PONTOS CHAVES K&B
HTTP://camilolopes.wordpress.com

Capitulo 2 - OO

Encapsulamento

- permite reutilização

- reuso

- ocultar a implementação

Subscrição/sobrecarga

- métodos são subscritos

- variáveis não são

- static,private não são herdados

- static chamado pela referencia

- sobrecarga métodos com mesmo nome porém tipos + args diferentes.

- método com o modificador final é herdado +
não pode ser subscrito

```
public class X {
final void H(){}
      public static void
main(String[] args) {}
}
class Y extends X{
      //metodo da superclass foi
herdado
      static void z(){}
}
```

```
-------------------------------
```

```
public class X {
 void H(){}
      public static void
main(String[] args) {}
}
class Y extends X{
      //posso mudar o
modificador da subclasse
      static void z(){}
      final void H(){}
}
```

Herança

- É-UM

- polimorfismo – muitas formas

Conversão

- redutora – do pai para o filho

- ampliadora – do filho para o pai

"o pai não pode ser filho, mais o filho pode ser pai"

```
public class Do {
      public static void
main(String[] args) {
            Dat d = new
Pit();//valido
            Pit p = new Dat();
//invalido
      }}
class Dat{}
class Pit extends Dat{}
```

```
----------------------------
```

```
public class Fruta {
      public static void
main(String[] args) {
            Fruta f = new
Maca();//valido
            Maca = new Fruta();
// invalido
      }
}
class Maca extends Fruta{}
```

Tipo de retorno

- posso retornar qualquer tipo que possa ser convertido implicitamente para o tipo

- retorno covariantes serve apenas para objeto. Com primitivos o código não compila

- o tipo de retorno de um método subscrito não pode ser alterado com execao do retorno covariantes.

```
public class Casa {
int num(){
      short s =4;
      return s; // valido
}
```

Construtores

- são sobrecarregados

- aceita todos os modificadores de acesso

- não pode ser final(aplicado apenas à : classes, métodos e variáveis)

- não pode ser static(aplicado à: variáveis e métodos e blocos)

- pode ter this e super dentro de um construtor desde que um chame o construtor e o outro se refere a uma variável de instancia

- classes abstract tem construtores + não pode ser instanciada.

- interface não tem construtores não faz parte da herança

```
public class Constr {
      int z;
 protected Constr(){}
 private Constr(int
z){}//sobrecarregado
 public Constr(double a){}
      public static void
main(String[] args) {
      }}
class ConstSu extends Constr{
      public ConstSu() {
            super();
            this.z=8;
      }}
```

Acoplamento e Coesão

- Acomplamento – define o grau que se conhece de outra classe.
desejável: fraco (baixo)

- coesão – verificar se sua classe está bem focada, com seu papel bem definido.
desejável: forte (alto)

Static

- os métodos não são herdados

- é compartilhado para toda a classe

- só pode chamar outro membro static diretamente.

- não pode ser chamados por nem membro não static.

Capitulo 3 – Atriuições

tipos primitivos

- boolean o valor padaro – false

- int – 0,double - 0.0

- float termina com F ou f.

Escopo

- refere-se tempo de vida de uma variável

- static → dura mesmo tempo que a classe

- instancia → dura mesmo tempo que os objetos

- local → dura mesmo tempo que o método

- bloco → dura em relação a conclusão do bloc.

Array

- na declaração recebe null e não o objeto em si.

- os primitivos recebem o valor padrão (0,0.0)

- no fundo array são objetos

- são inicializados automaticamente se for declarado como local

- usar um objeto array null lança um exccelçao, mais o código compila.

- incluor negativo na construção do array, lança ma exceção

- deve ter o tamanho quando é constrido

- variável length – traz a quantidade de elementos no array.

```
public class Fanta {
String s[];
String k[] = new
String[2];//null
int z[];//null
int[] b = new int[1]; //0
Integer[] y = new
Integer[1];//nul
      public static void
main(String[] args) {
      }}
```

Atribuições

- posso atribuir um float a um double, um int a long.

- atribuir do maior para o menor somente com *cast* .

Bloco de inicialização

- os staticos executa 1º

- o da instancia quando dar um new, e antes dos contrutores e depois dos construtores.

- se tiver dentro do método não executa, caso o método não for chamado.

Wrapper

- converte de primitivos para objeto

- XXValue() – objeto para primitivo

- parseXX() – String para primitivos.

- valueOf() – String para objetos

- os métodos que recebem argumentos lança uma exceao não verificada NumberFormatxception e eses método são static.

Boxing

- == funciona ate 127 – true

- não posso == com tipo diferente não compila

- classes finais: Integer, Long, Double...

- argumento aceita o tipo e string exceto Character.

- nao posso converter de Integer para Long

- posso usar varags

- ampliação é preferida que autoboxing.

```
public static void main(String[]
args) {
	Float f=new Float(10.0f);
Double d = new Double(10.0);
//tipos diferentes Float não é
um Double
	System.out.println(f==d);
		}
```

Coleta de Lixo

- excluir objetos perdidos

- não existe nenhum algoritmo

- jvm que controla sua excecucao

- o programador pode solicitar sua execução ,porem não pode força-la.

- **OutofMemoryError** – falta de memoeria, ´[e garantido que o coleto tenha executado antes desa exceção.

```
public class Stexcecao {

	public static void
main(String[] args) {
String s = "Lopes";
String n=s;
/* um  objeto perdido "lopes"*/
	}
}
```

```
public class Box {
static Integer t =10;static
Integer k=10;
	public static void
main(String[] args) {

	System.out.println(t==k);
//true



	System.out.println(k==10);
//true
	}}
```

```
public class Box2 {
void net(int...g){}
void net(Integer...a){}
	public static void
main(String[] args) {
/* compila nao tem problema
nenhuma nos
 * metodos acima
 */
	}}
```

Capitulo 5 – Fluxo,Exceções Assertivas

If e Switch

- só aceita booleano para teste

- switch cada case deve ter uma string literal ou final

- não é permitido usar String e class Wrappers nos cases. E String na condição.

- o valor da condição switch é: byte,short,int,char e enum.

- não usar prefixo do enum no case

- não pode ter case duplicado.

```
      public static void
main(String[] args) {
            String s="milo";
            switch(6){
            case "milo":
            /* nao compila por
ser uma String*/

            }}}
```

Loops

- o for sei quantas vezes vai ser executado

- for aprimorado – usado para array/conjuntos

- uma variável declarada dentro do for não pode ser acessada fora.

- a incrementação somente acontece após a execução da instrução for.

- break – para a instrução. Usado apenas dentro de um loop ou switch.

- continue – ele salta a próxima instrução. So pode ser executada dentro de um loop.

```
      public static void
main(String[] args) {
            for(int
x=0;x<2;++x){

      System.out.println(x);
                  if(x==1)

      continue;

      System.out.println(x);//so
qdo if é false
            }
      }

      public static void
main(String[] args) {
            if(7==7){
System.out.println("lopes");
                  break;
System.out.println("neto");
/* codigo não compila pq o break
nao estar dentro de um  * loop
                  */}}
```

Assertivas

- serve para depuração

- se for false lança uma exceção

- -ea/-da – habilita/desabilita comando java

- compila na versão desejada: javac -source 1.3

- até a versão 1.3, assert é um identificador

- o segundo args de assert, se for vazo e tiver um tipo de retorno void o código não compila.

- não usar para testar métodos públicos

- usar para métodos private

- execução não é garantida

- não pode provocar efeitos colaterais.

Manipulação Exceções

- temos as verificadas e não verificadas

- as verificadas precisam ser tratadas/declaradas.

- Exception: e suas derivadas são verificadas

- as que derivam de **RuntimeException** não são verificadas.

- em cada catch sempre usar as classes mais especializadas

- Error – são exceções mais não são do tipo Exception

- **Error** – não precisa ser tratado ou declarado

- **Throwable** – pai das exceções

- se System. exit(1) for lançado todo o código para de executar – jvm é encerrada

- **finally** – é sempre executado independente de ocorrer ou não uma exceção.

Capitulo 6 – API

String/StringBuilder/Buffer

- são imutaveis – String

- nao pode ser extends.

- String e seus métodos: chatAt(int z) ,length(),toString(),concat(), trim(),toUpperCase(), toLowerCase(),substring()

-StringBuilder – roda mais rápido que a builder devido a idéia de não ter métodos não sincronizados

- builder/buffer: append(),insert(),delete(),reverse().

- String – implementa o metodo equals()

- Builder/Buffer – não implementa o método equals.

E/S

- manipulação de arquivo

- File – não cria o arquivo, é o caminho a ser criado.

- isDirectory – verifica se existe.

- BufferedWriter – escreve dentro do arquivo e lança uma exceção. Deve ser declarado dentro de try/catch caso o método não declare a exceção

- métodos BufferedWriter: write(), newLine().

- close fecha o arquivo

- flush – libera os recursos de writer, presente apenas nas classes Writer.

- PrintWriter – construtor aceita File,String e Writer e cria automaticamente o arquivo, mais se for diretório ele não cria de forma automática.

- mkdir() – cria o diretório

- se FileWriter(file,true/false) informa que a string é para adicionar no final do arquivo.

```
public static void main(String[] args) throws Exception{
      File f = new File("tata");
System.out.println(f.exists());

            PrintWriter pw = new PrintWriter(f);//aqui nao aceita true
            FileWriter fw = new FileWriter(f,true);
      }
```

DATAS/NUMERO/MOEDA

- as classes Date e Locale usa **new** para criar instancia.

- As classes NumberFormat,DateFormat,Calendar usa métodos de fabrica parar criar instancia.

- **NumberFormat** usa getDateInstance/getInstance para formatar a data : Short, Long, Medium e Full (0),Medium(2), Long(1,),Short(3)

- posso usar no contrustor de NumberFormat/DateFormat um objeto Locale.

- Locale (língua,pais) crio formatações especificas de internalização

-NumberPercentFormat – traz em percentual o resultado.

REGEX

-metacaracteres \\d,\\s,\\w e . (ponto)

- Pattern.compile(“ “) – é o que quero

- start() – traz a posição inicial do elemento

- group() traz o elemento

-token – é o que quero. O delimitador é o que não quero

-String.split("\\d") – não quero digito

-\d – erro de compilação

```
public static void main(String[] args) {
		String s="cam23aa";
		//retorna um array de string
		String[] p = s.split("\\d");//exclua os digitos
		for(String sa : p){


	System.out.println(sa);

------------

public static void main(String[] args) {
		Pattern pt = Pattern.compile("aaa");
		Matcher m = pt.matcher("caaameaa");
		boolean b;
		while(b=m.find()){

	System.out.println(m.start());
				//imprime 1 devido começar a combinacao nessa posicao
		}
```

FORMATACAO

- pertence a class PrintStream

- se usar strings de formatação errada o codigo lança uma exceção.

-string %s é valido para qualquer formato

-%b – boolean, %c char, %d digito, %f, float/Double

-printf e format tem as mesmas funcionalidades

```
	public static void main(String[] args) {
		int d=10;
		float f=10.0f;

	System.out.printf("%s", f);//valido

	System.out.printf("%d",f); //excecao
			}
```

Serialização - Serializable

- Se a superclasse implements Serializable então todas as subclassses é serializable

- se uma classe não implementar serializable os objetos não pode ser serializados

- A interface Serializable não tem nenhum método a ser implementado

- se a classe tem um objeto de uma classe que não implementa Serializable uma exceção ocorre

- variáveis static não são serializadas pois elas pertencem a classe

- variáveis instancia são serializadas pois elas pertencem a um objeto

- Externalizable extendss Serializable

- Externalizable tem dois métodos readExternal e writeExternal.

Capitulo 9 – THREADS

Threads

-posso criar extends a class thread

- implements a interface Runnable

-metodos da class Thread: start(),yield(),Sleep(),Join()

- run(), da interface Runnable

- se chamar start duas vezes no mesmo objeto thread ocorre uma exceção

- para executar a thread da subclasse deve subscrever o método public void run()

- para executar a thread vc sempre vai precisar de um objeto da classe thread.(lembra que start() pertence a essa classe?)

#ESTADO THREAD

– quando se dar NEW cria-se a instancia da classe thread o objeto está no estado novo

- quando dar um start(), a thread fica no estado executavel esperando ser colocado em execução

- execução – é quando o agendador de thread decidiu chama-lo por algum motivo

- uma thread com o estado inativo é porque o método run() já concluiu sua tarefa

- **suspensão** - thread fica dormindo porem quando acorda nada garante que ela vai ser executada de imediato

#Sleep,Yield,Join

-Sleep – diz a thread o seguinte: "*fique sem executar por long tempo"*

- esse método é Static e lança uma exceção verificada InterruptedException caso for interrompido

-Yield - o Maximo que ele consegue fazer é tirar a thread de execução para executável porem a thread deve ser da mesma prioridade. A execução não é garantida. É um método Static por que não posso especificar qual thread quero tirar de execucacao

- Join – digo que a oura thread(main por exemplo) so vai executar depois que a thread do join( )- b.join() concluir. O tempo em long dizendo que é o tempo Maximo de espera. No-Static esses método – precisa de uma instancia

- Start() – digo que a thread já está apta a ser executada. Se der dois Start() ao mesmo objeto thread uma excecao não verificada ocorre **IllegalThreadStateException.**

#SINCRONIZAÇÃO – SINCHRONIZED

- se eu der um Sleep/Yield dentro de bloco sincronizado os bloqueios não são liberados, ou seja, o Sleep não acontece.

#NOTIFICACAO/ESPERA

-os métodos wait/notify manda uma thread parar de executar e ser notificada

- wait()/wait(long s) - lança uma exceção verificada**(InterrupedException)** e mando a thread aguardar alem disso deve ser executada dentro de um contexto sincronizado,senão uma exceção ocorre.

IllegalMonitorStateException() – não verificada

- notify() – avisa a uma das thread que está no pool de espera para executar.

- o objeto notify() tem que ser o mesmo do contexto sincronizado.

- notify()/notifyAll – lança uma exceção não-verificada se for chamado fora de um contexto sincronizado

- notifyAll() – notifica todas as thread no pool

- nao há garantia que a thread que está mais tempo no pool é que vai ser executada isso depende do agendador

- posso ter varias threads do mesmo objeto no pool aguardar uma notificação

- quando a linha alcança wait() o bloqueio é liberado. Se informar o tipo wait(2000) ele espera pelo tempo informado depois ele readiquire o bloqueio do objeto.

#THREAD

-O construtor da class thread pode ser sobrecarregado das seguintes formas

Thread();

Thread(objteto run,String);

Thread(String s);

-o metodo **isAlive()** é final e retorna um valor Boolean se a thread ainda é ativo. Esse método diz se o método run() já concluiu ou não.

- não se pode sincronizar variáveis quando o tipo primitivo – erro de compilação

- notify – execução - (Running)

- NORM,MAX,MIN_PRIORITY() – configura a prioridade de uma thread.

- wait – executável(Runnable)

- dois objetos thread pode acessar o mesmo código sincronizado se os objetos de destino for diferente.

```
public class ObjtDif extends
Thread{
	public void run(){
		chat();	}
	synchronized void chat(){

	System.out.println(Thread.
currentThread().getName());
		try{

	Thread.sleep(200);//aqui
ele suspende a thread em
execução

	}catch(InterruptedExceptio
n e){e.printStackTrace();}
System.out.println(Thread.curren
tThread().getName());
	}
public static void main(String[]
args) {
/* dois objetos thread porem com
objetos
 * de destino diferente entao
bloqueio diferentes
	 */
ObjtDif od = new ObjtDif();
ObjtDif od2 = new ObjtDif();
		od.start();
		od2.start();}
}

----------------------------
public class SobreConstThread
implements Runnable {

	public void run(){
	System.out.println(Thread.
currentThread().getName());}

	public static void
main(String[] args) {
		SobreConstThread sc
= new SobreConstThread();
//	executa o run da class
Thread
Thread t = new Thread("th1");
Thread t2 = new
Thread(sc,"th2");
		t.start();
		t2.start();
```

```
//invalido devido a ordem no
construtor
Thread t3 = new Thread("th",sc);
      }}
-----------------
public class EsperaThread
implements Runnable{
public synchronized void run(){

System.out.println(Thread.curren
tThread().getName() + " "+

Thread.currentThread().isAlive()
);
      for(int i=1;i<10;++i)
            System.out.print(i);
      System.out.println();

//vai para a linha apos o wait()
      notify();}
public static void main(String[]
args)throws Exception{
      EsperaThread et = new
      EsperaThread();

Thread th = new Thread(et);
                  th.start();

      synchronized(th){
System.out.println("Aguarde...")
;
      th.wait();
//so imprime depois da
notificacao
System.out.println("camilo");
      }}}
-------------------------

public class PrioridadeThr
extends Thread{
      public static void
main(String[] args) {
PrioridadeThr pr = new
PrioridadeThr();

      pr.setPriority(12);

//lanca uma excecao valor
invalido

/* se a class implementa tenho
que
 * usar o nome da classe Thread
       */
}}
```

```
public class Pri implements
Runnable{
      public void run(){}
      public static void
main(String[] args) {
      Pri pr = new Pri();
      Thread t = new Thread(pr);
//como a classe nao extends
//a Thread tenho que colocar
Thread no
//args

      t.setPriority(Thread.NORM_
PRIORITY);
      }}
```

Capitulo 8 – Classes Internas

- É uma classe dentro de outra classe

- Posso acessar membros private da classe encapsuladora

- É aceito todos os modificadores de acesso(private,default,final, protected)

- Abstract e private é válido

- Tem um relacionamento especial com a classe

```
public class Exter {
private int x;
class interna{
   int x=1;
   void print(){
//interna

System.out.println(x);
//interna

System.out.println(this.x);
//Exter

System.out.println(Exter.this
.x);}}
   public static void
main(String[] args) {

Exter ex = new Exter();
Exter.interna ei = ex.new
interna();
   ei.print(); }}

----------
```

```
public class Bola {
private class ball{}
//modificador váliado
   }
```

#CLASSE INTERNA DO MÉTODO

- não pode usar as variáveis declaradas dentro do método

- as variáveis no args do método também não podem

- o modificador final permite acesso as variáveis locais do método

- abstract e final é o único modificador permitido para uma classe local do método, nunca os dois ao mesmo tempo

- fica dentro do método da classe encapsuladora

- Somente objeto da classe interna do método local pode instanciá-la

```
public class Metd {
      void clase(){
      class claMet{
            void pri(){
System.out.println("zoa");}
      }
      //instanciada dentro do
metodo
      claMet cm = new claMet();
            cm.pri();}
      public static void
main(String[] args) {
Metd me = new Metd();
      me.clase();
      }}
```

CLASSE ANÔNIMA

- não tem nomes

- Termina com ponto-e-vírgula ;

- posso subscrever método da superclasse

- só posso chamar o método de referencia e na os novos métodos da anônima

- as interfaces também podem ser anônimas e terminar com ponto-e-vírgula

```
public class Fish {
void fis(){}
      public static void
main(String[] args) {

Fish f = new Fish(){
void fis(){
System.out.println("anonimo");}
};
//imprime o da classe anomina
devido
//a subscricao
      f.fis();
      }}
```

CLASSE ANONIMA ARGS

- terminar com });

- posso usar uma interface no args

- posso usar uma classe no args

```
public class ArgVarg {
void  fac(face f){}
      public static void
main(String[] args) {

ArgVarg ar = new ArgVarg();

      ar.fac(new face(){

public void nome(){
System.out.println("interface");
      }
});//fim do args anonimo
/* n imprime nada pq o metodo
 * nome nao é chamado em momento
 * nenhum
 */
}}
interface face{
      void nome();}
```

STATIC CLASS – ANINHADA

- não tem relacionamento especial com a encapsuladora

- não pode acessar as variáveis de instancia

- não posso usar o this

- não precisa de um objeto da class encapsuladora para ser executada

- É-UM membro static da classe externa.

```
public class StaEst {
static int z=9;
static class Zad{
      int z;
//imprime da class static - 0
{System.out.println(z);}
}
      public static void
main(String[] args) {
//instanciacao
StaEst.Zad sz = new
StaEst.Zad();}}
```

```
public class StaDf {
int zoda;
static class Azd{
      {System.out.println(zoda);
}
/* nao compila static
 * acessando um membro non-
static
 */
}
```

```
public class InstanDiret {

static class Broma{
      Broma(){
System.out.println("Broma");
      }
      void br(){

System.out.println("metodo
static class");
      }}
      public static void
main(String[] args) {

Broma br = new Broma();
      br.br();
      }}
/* executa normalmente já que
uma static class – aninhada não
precisa de um instancia da
externa. */
```

Capitulo 7 - COLLECTION/GENERICS

CONTRATO EQUALS() E HASHCODE()

- Quando Implementar equals implementa também o hashCode para ter uma iteração eficiente em conjuto e array.
- O argument de equals() é Object e pode ser sobrecarregado
- Sempre chamar equals na classe que deseja comparar os valores
- Um valor único para hashCode é válido e executável porem ineficiente
- Os objetos de teste no equals deve ser usado no hashcode de uma forma ineficiente
- Se equals() == true então o hashCode deve ser true, como esta no contrato
- Se o equals() == false o hashcode pode ser tru mais false é o mais próximo do contrato.

-

```
import static
java.lang.System.out;

public class EqHash {

   public static void
main(String[] args) {

String s =  new
String("neto");
String n = "neto";

   out.println(s.equals(n));
   out.println(s.hashCode() +
" " + n.hashCode());
   }
/* observe objetos iguais
segundo o equals
 * por ter valores iguais e
tem o mesmo
 * codigo hashing
 */}
--------------------
public class InefHash {
//tem q ser public senao n
compila
public int hashCode(){
      return 10;}
      public static void
main(String[] args) {

InefHash ih = new InefHash();
InefHash ih2 = new InefHash();

      System.out.println(ih.equa
ls(ih2));
      System.out.println(ih.hash
Code() + " " + ih2.hashCode());
}

/* objeto diferente porem

 * mesmo codigo hashing maneira

 * ineficiente */}
```

CONJUNTOS

- É uma estrutura de dados onde só armazena objetos

- **Collection-** é uma interface que tem Set e List como derivadas

- **Collections** – é uma classe utilitária

- **Listas –** ordenada, repetição permitida por índice(ArrayList,LinkedList,Vector)

- **Conjuntos** - podem ou não ser ordenados e classificados, não aceita duplicata(HashSet, LinkedHashSet, TreeSet)

- **Mapas** – pode o não ser ordenados/classificados, por chaves, aceita e não aceita valores null e chave duplicata não é permitido (HashMap , LinkedHashMap, TreeMap HashTable)

- Filas – ordenação por fila, 1º chegar é o 1º a sair ou por ordem de priooridade(PriorityQueue, LinkedList)

- **ArrayList** – acesso aleatório

- **Vector** – acesso aleatório e método sincronizados

- **LinkedList** – acesso por ordem de inserção

- **HashSet** – não tem ordenação e elementos duplicados não são aceitos

- **LinkedHashSet** – acesso por ordem de inserção e não aceita elementos duplicados

- **TreeSet** – ordenada e classificada e não aceita elemento duplicado

- **hashNext()** – determina se existe mais elementos

- **next()** – traz o próximo elemento

- as classes de um Map deve seguir o contrato equals e hashCode para funcionar corretamente.

- quando usar filas **PriorityQueue:**
offer – adiciona, peek, mostra o 1º elemento, poll – mostra remove o 1 º elemento da fila

- a interface **Lang.Comparable** tem o método compareTo.

- a interface **util.Comparator** tem o método compare().

- para poder classificar um conjunto de elementos, deve ser do mesmo tipo, senão uma exceção ocorre.

- enum subscreve o contrato equals e hashCode.

```
public class ClassConjt {
	public static void
main(String[] args) {
List l = new ArrayList();
	l.add(new Integer(4));
	l.add(new String("s"));
	Collections.sort(l);
	}}
/* lança uma excecao por eu
mandar classificar objetos
 * diferentes*/
```

```
public class EnumHash {
enum estacao{
	PRIM,VERA,OUT,INVER	}

	public static void
main(String[] args) {

Set<estacao> es = new
HashSet<estacao>();

	es.add(estacao.OUT);

	es.add(estacao.PRIM);

	for(estacao c: es)

	System.out.println(c);
//es.add(OUT);//isso aqui nao
compila

//es.add(estacao.MES); tb nao
esse tipo nao tem La

/
	}}
```

UTILITARIOS

- o método sort() usa comparator ou a ordem natural

- **binarySeach**(Array/Conj, “element”) em array ou list.

- **asList** cria uma List a partir do array – Array.asList();

- **toArray()** – cria um array de uma lista.
List.toArray – retorna um Object, cuidado que pode nem compilar se não fizer o cast.
List.toArray(new tipo[0]); não precisa de cast

GENERICS

- refere-se ao tipo do conjunto em tempo de compilação

- não posso colocar tipo diferente

- o polimorfismo não funciona na instanciação

- em tempo de execução os tipos são apagados

- quando uso **<? >** digo que é somente para leitura o meu conjunto

- quando passo o conjunto sem tipo a segurança não existe mais

- <?> é diferente de **<Object>** já que o segundo é permitido add.

- Map é uma collection mas que não pertence a interface Collection

- não instancia uma class Map usando um tipo de Collection

- Genéricos pode ser definido para classes, método, variáveis.

- Para usar no args declaro antes do tipo e depois do nível de acesso: nível + <T> + tipo
static <t> void

```
public class Genrc<T> {
T numero;

static <T> void mecha(T k){
System.out.println(k);
}
	public static void
main(String[] args) {

Genrc<Double> gn = new
Genrc<Double>();

System.out.println(gn.numero);
	gn.mecha(gn.numero);
	}
/* ele nao imprime 0.0 pq nao se
trata
 * de tipo primitivos lembra que
aqui
 * tratamos tudo como objeto?
 * imprime null, null
 */
}
```

 # REVISAO

- não envolver numero randômico e transient para hashCode

- **String e Wrapper e Enum** já implementam equals() e hashCode() - **lang.Comparable → compareTo()**

- Comparator – compare() pode ser usado com TreeSet e TreeMap

- os método **element(),** e remove da classe **PriorityQueue** lança uma exceção se o Queue estiver vazio – **NoSuchElementException**

- Em genéricos não posso n em referir a tipos diferentes.

- null pode ser adiciona a qualquer collections mesmo se tiver <?>.

```
public class SeNull {
	public static void
main(String[] args) {
		Set<?> ss = new
LinkedHashSet();
			ss.add(null);
	}
/* null é valiado em qualquer
conjunto*/
}
```

Capitulo 10 – DESENVOLV.

 # Pacotes e Procura

- o exame é baseado em Unix /

- para incluir o pacote atual usa . (ponto)

java –cp com/Camilo ele procura nesse diretório o arquivo .class

java –classpath com/vm: . ele procura no diretório atual

 # Direcionando o .class

- o –d diz aonde vou colocar o .class

- se o diretório de destino não existir erro de compilação

javac –d ../classes com/arquivo.java

- ../classes – destino o código acima diz: pegue o arquivo que está em com/ chamado arquivo.java e coloque no diretori classes

- o .. diz que é para voltar um diretório do atual

- se “classes” não existir não compila

- se “classes” existir posso criar diretórios dentro dele

 # Proprieadades do Sistema

- para –Dvalor=valorsi minhaclasse

- mim traz os dados do sistema

- -D usar aspas quando tiver espaço

getProperty – obtem minha propriedade.

 # Main

- é flexivo

- static public void main(String...args)

- o codigo acima é valido.

- a arvore padrão de JAR é /jre/lib/ext