.e. even though both iOb and strOb are of type Gen<T>, they are references to different types because their type arguments differ.

2. -The general form of a Generic Class: class class-name<type-param-list> { // ...

-Syntax for declaring a reference to a generic class and creating a generic instance:

class-name<type-arg-list> var-name = new class-name<type-arg-list>(cons-arg-list);

3. Bounded Types:

-When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived.This is accomplished through the use of an extends clause when specifying the type parameter:

<T extends superclass>

-Bounded types are especially useful when you need to ensure that one type parameter is compatible with another.

Eg:

-<T extends Number> means that T must be a Number or a subclass of Number.

-<T, K extends T> means that the second type parameter must be the same as the first parameter or a subclass of the first parameter.

4. Problem: Given the NumericFns class: class NumericFns<T extends Number> { , assume that you want to add a method called absEqual() that returns true if two NumericFns objects contain numbers whose absolute values are the same. Furthermore, you want this method to be able to work properly no matter what type of number each object holds. For example, if one object contains the Double value 1.25 and the other object contains the Float value –1.25, then absEqual() would return true.

To create a generic absEqual() method, you must use another feature of Java generics: the wildcard argument. The wildcard argument is specified by the ?, and it represents an unknown type.

class NumericFns<T extends Number> {

T num;

// Pass the constructor a reference to a numeric object.

NumericFns(T n) {

num = n;

}

// Write a method to determine if the absolute values of two objects are the same.

boolean absEqual(NumericFns<?> ob) { <---------- Notice the wildcard

if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()) return true;

return false;

}

}

Here, NumericFns<?> matches any type of NumericFns object, allowing any two NumericFns objects to have their absolute values compared.

It is important to understand that the wildcard does not affect what type of NumericFns objects can be created. This is governed by the extends clause in the NumericFns declaration. The wildcard simply enables any valid NumericFns type to be matched.

5. -A bounded wildcard is especially important when you are creating a method that is designed to operate only on objects that are subclasses of a specific superclass.

-In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

<? extends superclass>

where superclass is the name of the class that serves as the upper bound.

6. -You can write a single generic method declaration that can be called with arguments of different types.

-Here is the syntax for a generic method:

<type-param-list> ret-type meth-name(param-list) { // ...

- Following are the rules to define Generic Methods −

 All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type, eg. public static < E > void printArray( E[] inputArray ).

 Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

 The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

Q. How to declare a generic method called myMeth() that takes one generic type argument and have it return an argument of that generic type?

Ans: <T> T myMeth(T o)

7. A constructor can be generic, even if its class is not. For example, in the following example, the class Summation is not generic, but its constructor is.

class Summation {

private int sum;

<T extends Number> Summation(T arg) { //A generic constructor

sum = 0;

for(int i=0; i <= arg.intValue(); i++)

sum += i;

}

int getSum() {

return sum;

}

}

8. -In general, if a class implements a generic interface, then that class must also be generic.

Eg: class MyClass<T> implements Containment<T> {

-The type parameter(s) specified by a generic interface can be bounded. This lets you limit the type of data for which the interface can be implemented.

Eg: If you wanted to limit an interface Containment to numeric types, then you could declare it like this:

interface Containment<T extends Number> {

-Now, any implementing class must pass to Containment a type argument also having the same bound. For example, now MyClass must be declared as shown here:

class MyClass<T extends Number> implements Containment<T> {

9. Some Generic Restrictions: