**Generics**

* **Why Use Generics**
  + Generics enable types to be parameters when defining classes, interfaces and methods.
  + The type parameters provide a way for you to reuse the same code with different inputs.
  + The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.
  + The code that uses generics has many benefits over non-generic code.
    - Stronger type checks at compile time
    - Elimination of casts
      * The following code snippets without generics requires casting

List list = new ArrayList();

list.add("hello");

String s = **(String)** list.get(0);

* + - * When re-written to use generics, the code does not require casting.

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* + - Enabling programmers to implement generic algorithms.
* **Generics Types**
  + A generic type is a generic class or interface that is parameterized over types.
  + A generic class is defined with the following format

class name<T1, T2, ..., Tn> { /\* ... \*/ }

* + The type parameter section, delimited by angle brackets follows the class name
  + It specifies the type parameters(also called type variable) T0,T1,……Tn

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

* + **Type Parameter Naming Conventions**
    - Type parameter names are single, uppercase letters
    - The most commonly used type parameter names are
      * E – Element (Used extensively by the Java Collections Framework)
      * K – Key
      * N – Number
      * T – Type
      * V - Value
  + **Invoking and Instantiating a Generic Type**
    - Generic Type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument

*Box<Integer> integerBox;*

* + - An invocation of a generic type is generally known as a parameterized type.

*Box<Integer> integerBox = new Box<Integer>();*

* + **The Diamond**
    - In Java SE7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments as long as the compiler can determine or infer the type argument from the context.
    - This pair of angle brackets<> is informally called the diamond.

Box<Integer> integerBox = new Box<>();

* + **Multiple Type Parameter**
    - Generic class can have multiple type parameters.

public interface Pair<K, V> {

public K getKey();

public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V> {

private K key;

private V value;

public OrderedPair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

* + - The following statements create two instantiations of the OrderedPair class.

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");

* + - As mentioned in the diamond, it can be written as

OrderedPair<String, Integer> p1 = new OrderedPair**<>**("Even", 8);

OrderedPair<String, String> p2 = new OrderedPair**<>**("hello", "world");

* + **Parameterized Types**
    - You can also substitute a type parameter with a parameterized type

OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));

* **Raw Types**
  + A raw type is the name of generic class or interface without any type arguments.
  + For example given the generic Box class

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

* + To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T.

Box<Integer> intBox = new Box<>();

* + If the actual type argument is omitted, you create a raw type of Box<T>

Box rawBox = new Box();

* + Therefore Box is the raw type of the generic type Box<T>
  + However a non-generic class or interface is not a raw type.
  + Raw types show up in legacy code because lot of API classes (such as collection classes) were not generic prior to JDK5.0
  + When using raw types you essentially get pre-generic behaviour

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

* + But if you assign raw type to a parameterized type, you get the warning

Box rawBox = new Box(); // rawBox is a raw type of Box<T>

Box<Integer> intBox = rawBox; // warning: unchecked conversion

* **Generic Methods**
  + Generic methods are methods that introduce their own type parameters.
  + This is similar to declaring a generic type but the type parameter scope is limited to the method where its declared.
  + Static and non-static generic methods are allowed, as well as generic class constructors.
  + The syntax for a generic method includes a type parameter, inside angle brackets and appears before the methods return type.
  + The Util class includes a generic method, compare which compares two Pair objects.

public class Util {

// Generic static method

**public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

public class Pair<K, V> {

private K key;

private V value;

// Generic constructor

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

// Generic methods

public void setKey(K key) { this.key = key; }

public void setValue(V value) { this.value = value; }

public K getKey() { return key; }

public V getValue() { return value; }

}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.compare(p1, p2);

* + This feature is known as type interference, allows you to invoke a generic method as an ordinary method without specifying a type between angle brackets.
* **Bounded Type Parameters**
  + There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.
  + For example a method that operates on numbers might only want to accept instance of Number or its subclasses. This is what bounded type parameters are for.
  + To declare the bounded type parameter, list the type parameters name followed by the extends keyword, followed by its upper bound.

public class Box<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

public <U **extends Number**> void inspect(U u){

System.out.println("T: " + t.getClass().getName());

System.out.println("U: " + u.getClass().getName());

}

Public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

integerBox.set(new Integer(10));

integerBox.inspect("some text"); // **error: this is still String!**

}

}

* + In addition to limiting the types you can use to instantiate a generic type.
  + Bounded type parameters allow you to invoke methods defined in the bounds.

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return **n.intValue()** % 2 == 0;

}

// ...

}

* + The isEven() method invokes the intValue() method defined in the Integer class through n.
  + **Multiple Bounds**
    - A type parameter can have multiple bounds.
    - <T extends B1 & B2 & B3>
* **Generic Methods and Bounded Type Parameters**
  + Bounded type parameters are key to the implementation of generic algorithms.
  + Consider the following method that counts the number of elements in an array T[] that are greater than a specified element elem.

public static <T> int countGreaterThan(T[] anArray, T elem) {

int count = 0;

for (T e : anArray)

if (e > elem) // compiler error

++count;

return count;

}

* + The implementation of the method is straightforward, but it does not compile because greater than (>) operator applies only to primitive types.
  + You cannot use > operator to compare objects.
  + To fix this problem use a type parameter bounded by the Comparable<T> interface.

public interface Comparable<T> {

public int compareTo(T o);

}

* + The resulting code will be

public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {

int count = 0;

for (T e : anArray)

if (e.compareTo(elem) > 0)

++count;

return count;

}

* **Generics, Inheritance and Subtypes**
  + It’s possible to assign an object of one type to an object of another type provided that the types are compatible.
  + You can assign an Integer to an Object, Since Object is one of Integer’s super types.

Object someObject = new Object();

Integer someInteger = new Integer(10);

someObject = someInteger; // OK

* + In object oriented terminology, this is called an “is-a” relationship.
  + Integer is also kind of Number, so the following code is valid as well.

public void someMethod(Number n) { /\* ... \*/ }

someMethod (new Integer (10)); // OK

someMethod (new Double (10.1)); // OK

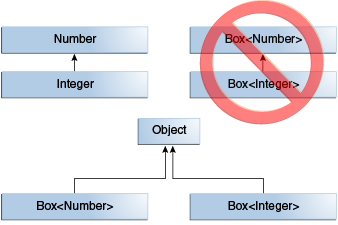
* + The same is also true with generics.
  + You can perform generic type invocation, passing Number as its type argument and any subsequent invocation of add will be allowed, if the argument is compatible with Number.

Box<Number> box = new Box<Number>();

box.add(new Integer(10)); // OK

box.add(new Double(10.1)); // OK

* + Box<Integer> is not a subtype of Box<Number> even though Integer is subtype of Number.



* + **Generic Classes and Sub Types**
    - You can subtype a generic class or interface by extending or implementing it.
    - The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.
    - Using the Collection classes as an example, ArrayList<E> implements List<E> and List<E> extends Collection<E>.
    - So ArrayList<String> is a subtype of List<String>, which is the subtype of Collection<String>.
    - The subtyping relationship is preserved between the types.
    - Now imagine we want to define our own list interface PayLoadList, that associates the optional value of generic type p with each element.
    - Its declaration might looks like

interface PayloadList<E,P> extends List<E> {

void setPayload(int index, P val);

...

}

* + - The following parameterizations of PayLoadList are subtypes of List<String>

PayloadList<String,String>

PayloadList<String,Integer>

PayloadList<String,Exception>

* **Type Inference**
  + Type Inference is a Java compiler’s ability to look at each method invocation and corresponding declaration to determine the type argument that makes the invocation applicable.
  + **Type Inference and Generic Methods**
    - Generic methods introduce you to type inference.
    - Which enables you to invoke generic method as you would an ordinary method, without specifying the type between angle brackets.

public class BoxDemo {

public static <U> void addBox(U u,

java.util.List<Box<U>> boxes) {

Box<U> box = new Box<>();

box.set(u);

boxes.add(box);

}

public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {

int counter = 0;

for (Box<U> box: boxes) {

U boxContents = box.get();

System.out.println("Box #" + counter + " contains [" +

boxContents.toString() + "]");

counter++;

}

}

public static void main(String[] args) {

java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =

new java.util.ArrayList<>();

BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);

BoxDemo.outputBoxes(listOfIntegerBoxes);

}

}

* + - To invoke the generic method addBox, you can specify the type parameter as follows.
      * BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    - Alternatively If you omit the type parameters, a java compiler automatically infers that the type parameter is Integer.
      * BoxDemo.addBox(Integer.valueOf(10), listOfIntegerBoxes);
  + **Type Inference and Instantiation of Generic Classes**
    - You can replace the type arguments required to invoke the constructor of generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context.
    - This pair of angle brackets is informally called the diamond.

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

* + - You can substitute the parameterised type of the constructor with an empty set of type parameters (<>).

Map<String, List<String>> my Map = new HashMap<>();

* + - In the following example the compiler generates an unchecked conversion warning because HashMap() constructor refers to the HashMap raw type.

Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning

* + **Type Inference and Generic Constructors of Generic and Non-Generic Classes**
    - Note that constructors can be generic in both generic and non-generic classes.

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

* + - Consider the following instantiation of the class MyClass

new MyClass<Integer> ("")

* + - This statement creates an instance of the parameterized type MyClass<Integer>.
    - The compiler infers the type String for the formal type parameter, T of the constructor of this generic class (Because actual parameter of this constructor is a String object).
    - However compilers in java SE7 and later infer the actual type parameter of the generic class being instantiated if you use diamond (<>).

MyClass<Integer> myObject = new MyClass<> ("");

* **Wildcards**
  + In Generic code, the question mark (?) called the wildcard represents an unknown type.
  + The wildcard can be used in a variety of situations
    - As the type of parameter
    - Field
    - Local variable
    - Sometimes as a return type.
  + The wildcard is never used as a
    - Type argument for a generic method invocation
    - Generic class instance creation
    - Or a supertype
* **Upper Bound Wildcards**
  + You can use an upper bound wildcard to relax the restriction on variables.
  + For example say you want to write a method that works on List<Integer>, List<Double> and List<Number>
  + You can achieve this by using an upper bounded wildcard.
  + To declare an upper bound wildcard, use the wildcard character (?) followed by the extends keyword, followed by its upper bound.
  + The term List<Number> is more restrictive than List<? extends Number> because former matches a list of type Number only, whereas the later matches a list of type Number or any of its subclasses.
  + The sumOfList method returns the sum of the number in a list.

public static double sumOfList(List<? extends Number> list) {

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

* + The following code using a list of Integer objects, prints sum = 6.0

List<Integer> li = Arrays.asList(1, 2, 3);

System.out.println("sum = " + sumOfList(li));

* + The list of Double values can use the same sumOfList method, The following code prints sum = 7.0

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);

System.out.println("sum = " + sumOfList(ld));

* **Unbounded Wildcards**
  + The unbounded wildcard type is specified using the wildcard character (?), for example List<?>.
  + This is called a list of unknown type.
  + Example:

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

* + You can use printList to print a list of any type.

List<Integer> li = Arrays.asList(1, 2, 3);

List<String> ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);

* **Lower Bounded Wildcards**
  + The upper bound wildcards section shows that an upper bound wildcard restricts the unknown type to be a specific type or sub type of that type and is represented using the extends keyword.
  + In a similar way, a lower bound wildcards restricts the unknown type to be a specific type or super type of that type.
  + A lower bound wildcard is expressed using the wildcard character (?) following by the super keyword followed by its lower bound <? super A>
  + You can specify an upper bound for a wildcard or you can specify a lower bound, but you cannot specify the both.
  + To write the method that works on list of Integer and the supertypes of Integer, such as Integer, Number and Object, you would specify <? super Integer>

public static void addNumbers(List<? super Integer> list) {

for (int i = 1; i <= 10; i++) {

list.add(i);

}

}

* **Wildcards and Subtyping**
  + You can use wildcards to create a relationship between generic classes and interfaces.

class A { /\* ... \*/ }

class B extends A { /\* ... \*/ }

* + It would be reasonable to write the following code.

B b = new B();

A a = b;

* + This rule does not apply to generic types.

List<B> lb = new ArrayList<>();

List<A> la = lb; // compile-time error

* + In order to create a relationship between these classes so that the code can access Number’s method through List<Integer>’s elements use an upper bound wildcard.

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>

* **Wildcard Capture and Helper Methods**
  + In some cases, the compiler infers the type of the wildcard.
  + For example a list may be defined as List<?> but evaluating an expression the compiler infers the particular type from the code. This scenario is known as wildcard capture.
* **Guidelines for Wildcard Use**
  + One of the more confusing aspects when to use an upper bound wildcard and when to use a lower bound wildcard.
  + For purpose of this discussion, it’s helpful to think of variables as providing one of two functions.
    - An “In” Variable
      * An in variable serves up data to the code.
      * Imagine a copy() method with two arguments copy (src, dest).
      * The src argument provides the data to be copied. So it’s the in parameter.
    - An “Out” Variable
      * An out variable holds data for use elsewhere.
      * In the copy example dest argument accepts the data. So it’s the out parameter.
  + Wildcard Guidelines
    - An “in” variable is defined with an upper bounded wildcard, using the extends keyword.
    - An “out” variable is defined with a lower bounded wildcard, using the super keyword.
* **Type Erasure**
  + Generics were introduced in java programming language to provide tighter type checks at compile time and to support generic programming.
  + To implement generics, the java compiler applies type erasure to
    - Replace all type parameter in generic types with their bounds or object if the type parameters are unbounded.
    - The produced byte code therefore contains only ordinary classes, interfaces and methods.
    - Insert type caste if necessary to preserve type safety.
    - Generate bridge methods to preserve polymorphism in extended generic types.
* **Erasure of Generic Types**
  + During the type eraser process, the java compiler erases all type parameters and replaces each with its first bound if the parameter is bounded or Object if the type parameter is unbounded.
  + Consider the following generic class that represents a node in a singly linked list.

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; }

// ...

}

* + Because the type parameter T is unbounded java compiler replaces it with object.

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; }

// ...

}

* + In the following example, the generic Node class uses a bounded type parameter.

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() { return data; }

// ...

}

* + The java compiler replaces bounded type parameter T with the first bound class Comparable.

public class Node {

private Comparable data;

private Node next;

public Node(Comparable data, Node next) {

this.data = data;

this.next = next;

}

public Comparable getData() { return data; }

// ...

}

* **Erasure of Generic Methods**
  + Java compiler also erases type parameters in generic method arguments.
  + Consider the following generic method.

// Counts the number of occurrences of elem in anArray.

//

public static <T> int count(T[] anArray, T elem) {

int cnt = 0;

for (T e : anArray)

if (e.equals(elem))

++cnt

return cnt;

}

* + Because T is unbounded, the java compiler replaces it with object.

public static int count(Object[] anArray, Object elem) {

int cnt = 0;

for (Object e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

* + Suppose the following classes are defined

class Shape { /\* ... \*/ }

class Circle extends Shape { /\* ... \*/ }

class Rectangle extends Shape { /\* ... \*/ }

* + You can write a generic method to draw the different shapes

public static <T extends Shape> void draw(T shape) { /\* .. \*/ }

* + The java compiler replaces T with shape.

public static void draw(Shape shape) { /\* ... \*/ }

* **Effects of Type Erasure and Bridge Methods**
  + Sometimes type erasure causes a situation that you may not have anticipated.
  + The following example shows how this can occur.

public class Node<T> {

private T data;

public Node(T data) { this.data = data; }

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

* + Consider the following code.

MyNode mn = new MyNode(5);

Node n = mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello"); // Causes a ClassCastException to be thrown.

Integer x = mn.data;

* + After type erasure the code becomes

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

* + **Bridge Methods**
    - When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a bridge method, as a part of type erasure process.
    - After type erasure the Node and MyNode classes become

public class Node {

private Object data;

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

public void setData(Object data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println(Integer data);

super.setData(data);

}

}

* + - After type erasure the method signature do not match
    - The Node method becomes setData(Object) and MyNode method becomes setData(Integer).
    - Therefore the MyNode setData method does not override the Node setData method.
    - To solve this problem, a java compiler generates a bridge method so ensure that subtyping work as expected.
    - For the MyNode class, the compiler generates the following bridge method for setData.

class MyNode extends Node {

**// Bridge method generated by the compiler**

**//**

**public void setData(Object data) {**

**setData((Integer) data);**

**}**

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

// ...

}

* **Non-Reifiable Types**
  + **Non-Reifiable Types**
    - A Reifiable type is a type whose type information is fully available at run time. This includes primitives, non-generic types, raw types and invocation of unbounded wildcards.
    - Non-Reifiable types are types where information has been removed at compile time by type erasure.
    - A non-Reifiable type does not have all of its information available at run time.
  + **Heap Pollution**
    - Heap pollution occurs when a variable of parameterized type refers to an object that’s not of that parameterized type.
  + **Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters**
    - If you declare a varargs method that has parameter of a parameterized type and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of that varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs method by adding the following annotation to static and non-constructor method declarations.
      * + @SafeVarargs
    - It’s also possible, though less desirable, to suppress such warnings by adding the following to the method declaration.
      * + @SupressWarnings({“unchecked”, ”varargs”})

* **Restrictions on Generics**
* **Cannot Instantiate Generic Types with Primitive Types**
  + Consider the following parameterized type

class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

// ...

}

* + When creating a Pair object you cannot substitute a primitive type for the type parameter K or V

Pair<**int, char**> p = new Pair<>(8, 'a'); // compile-time error

* + You can substitute only non-primitive types for the type parameter K or V

Pair<**Integer, Character**> p = new Pair<>(8, 'a');

* + Note that the java compiler autoboxes 8 to Integer.valueOf(8) and a to Character(‘a’)

Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

* **Cannot Create Instance of Type Parameters**
  + You cannot create an instance of a type parameter.
  + For example the following code causes a compile time error.

public static <E> void append(List<E> list) {

E elem = new E(); // compile-time error

list.add(elem);

}

* **Cannot Declare Static Fields Whose Types are Type Parameters**
  + A class’s static field is a class level variable shared by all non-static objects of the class.
  + Hence that static field of type parameters are not allowed.

public class MobileDevice<T> {

private static T os;

// ...

}

* + If static fields of type parameters are allowed, then the following code would be confused.

MobileDevice<Smartphone> phone = new MobileDevice<> ();

MobileDevice<Pager> pager = new MobileDevice<> ();

MobileDevice<TabletPC> pc = new MobileDevice<> ();

* + Because static field os is shared by phone, pager and pc, what’s the actual type of os?, It cannot be Smartphone, Pager or TabletPc at the same time.
* **Cannot Use Casts or instanceof With Parameterized Types**
  + Because java compiler erases all type parameters in generic code.
  + You cannot verify which parameterized type for a generic type is being used at run time.

public static <E> void rtti(List<E> list) {

if (list instanceof ArrayList<Integer>) { // compile-time error

// ...

}

}

* **Cannot Create Arrays of Parameterized Types**
  + You cannot create arrays of parameterized types.
  + For example the following code does not compile.

List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

* **Cannot create, Catch or Throw Objects of Parameterized Types**
  + A generic class cannot extend the Throwable class directly or indirectly.

// Extends Throwable indirectly

class MathException<T> extends Exception { /\* ... \*/ } // compile-time error

// Extends Throwable directly

class QueueFullException<T> extends Throwable { /\* ... \*/ // compile-time error

* + A method cannot catch an instance of type parameter.

public static <T extends Exception, J> void execute(List<J> jobs) {

try {

for (J job : jobs)

// ...

} catch (T e) { // compile-time error

// ...

}

}

* **Cannot Overload a method Where the Formal Parameter Types of Each Overload Erasure to the Same Raw Type**
  + A class cannot have two overloaded methods that will have the same signature after type erasure.

public class Example {

public void print(Set<String> strSet) { }

public void print(Set<Integer> intSet) { }

}